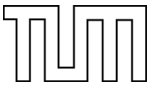


Technische Universität München

Department of Mathematics

Bachelor's Thesis

GAIO.jl: Set-oriented Methods for Approximating Invariant Objects, and their Implementations in julia

April Hannah-Lena Herwig
april.herwig@tum.de

Supervisor: Prof Oliver Junge

Submission Date: ...

I assure the single handed composition of this bachelor's thesis only supported by declared resources.

Garching,

Zusammenfassung

Bei einer in englischer Sprache verfassten Arbeit muss eine Zusammenfassung in deutscher Sprache vorangestellt werden. Dafür ist hier Platz.

Contents

Remark. Credit for cited images goes entirely to image authors. Where a cited image is used, see its citation for more information.

1 Dynamical Systems

1.1 Motivation

Our goal is to investigate the qualitative, long-term behavior of systems in which a given function describes the trajectory of a point in an ambient space. Such dynamical systems are used in modelling physcial phenomena, economic forecasting, differential equations, etc. We wish to construct topological *covers* of sets which describe the infinite-time dynamics of some portions of the system, as well as statistical *invariant measures* which describe much larger sets in the space, but with less information.

The basic technique of all the topological algorithms is to split a compact set Q into a partition $\mathcal{P}$ of *boxes* - that is, generalized rectangles $[c_1-r_1,\ c_1+r_1)\times\ldots\times[c_d-r_d,\ c_d+r_d)$, with center vector $c \in \mathbb{R}^d$ and componentwise radii $r \in \mathbb{R}^d$. The algorithms will begin with a set of boxes $\mathcal{B}$, and then repeatedly subdivide each box in $\mathcal{B}$ into two (or more) smaller boxes, examine the dynamics of the subdivided boxes, and refine the box set to include only the boxes we are interested in.

The algorithms described in the present paper have been previously implemented in the statistical programming language matlab [34], but is now being fully refactored and reimplemented in the open-source, composable language julia [9]. The reason for this change is julia's high-level abstraction capabilities, just-in-time compilation, and in-built set-theoretical functions, which create short, elegant code which is nonetheless more performant. Source code for GAIO in matlab and julia and be found in [25] and [26], respectively.

In the following, we assume M is a compact or a smooth manifold in $\mathbb{R}^d$, endowed with a metric d, and the map $f : M \to M$ is at least a homeomorphism. We further assume that P is a partition of a compact set $Q \subset M$ into (up to a Lebesque-null set) disjoint boxes, and $\mathcal{B} \subset \mathcal{P}$ is a subset of boxes. Our setting is a *discrete, autonomous dynamical system*, that is, a system of the form:

$$x_{k+1} = f(x_k), \quad k = 0,\ 1,\ 2,\ \ldots \tag{1.1}$$

A continuous dynamical system $\dot{x} = F(x)$ can be *discretized* by, for example, considering the *Poincaré time-t map* over some $d-1$ dimensional hyperplane, or by setting one "step" of the system as integrating F for a set time t.

1.2 Some Notation

Definition 1.1 (Image of a Box Set)**.** We will call the *image of* $\mathcal{B}$ *under* f the set of boxes which intersect with the image $f(B)$, for at least one $B \in \mathcal{B}$. More precisely, it is

$$f(\mathcal{B}) = \left\{ R \in \mathcal{P} \quad | \quad f^{-1}(R) \cap \bigcup_{B\in\mathcal{B}} B \neq \emptyset \right\}. \tag{1.2}$$

Theorem 1.2 (Image of a Box Set)**.** $f(\mathcal{B})$ is the inclusion-minimal cover of $f(\bigcup_{B\in\mathcal{B}} B)$ with boxes from $\mathcal{P}$.

Proof. We have the equivalent characterisation

$$f(\mathcal{B}) = \{R \in \mathcal{P} \quad | \quad \exists\ B \in \mathcal{B} \text{ and } x \in B \ :\ f(x) \in R\}\,. \tag{1.3}$$

Hence if we remove one box R from $f(\mathcal{B})$, then there exists an $x \in \bigcup_{B\in\mathcal{B}} B$ which maps outside of the created box set.

□

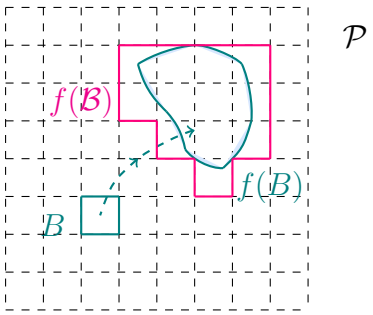


Figure 1.1: Image of the simple box set $\mathcal{B} = \{B\}$

Definition 1.3 (Diameter of a Box Set)**.** Denote by $d(\mathcal{B})$ the maximum diameter of a box $B \in \mathcal{B}$, ie $d(\mathcal{B}) = \max_{\substack{B\in\mathcal{B},\\ x,y\in B}} d(x,y)$.

1.3 Topological Definitions

Definition 1.4 ((Forward-, Backward-) Invariant)**.** [17] A set A is called *forward-invariant* if $f(A) \subset A$, *backward-invariant* if $f^{-1}(A) \subset A$, and *invariant* if it is both forward- and backward-invariant.

Definition 1.5 (Attracting Set)**.** [13] An invariant set A is called *attracting* with *fundamental neighborhood* U if for every open set $V \supset A$ there is an $N \in \mathbb{N}$ such that the tail $\bigcup_{k\geq N} f^k(U)$ lies entirely within V. The attracting set is also called *global* if the *basin of attraction*

$$B(A) = \bigcap_{k\geq 0} f^{-k}(U) \tag{1.4}$$

is the whole of $\mathbb{R}^n$.

The basin of attraction acts in some sense as the set for which all points eventually arrive in A. If the map f is smooth, then the closure $\bar{A}$ is invariant too. With continuity it becomes clear that

$$A = \bigcap_{k \geq 0} f^k(U). \tag{1.5}$$

The global attractor is maximal in the sense that it contains all backward-invariant sets within the system. In particular, it contains local unstable manifolds.

Definition 1.6 (Stable and Unstable Manifolds)**.** [40] Let $\bar{x}$ be a fixed point of the diffeomorphism f, and U a neighborhood of x. Then the *local unstable manifold* is given by

$$W^u(\bar{x}, U) = \left\{ x \in U \mid \lim_{k \to \infty} d(f^{-k}(x),\ \bar{x}) = 0 \text{ and } f^{-k}(x) \in U\ \forall\, k \geq 0 \right\}. \tag{1.6}$$

The *global unstable manifold* is given by

$$W^u(\bar{x}) = \bigcup_{k \geq 0} f^k(\, W^u(\bar{x}, U)\,). \tag{1.7}$$

The dual definition of the *(local) stable manifold* is obtained by reversing the sign of k in the above equations.

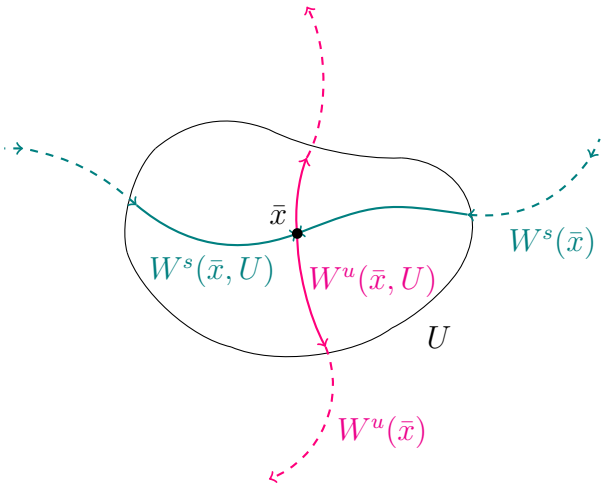


Figure 1.2: [24] Stable and unstable manifolds, local and global

Definition 1.7 (Pseudoperiodic)**.** [40] Let $n \in \mathbb{N}$. A set $\{x_k \mid k \in \{0,\ \ldots,\ n\}\}$ is called *ϵ-pseudoperiodic* if for any k, $d(f(x_{k \bmod n}),\ x_{k+1 \bmod n}) < \epsilon$.

As the name suggests, an ϵ-pseudoperiodic orbit is "almost" periodic in the sense that it represents a "small" perturbation of a theoretically periodic orbit. In practice, such pseudoperiodic orbits will not be known a-priori, but it will represent a naturally useful definition in our approximations.

Definition 1.8 (Chain Recurrent)**.** [40] The point $\bar{x} \in M$ is called *chain recurrent* if for any $\epsilon > 0$ there exists an ϵ-pseudoperiodic orbit containing $\bar{x}$. The *chain recurrent set* R_M is the set of all chain recurrent points in M.

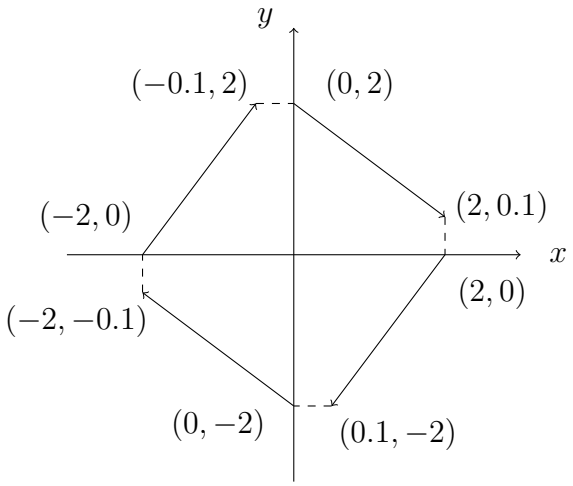


Figure 1.3: [40] A 0.1-pseudoperiodic orbit of the map $f(x, y) = (y,\ 0.05\,(1 - x^2)\,y - x)$. One can verify that $f(2, 0) = (0.1, -2)$, $f(0, -2) = (-2, -0.1)$, $f(-2, 0) = (-0.1, 2)$, $f(0, 2) = (2, 0.1)$.

1.4 Stochastic Definitions

Since our goal is to partition the manifold into a finite set of boxes, we must accept some amount of "uncertainty" in how our sets look, and how *exactly* f maps such a set. We describe this noise using a stochastic transition function.

Definition 1.9 (Transition Function)**.** [16] Let $\mathcal{A}$ be a σ-algebra on M. A function $p : M \times \mathcal{A} \to [0, 1]$ is called *transition function* if

1. $p(\cdot, A) : M \to [0, 1]$ is measurable for all $A \in \mathcal{A}$,
2. $p(x, \cdot) : \mathcal{A} \to [0, 1]$ is a probability measure for all $x \in M$.

Example 1.10.

- [16] We can model the deterministic system using the Dirac measure $p(x,\, A) = \delta_{f(x)}(A)$.
- The approximate box version of the system can be modelled as using a uniform probability density: for a point x let $\mathcal{B}(x) = \{B \in \mathcal{B} \mid x \in B\}$ be the (singleton) box set containing x. The define p as

$$p(x, A) = \frac{\mathcal{L}(\ A \cap f(\mathcal{B}(x))\ )}{\mathcal{L}(\ f(\mathcal{B}(x))\ )} \tag{1.8}$$

  where $\mathcal{L}$ represents the d-dimensional Lebesque measure.

Definition 1.11 (Perron-Frobenius Operator, Invariant Measure)**.** [16] Let p be a transition function, and μ a measure on M. We define the *Perron-Frobenius operator* as

$$(P\mu)(A) = \int p(x,\, A)\; d\mu(x) \tag{1.9}$$

A measure μ is called *invariant* if it is a fixed point of P.

Remark. The Perron-Frobenius operator is often also called *transfer operator.*

Example 1.12. [16] We calculate using the function from Ex. 1.10

$$(P\mu)(A) = \int \delta_{f(x)}(A)\; d\mu(x) = \int \chi_A(f(x))\; d\mu(x) = \mu \circ f^{-1}(A). \tag{1.10}$$

An invariant measure can be used to understand the global behavior of a dynamical system, with more μ-mass assigned to regions which are visited frequently over long trajectories, and less μ-mass to regions visited less frequently.

Our "noisy" approximated system poses the benefit that while deterministic dynamical systems generally support the existence of multiple invariant measures, the stochastic system will (if the transition function has a strictly positive density for each $x \in M$) have a unique invariant measure, as shown in [30].

Definition 1.13 (Almost Invariant)**.** [16] A set $A \subset M$ is called δ*-almost invariant* with respect to μ if $\mu(A) \neq 0$ and

$$\int_A p(x, A)\; d\mu(x) = \delta\mu(A). \tag{1.11}$$

Example 1.14. [16] Consider again the deterministic case $p(x, \cdot) = \delta_{f(x)}$. Then using the same calcuation as Ex. 1.12 we get

$$\delta = \frac{1}{\mu(A)} \int_A p(x, A)\; d\mu(x) = \frac{\mu(A \cap f^{-1}(A))}{\mu(A)}. \tag{1.12}$$

Theorem 1.15 (Hahn-decomposition)**.** [3] Let $(M, \mathcal{A}, \mu)$ be a measure space on M, where μ is any signed measure. Then there exists $E^+ \in \mathcal{A}$ with

$$\mu(E \cap E^+) \geq 0, \quad \mu(E \setminus E^+) \leq 0 \quad \text{for all } E \in \mathcal{A}\,. \tag{1.13}$$

In particular $\mu(E^+) \geq 0$ and $\mu(M \setminus E^+) \leq 0$.

A fixed point - or eigenmeasure with eigenvalue 1 - is not the only object of interest when considering the operator P. Suppose we have a finite (potentially complex valued) measure with $P\nu = \lambda\nu$. Then we must have

$$\lambda\nu(M) = P\nu(M) = \int p(x, M)\; d\nu(x) = \int 1\; d\nu(x) = \nu(M). \tag{1.14}$$

Hence either $\lambda = 1$ or $\nu(M) = 0$. Now assume $\lambda \neq 0$, and that ν is scaled such that $|\nu|$ is a probability measure. Then by the Hahn decomposition we have two sets E^+ and $E^- = M \setminus E^+$ which partition M into positive and negative "regions" of M. From σ-additivity we have

$$0 = \nu(M) = \nu(E^+ \cup E^-) = \nu(E^+) + \nu(E^-). \tag{1.15}$$

Hence $\nu(E^+) = -\nu(E^-)$, which means $|\nu|(E^+) = |\nu|(E^-) = \frac{1}{2}$.

Theorem 1.16 (Almost Invariant Decomposition)**.** [16] Suppose ν is scaled such that $|\nu|$ is a probability measure, and $\nu(E^+) = \frac{1}{2}$. Then $\nu = |\nu|$ over E^+ and

$$\delta + \sigma = \lambda + 1 \tag{1.16}$$

if E^+ is δ-almost invariant and $E^- = M \setminus E^+$ is σ-almost invariant.

Example 1.17. Finally, we can consider the deterministic case with $\lambda \approx -1$. Then the sets E^+ and E^- satisfy

$$\delta = \nu(E^+ \cap f^{-1}(E^+)) \,/\, \nu(E^+) \;\geq 0, \tag{1.17}$$
$$\sigma = \nu(E^- \cap f^{-1}(E^-)) \,/\, \nu(E^-) \;\geq 0, \tag{1.18}$$
$$\delta + \sigma = \lambda + 1 \approx 0. \tag{1.19}$$

This implies that nearly all the ν-mass of E^+ gets transported to E^- by f, and vice versa. We call this an *almost invariant two-cycle*.

2 Algorithms

2.1 Relative Attractor

The construction of a fundamental neighborhood U for a global attractor A is relatively difficult, but the description Eq. 1.4 lends to a natural *ansatz* for its approximation using a compact subdomain $Q \subset M$.

Definition 2.1 (Relative Global Attractor)**.** Let Q be compact. Then we define the *attractor relative to* Q as

$$A_Q = \bigcap_{k \geq 0} f^k(Q) \tag{2.1}$$

Remark. It follows from the definition that the relative global attractor is a subset of the global attractor.

The idea to approximate the relative global attractor is in two steps:

1. First, *subdivide* the box set $\mathcal{B}$: construct a new set $\mathcal{B}'$ by splitting each box $B \in \mathcal{B}$ into two (or more) smaller boxes $B_1,\ B_2$ such that $B_1 \cup B_2 = B$.
2. Second, discard all those boxes whose image does not intersect with the previous box set.

Algorithm 1 Relative Attractor

$\mathcal{B}_0 \leftarrow \mathcal{B}$
for $i = \{1,\ \ldots,\ n\}$ **do** ▷ *n is a predefined number of iteration steps*
    $\mathcal{B}_i \leftarrow \text{SUBDIVIDE}(\mathcal{B}_{i-1})$
    $\mathcal{B}_i \leftarrow \mathcal{B}_i \cap f(\mathcal{B}_i)$
return $\mathcal{B}_n$

Remark.

- Optionally, the set $\{\mathcal{B}_0,\ \mathcal{B}_1,\ \ldots,\ \mathcal{B}_n\}$ can be returned instead. This will be true for all algorithms of this type.
- The precise technique for subdivision can be tuned depending on the situation. In the julia implementation presented in the current paper, boxes are bisected evenly along one dimension $k \in \{1,\ \ldots,\ d\}$. The dimension k along which to bisect is cycled through during the steps.

Proposition 2.2. [17, 13] Set $Q_n = \bigcup_{B\in\mathcal{B}_n} B, \quad Q_\infty = \bigcap_{n\geq 1} Q_n$. Assume the subdivision technique satisfies $d(\mathcal{B}_n) \to 0$ as $n \to \infty$. For all n we have

1. $Q_{n+1} \subset Q_n$,
2. $A_Q \subset Q_n$,
3. $A_Q = Q_\infty$.

In particular, this shows that Q_∞ is backward-invariant.

2.2 Unstable Manifold

From the definition of the local unstable manifold $W^u(\bar{x}, U)$ we see that the relative global attractor A_Q contains the local unstable manifold, and, provided the set Q is sufficiently small, $W^u(\bar{x}, U)$ coincides with A_Q. For further details, see [12, 41]. Using this knowledge, we can approximate the global unstable manifold $W^u(\bar{x})$:

1. First, we perform an *initialiation step*: replace the calculation of the local unstable manifold with the calculation of the relative attractor for a small set Q' surrounding the fixed point $\bar{x}$, using Algorithm 1.

2. Second, we repeat the following *continuation step*: map the current box set forward one iteration, and note any new boxes which are hit. These new boxes get added to the box set. Repeat until there are no new boxes added to the set.

Algorithm 2 Continuation Step

$\mathcal{B}_0 \leftarrow \mathcal{B}$
$\mathcal{B}_1 \leftarrow \mathcal{B}$
while $\mathcal{B}_1 \neq \emptyset$ **do**
    $\mathcal{B}_1 \leftarrow f(\mathcal{B}_1)$
    $\mathcal{B}_1 \leftarrow \mathcal{B}_1 \setminus \mathcal{B}_0$
    $\mathcal{B}_0 \leftarrow \mathcal{B}_1 \cup \mathcal{B}_0$
return $\mathcal{B}_0$

Proposition 2.3. [12] Let $\bar{x}$ by a hyperbolic fixed point of f. The algorithm in general cannot guarantee covering of the entire unstable manifold, nor can it guarantee covering of the entirety of $W^u(\bar{x}) \cap Q$. This is because $W^u(\bar{x})$ could in theory exit Q, but return at another point. The algorithm can however guarantee covering of the connected component of $W^u(\bar{x}) \cap Q$ which contains $\bar{x}$.

2.3 Chain Recurrent Set

The algorithm to compute the chain recurrent set is first due to [39]. The idea is to construct a directed graph whose vertices are the box set $\mathcal{B}$, and for which edges are drawn from B_1 to B_2 if f maps any part of B_1 into B_2. Call this graph $\textsc{graph}(f, \mathcal{B})$. We can now ask for a subset of the vertices, for which each vertex is part of a directed cycle. We can equivalently characterize this set as follows:

Definition 2.4 (Strongly Connected)**.** [40] For a directed graph $G = (V, E)$, a subset $H \subset V$ of vertices is called *strongly connected* if for all $u, v \in H$ there exist paths in both directions between v and u. Denote by $\textsc{scc}(G, v)$ the strongly connected subgraph which includes v, and by $\textsc{sccs}(G) = \bigcup_{v \in V} \textsc{scc}(G, v)$ the union of strongly connected components.

Proposition 2.5. [39, 40] Let $\mathcal{P}$ be a cover of Q into *closed* sets. Denote by $r(B)$ the minimum Hausdorff distance between the image $f(B)$ and a box B' that does not intersect the box covering $f(\mathcal{B})$, ie

$$r(B) = \min_{\substack{B' \in \mathcal{P} \\ B' \cap f(\mathcal{B}) = \emptyset}} d(B', f(B)) \quad \text{where} \quad \mathcal{B} = \{B\}\,. \tag{2.2}$$

Denote by r the minimum of $r(B)$ over $\mathcal{P}$, ie $r = \min_{B \in \mathcal{P}} r(B)$. Then we have

Algorithm 3 Chain Recurrrent Set

```
B_0 ← B
for i = {1, ..., n} do
    B_i ← SUBDIVIDE( B_{i-1} )
    G ← GRAPH( f, B_i )
    B_i ← SCCS( G )
return B_n
```

1. If a cycle $\{B_0,\ \ldots,\ B_{n-1}\}$ exists in G, then theres exists an ϵ-pseudoperiodic orbit $\{x_0,\ \ldots,\ x_{n-1}\}$ with $x_i \in B_i$ for any $\epsilon \geq d(\mathcal{P})$.

2. If an ϵ-pseudoperiodic orbit $\{x_0,\ \ldots,\ x_{n-1}\}$ exists with $\epsilon \leq r$, then the boxes $\{B_0,\ \ldots,\ B_{n-1}\}$ with $x_i \in B_i$ form a cycle in G.

Corollary 2.6. Proposition 2.2 holds for Algorithm 3. That is, replacing A_Q with R_Q in Proposition 2.2 remains true.

Proof. The inclusions 1. and 2. from Proposition 2.2 are clear (cf. [40]). For the equality 3. consider the following. The existence of a cycle in G is a *necessary* condition for an ϵ-pseudoperiodic orbit with $\epsilon < r$, and it is also a *sufficient* condition for an ϵ-pseudoperiodic orbit with $\epsilon \geq d(\mathcal{P})$. Letting $d(\mathcal{P}) \to 0$ by successive subdivision (which imples $r \to 0$) gives a condition which is both necessary and sufficient for the existence of ϵ-pseudoperiodic orbits with $\epsilon > 0$, and hence the box covering Q_∞ is precisely the chain recurrent set R_Q. For further details, see [39, 40]. □

2.4 Invariant Measure

We shift focus to approximating invariant measures for the Perron-Frobenius operator P. For simplicity we will work only with transition functions absolutely continuous with respect to the Lebesque measure on M, ie there exists a *kernel* k with

$$p(x, A) = \int_A k(x, y)\, dy. \tag{2.3}$$

In this case we can define $P : L^1 \to L^1$ as

$$(P\phi)(y) = \int \phi(x)\, k(x, y)\, dx. \tag{2.4}$$

Note that a fixed point ϕ of P is the density of an invariant measure $\mu_\phi(A) = \int_A \phi(x)\, dx$.

We will use a Galerkin approximation for P which maintains the eigenvalues and cyclic behavior of P. To do this, we project to a subspace $\chi_\mathcal{P}$ generated by the basis $\{\chi_B \mid B \in \mathcal{P}\}$ of indicator functions on the boxes of our partition. Further, enumerate the partition $\mathcal{P} = \{B_1,\ B_2,\ \ldots,\ B_n\}$ and define the matrix

$$(P^n)_{ij} = \frac{\mathcal{L}\left(B_j \cap f^{-1}(B_i)\right)}{\mathcal{L}(B_j)}, \quad i,\, j = 1, \ldots, n, \tag{2.5}$$

as well as a linear operator $Q_n P : \chi_{\mathcal{P}} \to \chi_{\mathcal{P}}$ as the linear extension of

$$(Q_n P)\, \chi_{B_i} = \sum_{j=1}^{n} P_{ij}^n\, \chi_{B_j}, \quad i = 1, \ldots, n. \tag{2.6}$$

To realize this approximation, we need to calculate P_{ij}. For this there are two techniques discussed in [17]. The simpler of the two techniques is a Monte-Carlo approach. Namely, we choose a fixed number r of test points in B_i, and set P_{ij} as the fraction of test points which land in B_j:

Algorithm 4 Invariant Measure

1: **for** $i, j \in \{1,\ \ldots,\ n\}$ **do**
2: $\quad p \ \leftarrow$ Choose $\{p_1,\ \ldots,\ p_r\}$ randomly from a uniform distribution on B_i
3: $\quad P_{ij}^n \leftarrow \mid B_j \cap f(p) \mid / \mid p \mid$
4: $v^n \leftarrow$ Find a fixed point of P^n
5: $\phi^n \leftarrow \sum_{i=1}^{n}\ v_i^n\, \chi_{B_i}$
6: **return** ϕ^n

Remark.

- In practice the Monte-Carlo test points are only generated once from the unit cube $[-1, 1]^d$, and then scaled to fit inside the box B_i.
- Line 4 is acheived using the Fortran library ARPACK [31], which has been wrapped in julia by the creators of Arpack.jl [36]. Recently, the underlying algorithm (an implicitly restarted Arnoldi method) has been implemented in pure julia under the package name ArnoldiMethod.jl [46].

Proposition 2.7. [30] Let $p(x, A)$ be a transition function with globally Lipschitz continuous kernel k. Then the operator $P : L^1 \to L^1$ is compact. Further, if k is strictly positive, P has a unique fixed point $\phi \in L^1$.

Proposition 2.8. [16, 32, 17, 28, 27] Suppose the transition function $p = p_\epsilon$ converges in the weak* sense to the dirac measure, ie

$$p_\epsilon(x, \cdot) \overset{*}{\rightharpoonup} \delta_{f(x)} \quad \text{as} \quad \epsilon \to 0. \tag{2.7}$$

(An example of this would be Ex. 1.10 for a sequence of partitions $\mathcal{P}_\epsilon$ with diameters $d(\mathcal{P}_\epsilon) \to 0$). Suppose further that the diffeomorphism f has a hyperbolic attracting set A with and an open set $U \supset A$ such that for the kernels we have

$$k_\epsilon(f(x), y) = 0 \quad \text{if} \quad x \in \overline{U},\ y \notin U. \tag{2.8}$$

Let P_ϵ be the Perron-Frobenius operator for the transition function p_ϵ. Then there exist unique fixed points ϕ_ϵ^n of $Q_n P_\epsilon$, and the sequence of fixed points converge to the fixed point ϕ of the true operator P, ie

$$\phi_\epsilon^n \to \phi \quad \text{as} \quad n \to \infty,\ \epsilon \to 0. \tag{2.9}$$

2.5 Almost Invariant Sets (and Cycles)

Our final algorithm is quite analogous to the algorithm for invariant measures of P. We begin with a theorem about the convergence our discretized operator.

Theorem 2.9. [16] Suppose our transition function p has a kernel k (see Eq. 2.3) which satisfies

$$\iint |k(x,y)|^2 \, dx\, dy < \infty. \tag{2.10}$$

Then the Frobenius Perron operator $P : L^2 \to L^2$ is compact. Further $Q_n P$ converges strongly to P in the operator norm on L^2, ie

$$\| Q_n P - P \| \to 0 \quad \text{as} \quad n \to \infty. \tag{2.11}$$

Assume now that there exists an $n \in \mathbb{N}$ such that P^n has an eigenvalue 1 with multiplicity 2. Then as described in [17]: by the above theorem, as $n \to \infty$ the eigenvalue will split into two simple eigenvalues, one of which will move away from 1 (the other will stay at unity since P^n is column stochastic). If we now find this eigenvector v^n of P^n, then (again by the above theorem and the continuity of P) the sets

$$E_n^+ = \bigcup_{i \mid v_i^n \geq 0} B_i \quad \text{and} \quad E_n^- = \bigcup_{i \mid v_i^n < 0} B_i \tag{2.12}$$

will converge to the sets E^+ and E^- from Theorem 1.16 as $n \to \infty$. That is, they are almost invariant.

Algorithm 5 Almost Invariant Sets

1: **for** $i, j \in \{1, \ldots, n\}$ **do**
2: $\quad p \leftarrow$ Choose $\{p_1, \ldots, p_r\}$ randomly from a uniform distribution on B_i
3: $\quad P_{ij}^n \leftarrow \mid B_j \cap f(p) \mid / \mid p \mid$
4: $v^n \leftarrow$ Find an eigenvector to an eigenvalue $\lambda \neq 1$ of P^n
5: $E_n^+ \leftarrow \cup \{B_i \mid v_i^n \geq 0\}$
6: $E_n^- \leftarrow \cup \{B_i \mid v_i^n < 0\}$
7: **return** $E_n^+,\ E_n^-$

Remark. The algorithm can also be used to find almost invariant cycles. In this case Line 4 simply needs to be changed to find an eigenvector near -1.

3 Julia

3.1 A (Very) Brief Introduction to the Julia Language

The julia language GitHub page decribes julia as "a high-level, high-performance dynamic language for technical computing" [7]. Its creators come from matlab, python, lisp among others, and they desired a langauge as syntactically simple as python, as powerful for linear algebra as matlab, and as fast as C [9]. To achieve this, julia is just-in-time compiled using LLVM to generate native machine code. Further, "julia uses multiple dispatch as a paradigm, which makes it easy to express many object-oriented and functional programming patterns" [8]. As a brief example of multiple dispatch, consider a simple user defined type `2dBox` (similar to the implementation in `GAIO.jl`):

```
1    struct 2dBox
2        center::Tuple{Float64,Float64}
3        radius::Tuple{Float64,Float64}
4    end
```

Then we can create a `2dBox` by calling `b = 2dBox((0.0, 0.0), (1.0, 1.0))`. A natural question to ask is whether a vector, say `x = [0.5, 0.6]`, lies within `b`. To do this, we can *overload* the function `in` from julia Base:

```
1    function Base.in(x::Vector, b::2dBox)
2        all( b.center .- b.radius .≤ x .≤ b.center .+ b.radius )
3    end
```

This function uses julia's *dot-syntax* to vectorize operations, eg. `+` by writing `.+`. We can now call the base function `in` or its unicode alias `∈`:

```
1    x = [0.5, 0.6]
2    b = 2dBox((0.0, 0.0), (1.0, 1.0))
3
4    x ∈ b           # returns true
5    x .- 2 ∈ b      # returns false
6    x .- 2 ∉ b      # returns true
```

The above example demonstrates the syntactical simplicity of julia. It should be noted that despite this simplicity, julia can generate highly performant machine code. For further illustration, consider the algorithm for the global unstable manifold (see Algorithm 2) implemented in matlab (Listing 1) and in julia (Listing 2). The julia code is one third

as long, mirrors the pseudocode much more closely, and still runs faster than the matlab code.

3.2 GAIO.jl

This section is meant as an introduction to the most important concepts in the implementation of `GAIO.jl`. For more information see the respective docstrings, eg. by typing `julia> ? Box` into the julia REPL.

The package is built off of three base structures `Box`, `BoxPartition`, `BoxMap`:

```
1    struct Box{N,T <: AbstractFloat}
2        center::SVector{N,T}
3        radius::SVector{N,T}
4    end
```

A Box is a half-open generalized rectangle, that is, a product of half-open intervals $[c_1 - r_1,\ c_1 + r_1) \times \ldots \times [c_d - r_d,\ c_d + r_d)$. We now need to represent a partition $\mathcal{P}$ of boxes. GAIO.jl offers two way to do this, we show only the simpler version:

```
1    struct BoxPartition{N,T,I<:Integer} <: AbstractBoxPartition{Box{N,T}}
2        domain::Box{N,T}
3        left::SVector{N,T}
4        scale::SVector{N,T}
5        dims::SVector{N,I}
6        dimsprod::SVector{N,I}
7    end
```

Instead of keeping track of positions for each individual box, we use integer indices for an equidistant *box grid* and only store the dimensions of the grid using the attribute `dims`. The attributes `left, scale, dimsprod` are used to quickly calculate box indices from a given point (see `point_to_key, key_to_box` in `GAIO.jl`). We further use the index structure to our advantage when storing box sets $\mathcal{B}$, only storing sets of indices corresponding to a partition:

```
1    struct BoxSet{B,P<:AbstractBoxPartition{B},S<:AbstractSet} <: AbstractSet{B}
2        partition::P
3        set::S
4    end
```

Finally, we need a way to convert a map f defined on Q to a map `F` defined on boxes of a `BoxPartition`:

```
function gum(t, f, X, depth)
dim = t.dim;
none = 0; ins = 2; expd = 4;              % defining flags
nb0 = 0;  nb1 = t.count(depth);           % bookkeeping the no. of boxes
tic; j = 1;
t.set_flags('all', ins, depth);
while nb1 > nb0                           % while new boxes nonempty
t.change_flags('all', ins, expd);         % mark inserted boxes
b = t.boxes(depth); M = size(b,2);        % get the geometry of the boxes
flags = b(2*dim+1, :);
I = find(bitand(flags,expd));             % find boxes to expand
b = b(:,I); N = size(b,2);
S = whos('X'); l = floor(5e7/S.bytes);
for k = 0:floor(N/l),                     % split in chunks of size l
    K = k*l+1:min((k+1)*l,N);
    c = b(1:dim,K);                       % center ...
    r = b(dim+1:2*dim,1);                 % ... and radii of the boxes
    n = size(c,2); E = ones(n,1);
    P = kron(E,X)*diag(r) + ...           % sample points in all boxes
        kron(c',ones(size(X,1),1));
    t.insert(f(P)', depth, ins, none);    % map sample points, insert boxes
end
t.unset_flags('all', expd);               % unflag recently expanded boxes
nb0 = nb1; nb1 = t.count(depth);
j = j+1;
end
```

Listing 1: Unstable manifold algorithm in matlab

```
function unstable_set(F::BoxMap, B::BoxSet)
    B₀ = B
    B₁ = B
    while B₁ ≠ ∅
        B₁ = F(B₁)
        B₁ = B₁ \ B₀
        B₀ = B₁ ∪ B₀
    end
    return B₀
end
```

Listing 2: Unstable manifold algorithm in julia

```
struct SampledBoxMap{A,N,T,F,D,I}
    map::F
    domain::Box{N,T}
    domain_points::D
    image_points::I
    acceleration::A
end
```

When a `SampledBoxMap` is initialized, we require a function to calculate test points that are mapped by the given function f. This function is `domain_points`. These test points are then mapped forward by f, and the boxes which are hit become the image set. More precisely, mapping a box set is done in two main steps within GAIO (see Listing 6):

1. Test points within the box are generated (or retrieved) using `domain_points(box_center, box_radius)`. These test points are mapped forward by the given function f.
2. For each mapped test point `fp`, an optional set of "perturbations" are generated using `image_points(fp, box_radius)`. For each of the perturbed points, the index of the box within the `BoxPartition` containing this point is calculated. This index gets added to the image set.

The `acceleration` attribute is what we will concern ourselves with for the remainder of the present paper. Naturally, if an increase in accuracy is required, a larger set of test points may be chosen. This leads to a dilemma: the more accurate we wish our approximation `F` to be, the more we need to map very similar test points forward, causing a considerable slow down for complicated dynamical systems. However, the process of mapping each test point forward is completely independent on other test points. This means we do not need to perform each calculation sequentially; we can *parallelize*.

4 Parallelization using the CPU

4.1 CPU Architechture

When referring to CPU architecture one typically either means *instruction set architecture (ISA)* referring to a range of CPUs, or the *microarchitechture* of a specific CPU model. The ISA is the blueprint for a set of abstract CPU characteristics such as supported instructions, data types, register count, etc. The microarchitechture is the *implementation* of this blueprint for a CPU. We restrict to the discussion of the microarchitecture for the Intel 3rd gen (Ivy Lake) line of processors, though the level of detail is low enough that the statements will apply to most modern CPUs built on the x86 64-bit ISA.

A processor microarchitecture can be further split to 3 components (see Fig. 4.1):

- The front end, which consists of instruction register (also called instruction cache) and instruction fetch / decode units. This is responsible for fetching batches of instructions from memory, storing the batches in the instruction cache, and decoding the instructions into a set of *micro-operations, or μOps*.
- The back end (also called execution engine), which includes the reorder buffer, unified scheduler (also called reservation station), and various execution ports. Since not all instructions are necessarily dependent on one-another, they can often be reordered, or multiple instructions can be executed simultaneously. This is known as *out-of-order execution*. The reorder buffer stores the the *order* of μOps until they are retired. The scheduler takes μOps and dispatches them to the various ports, each of which is specialized for a subset of instructions.
- The memory system. This includes (typically) three cache levels: L1, L2, and L3. The L1 cache is faster than L2, but has a lower storage capacity. Similarly L2 is faster than L3, but has lower storage capacity.

All three can be limiting factors in a computation, though we will mainly consider optimizations for the back end. Additionally, a modern processor will copy this basic structure multiple times, each copy is referred to as a *thread*.

The naivest but most practically difficult technique to increase performance is to utilize the whole instruction set available in the acrhitecture. Obviously it can't be expected that the programmer optimizes for every microarchitecture and every instruction; if that were true we would all be writing pure assembly language. But there are some easy changes one can make, primarily using *fused-multiply-add (FMA)* instructions. An example from [35]: consider the simple dynamical system over $\mathbb{R}$:

$$x_{k+1} = x_k^2 + p \tag{4.1}$$

for some p. To perform one iteration, the CPU needs to call `MUL` once and then `ADD` once, both of which have latency of roughly 5 clock cycles. Using `FMA`, this can be performed in one operation, nearly doubling performance. A consequence of this is that code which does not harness FMA instructions can harness at most half of the peak theoretical power of the CPU. In julia, this can be performed with the `@muladd` macro from `MuladdMacro.jl` [42], which automatically converts all combinations of multiply and add into calls to the inbuilt `muladd` function.

Conversely, a conceptually difficult but practically easy technique to implement is multithreading. Multithreading is more conceptually difficult because the process of spawning, scheduling and synchronizing processes over multiple cores is relatively complicated. However, due to the obvious benefit offered by multithreaded programming, mature libraries have developed to hide these complications under a layer of abstraction. In julia, examples include the `Threads` module from julia base, as well as the package `Transducers.jl` [5] which offers a backend for packages such as `FLoops.jl` [4]. Both options offer macros

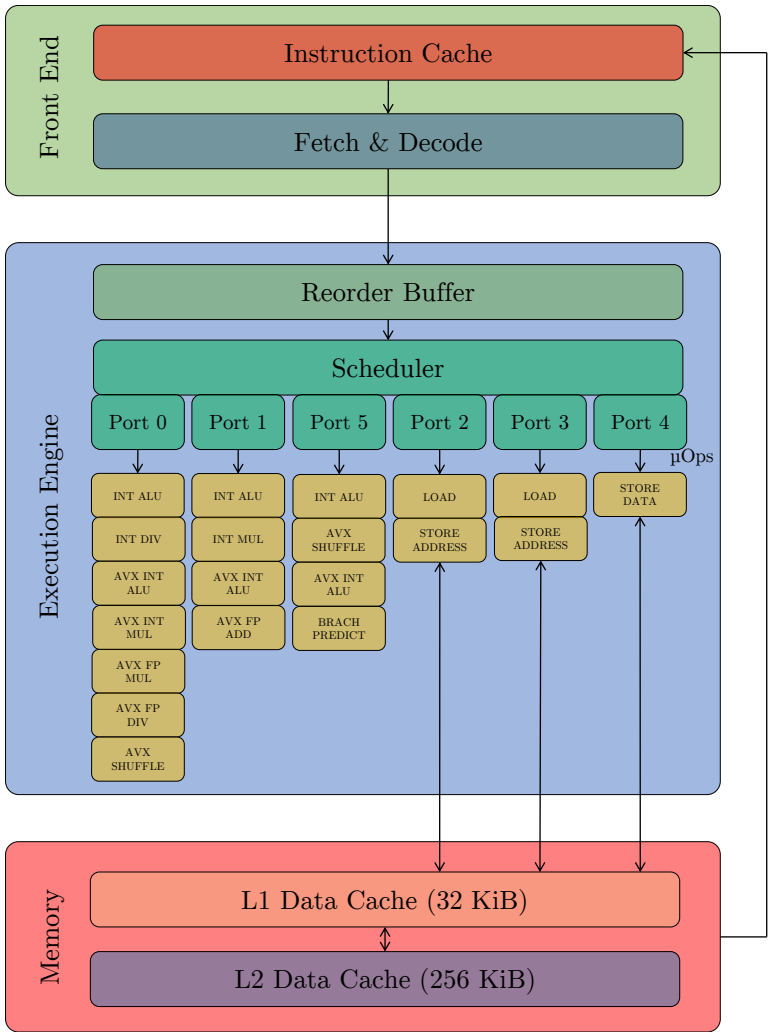


Figure 4.1: CPU microarchitecture (figure inspiration [35, 1])

- `@threads` and `@floop` respectively - which can simply be appended to **`for`** loops to automatically execute them over multiple threads.

Our main technique for parallelization on the CPU is the *Advanced Vector eXtension (AVX)*. The philosophy of AVX was initially characterized in [20] as *Single Instruction Multiple Data (SIMD)*. As the name would suggest: when a single instruction, eg. `ADD`, needs to be applied to multiple floating point numbers, then one can pack 4 double precision or 8 single precision floating point numbers into a single *vector* stored into the YMM (vector) register. This is called a *gather* operation. Two of these vectors can then be passed into port 1 to the AVX floating point arithmetic-logic-unit (`AVX FP ADD` in Fig. 4.1), and added all at once. This reduces the execution time from 20 clock cycles (4 double precision add operations at 5 clock cycles each) to just 8 clock cycles (1 packed add operation at 8 clock cycles).

The majority of modern compilers can automatically convert simple loops and vectorized functions into SIMD instructions. This however is not gauranteed, as the compiler neeeds to first prove that there are no data dependencies. Hence to reach peak CPU performance, it is often required to manually vectorize.

4.2 Implementation in GAIO.jl

We wish to map an array of test points $x = [x_1,\ x_2,\ \ldots,\ x_n],\ \ x_i \in \mathbb{R}^d$ forward, with as much parallelism as possible. For example, consider for $d = 3$ and a box $[0, 1]^3$ the test points:

```
x = [   # each tuple is seen as a point in 3d space
    (1., 0., 0.),
    (0., 1., 0.),
    (0., 0., 1.),
    (0., 0., 0.)
    # etc ...
]
```

We could equivalently characterize this array as an array of packed floats in the ”packed” space $(\mathbb{R}^4)^3$:

```
x = [   # three `4xFloat64<...>` are seen as a point in "packed" 3d
    (
        4xFloat64<1., 0., 0., 0.>,
        4xFloat64<0., 1., 0., 0.>,
        4xFloat64<0., 0., 1., 0.>
    )
    # etc ...
]
```

The packed SIMD vectors can be constructed using the `Vec` type from `SIMD.jl` [44], a convenience wrapper around julia’s base SIMD vector type (which itself is just a special tuple). So our goal becomes managing memory careully to convert from one vector of vectors to a vector of packed vectors, and vice versa.

A convenient fact about the `Tuple` type in julia is that if its elements are ”bits” types (that is, they can be represented by a string of bits that can be *stack-allocated*), then the tuple is stored *contiguously* in memory. For eg. numeric types like `Float32`, this means that the exact addresses in memory of each element can be deduced from the memory address of the tuple. Julia provides the convenience function `reinterpret` which changes the type interpretation of a block of memory.

Using `reinterpret`, we can change the problem of converting to (and from) packed vectors to the (simpler) problem of reordering the indices of x. Hence consider the vector i of

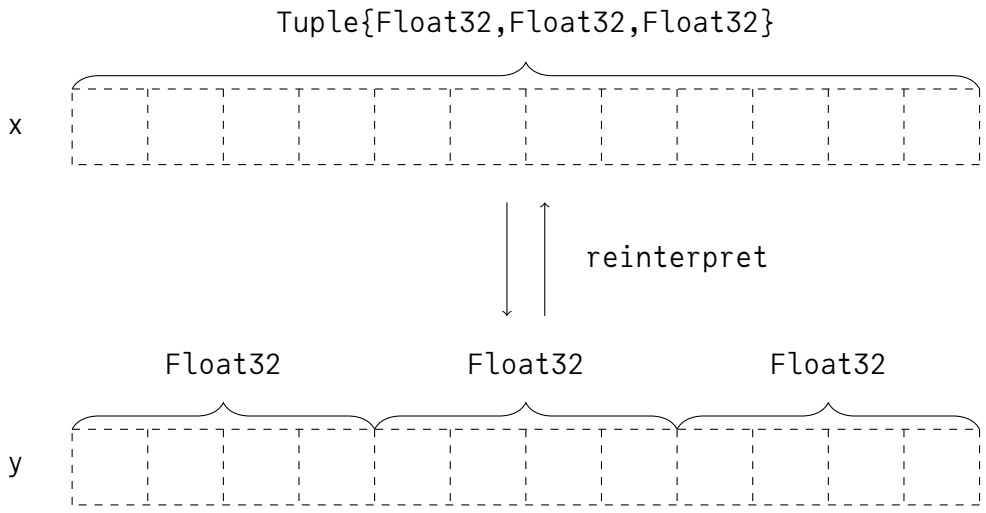


Figure 4.2: Illustration of julia's `reinterpret` function, where each square is 1 byte

indices of x:

$$i = (\underbrace{1,\ 2,\ 3,}_{\text{first point}}\quad \underbrace{4,\ 5,\ 6,}_{\text{second point}}\quad \ldots,\ \underbrace{3n-2,\ 3n-1,\ 3n}_{\text{nth point}}\ ). \tag{4.2}$$

We wish to permute i such that each group of four points has its respective elements stored contiguously. Once this is done, we can call `reinterpret` again to convert the type representation to a vector of packed tuples. In particular we therefore require that the number of points n is divisible by 4.

$$S(i) = (\underbrace{\underbrace{1,\ 4,\ 7,\ 10,}_{\text{first elements}}\quad \underbrace{2,\ 5,\ 8,\ 11,}_{\text{second elements}}\quad \underbrace{3,\ 6,\ 9,\ 12,}_{\text{third elements}}}_{\text{first packed point}}\ \ldots,\ 3n-6,\ 3n-3,\ 3n\ ) \tag{4.3}$$

We generalize this permutation S to arbitrary dimension d, SIMD vector length s and vector of points x with length n in Listing 3.

This function provides the fundamental technique for parallelization on the CPU. We first gather the points, then pass the gathered points to the function f. This way, f maps multiple points in parallel. However, we still need to convert the packed mapped points back to single points for use with other functions afterward. This is the inverse of a gather operation, and is called *scatter* (see Listing 4). We now have the full tool set to accelerate the function `map_boxes` using SIMD. For the implementation, see Listing 7.

It is important to note that not all elementary instructions have SIMD equivalents. Hence some more complicated functions cannot be parallelized in this way. Since GAIO.jl only controls test points, box partitions, etc., *the efficient implementation of the function f is solely up to the user.*

```
function tuple_vgather(x::Vector{NTuple{d,T}}, s) where {d,T}
    # `x` is a vector of tuples, each of length `d` and datatype `T`

    n = length(x)
    m = n ÷ s
    if n != m * s
        throw(DimensionMismatch("length of input not divisible by simd"))
    end

    # Change type interpretation of the memory of `x`
    vr = reinterpret(T, x)

    # Initialize the vector of packed tuples
    vo = Vector{NTuple{d,Vec{s,T}}}(undef, m)

    # The indices that form the first element of the first packed vector
    idx = Vec(ntuple( i -> (i-1) * d, s ))

    # Grab the indices of the first element of the i-th packed vector,
    # then jump by `d*s` to grab indices of the second element, etc.
    for i in 1:m
        vo[i] = ntuple( j -> vr[idx + (i-1) * d * s + j], d )
    end

    return vo
end
```

Listing 3: Conversion function to packed tuples

```
function tuple_vscatter(y::Vector{NTuple{d,Vec{s,T}}}) where {s,d,T}
    # `y` is a vector of packed tuples, tuples of SIMD Vecs

    # Initialize the unpacked vector
    vo = Vector{NTuple{d,T}}(undef, s * length(y))

    # Create a view of `vo` which is made of individual numbers of type `T`
    # as well as a view of `y` which is made up of packed `T`-vectors
    vr = reinterpret(T, vo)
    yy = reinterpret(Vec{s,T}, y)

    # The indices that form the first element of the first packed vector
    idx = Vec(ntuple( i -> d * (i-1), s ))

    # set the values of `vr` as the permuted values of `y`
    for i in 1:d, j in 1:length(vi)
        vr[idx + (j-1) * d * s + i] = yy[i + (j-1) * d]
    end

    return vo
end
```

Listing 4: Conversion back to single tuples

5 Parallelization using the GPU

5.1 GPU Architechture

While parallelizing code specifically for a CPU, one wishes to increase the amount of instruction-level-parallelism *per thread*. The CPU is built to perform tasks which are as large as possible, and with a little latency as possible, in a serial fashion. Latency is hidden on the CPU via small instruction registers but large low-latency on-chip caches and by out-of-order execution. However, one could alternatively approach the problem of increasing parallelism and hiding latency by increasing the number of threads entirely. This is the core core philosophy behind *massively parallel computing* - the philosophy adopted in the GPU. The GPU follows a throughput-oriented design in which as many tasks as possible are assigned to different threads concurrently. Less focus is given to low-latency memory access and operations, and more focus is given to launching many separate threads.

To manage this amount of parallel work, the components of a GPU are split into a hierarchy [18, 21, 35]:

- A single processing unit is known as a thread. A single consumer-level GPU can have on the order of tens of thousands of threads (Compare this with the thread count of a typical consumer-level CPU of 16 or 32 threads).

- Threads are grouped into warps. Each warp consists of 32 threads and a *warp scheduler*. This is the most granular level of scheduling that a GPU has. In particular this means that every thread in a warp performs a single instruction on different data. If a SIMD set (a set of data for which one operation should occur) is larger than the size of a warp, then the whole warp is launched multiple times.
- Threads are further grouped into (variably sized) blocks. Blocks are used by the next level in the hierarchy (SM) to manage execution over the GPU.
- Finally, blocks are grouped into streaming multiprocessors or SMs. Each SM contains a set of blocks as well as an instruction register and a scheduler. Instructions and data which get sent to the GPU are split and stored in the SMs, and each SM can then schedule tasks to the blocks that it controls.

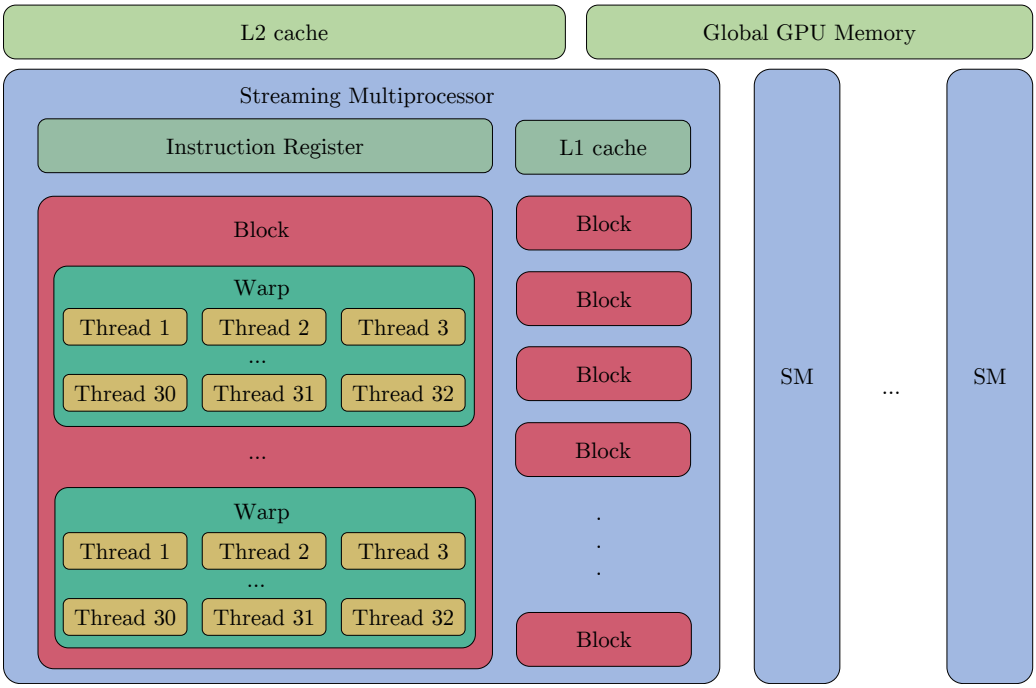


Figure 5.1: GPU microarchitecture (figure inspiration [35])

This hierarchical structure simplifies the task of handling such a massive number of threads. Since only one control unit is required per warp, this leaves more space for packing additional threads onto the chip. Further, the large instruction register means that each SM can hold many instructions and dispatch them to eligible warps before stalling due to a data request [35]. A warp may of course need to access the slower L1 or L2 caches for data, but while that is happening other warps are already performing different operations, so that - in effect - this memory access latency is hidden. A potential disadvantage of this approach is that the execution time of a warp is limited by the slowest thread in the warp forcing others to idle. If the execution time between threads in a warp

is large, this effect is called *thread divergence*.

An important consideration when launching computations on the GPU (known as *kernels*) is the block size. Since all threads in a SM must share the instruction register, increasing the block size (and therefore the number of threads controlled by the SM) means that there will necessarily be less register slots per thread. Conversely, decreasing block size will increase available register slots per thread, but will mean less threads total can be launched by the SM, since each SM has a maximal number of residing threads determined by the hardware.

Definition 5.1. [2, 35] We call the ratio of the number of residing threads n in a SM vs the theoretical maximum number of threads N per SM allowed by the hardware the *occupancy* O, that is, $O = n/N$.

Example 5.2. Nvidia's Ampere architecture supports a maximum of $N = 2048$ threads per SM, each with 65536 register slots and maximal block count of 32. Supported block sizes range from 1 to 1024, and a maximum of 255 register slots per thread [37].

- If we choose a block size of 32 to match the size of a warp, we get the maximum number of residing threads $n = 32 \cdot 32 = 1024$ and our occupancy is $O = n/N = 50\%$. Each thread has access to $65536/1024 = 64$ register slots.
- If we instead choose a block size of 64, then we reach full occupancy, but each thread now only has access to 32 register slots.

The precise "best" balance between occupancy and register size is highly problem dependent, and there are many other hardware-based factors which affect the true achievable occupancy. These factors iclude, but are not limited to, unbalanced workload within / across blocks, launching too few blocks per SM, and not launching enough calculations to saturate all threads in a warp [2]. This makes optimal GPU performance not always feasible when writing generic code. However, Nvidia provides an official occupancy calculator API [38] which suggests appropriate thread and block counts to approximately maximize occupancy while attempting to ensure that threads have large enough register availability. The julia package `CUDA.jl` [6] provides a wrapper for this API, accessible by calling `launch_configuration(cf)` for compiled cuda kernels `cf`.

5.2 Implementaion in GAIO.jl

The technique for parallelization via the GPU is quite natural: one thread is responsible for one test point. Initially, a more involved approach was tested, though this was replaced by a "naive" approach, as described in [45, 35]. The choice was made under the motto "keep it simple, stupid".

With the exception of Apple's "unified" desktop system-on-chip design, memory for the CPU and GPU are kept separate. Hence if a computation is to be launched on the GPU, the respective instructions and data must first be transferred from the system RAM via

a PCIe bus to the GPU's internal memory (VRAM). Transferring data via a PCIe bus is much slower than transferring data within a chip (eg. from the CPU's L1/L2 cache to instruction registers). Hence it is ideal to reduce necessary transfers and perform as much of a computation solely on the GPU or CPU.

In order to reduce transfers, a global set of test points is calculated when the `SampledBoxMap` is initialized. These points are normalized to the unit cirle w.r.t. $||\cdot||_\infty$, that is, the unit cube $[-1,1]^d$. The global test points are transferred only once to the GPU's VRAM at initialization, and at runtime each point p is appropriately rescaled using `@muladd p .* r .+ c` where r and c are the box's center and radius, respectively. This way, only the information about the map f, the partition $\mathcal{P}$, and the indices of the boxes to be mapped must be transferred to the GPU at runtime. Each thread calculates one index for the box containing the thread's mapped point, and writes this index to an array. Across threads we loop over test points then over box indices. This way consecutive threads will be assigned to test points in the same box, which (using continuity of f) reduces the chance of thread divergence. Finally, the array is transferred to the CPU and reduced to become the image set. The implementation of the kernel is shown in Listing 5, and the GPU accelerated `map_boxes` function in Listing 8.

As with the CPU kernel, there are some factors only in control of the user. Modern GPUs are primarily designed for *single-precision* operations, and are correspondingly much less efficient using double precision arithmetic. GAIO.jl does its best to convert all objects such as partitions and box sets to single precision, but again *the efficient implementation of the function f is solely up to the user.*

6 Results

The aforementioned parallelization techinques were tested using the Lorenz system [33]

$$\dot{x} = \sigma(y - x), \tag{6.1}$$
$$\dot{y} = x(\rho - z) - y, \tag{6.2}$$
$$\dot{z} = xy - \beta z, \tag{6.3}$$

with the parameters $\sigma = 10,\ \rho = 28,\ \beta = 2/5$. The function f_k was defined as performing k steps of the classic fourth order Runge Kutta scheme [43, 29] with step size $1/10k$. Further, the `SampledBoxMap` $\texttt{F}_\texttt{k}^\texttt{N}$ was generated to use N Monte-Carlo test points to map boxes in a $128 \times 128 \times 128$-element equidisdant `BoxPartition` over the compact domain $Q = [-30, 30] \times [-30, 30] \times [-5, 55]$.

One trial of the test was calculating the unstable set for the equilibrium point

$$(x, y, z) = \left(\sqrt{\beta(\rho - 1)},\ \sqrt{\beta(\rho - 1)},\ \rho - 1 \right) \tag{6.4}$$

```
function map_boxes_kernel(
        f,          # Function
        keys,       # Keys to be mapped
        points,     # "Global" test points
        out_keys,   # Array to hold mapped box indices ...
        P           # ... wrt the partition P
    )

    # Calculate the linear index of the GPU thread
    ind = (blockIdx().x - 1) * blockDim().x + threadIdx().x - 1
    stride = gridDim().x * blockDim().x
    nk, np = length(keys), length(points)
    len = nk * np - 1

    # In case `len` is too large, launch the warp repeatedly
    for i in ind : stride : len

        # Loop over the test points, then over the box index
        m, n = divrem(i, np) .+ 1
        p    = points[m]
        key  = keys[n]
        box  = key_to_box(P, key)
        c, r = box.center, box.radius

        # Map the test point forward
        fp   = f(@muladd p .* r .+ c)
        hit  = point_to_key(P, fp)

        # If `fp` lands outside the partition, assign it the
        # out-of-bounds index 0. This will later be discarded
        out_keys[i+1] = isnothing(hit) ? 0 : hit

    end
end
```

Listing 5: GPU kernel to calculate $f(\mathcal{B})$

using Algorithm 2 (see also Listing 2). Algorithm 2 was chosen for its simplicity: nearly all of the effective run time is due to Line 4 calculating $f(\mathcal{B})$. This makes it ideal for demonstration. The test was performed on a Dell G15 5510 laptop with Intel i7-10870H CPU and Nividia GeForce RTX 3060 laptop GPU, running Ubuntu 22.04 LTS. A depiction of the unstable set can be found in Fig. 6.1, result graphs can be found in Fig. 6.2, and code for the tests can be found in [22].

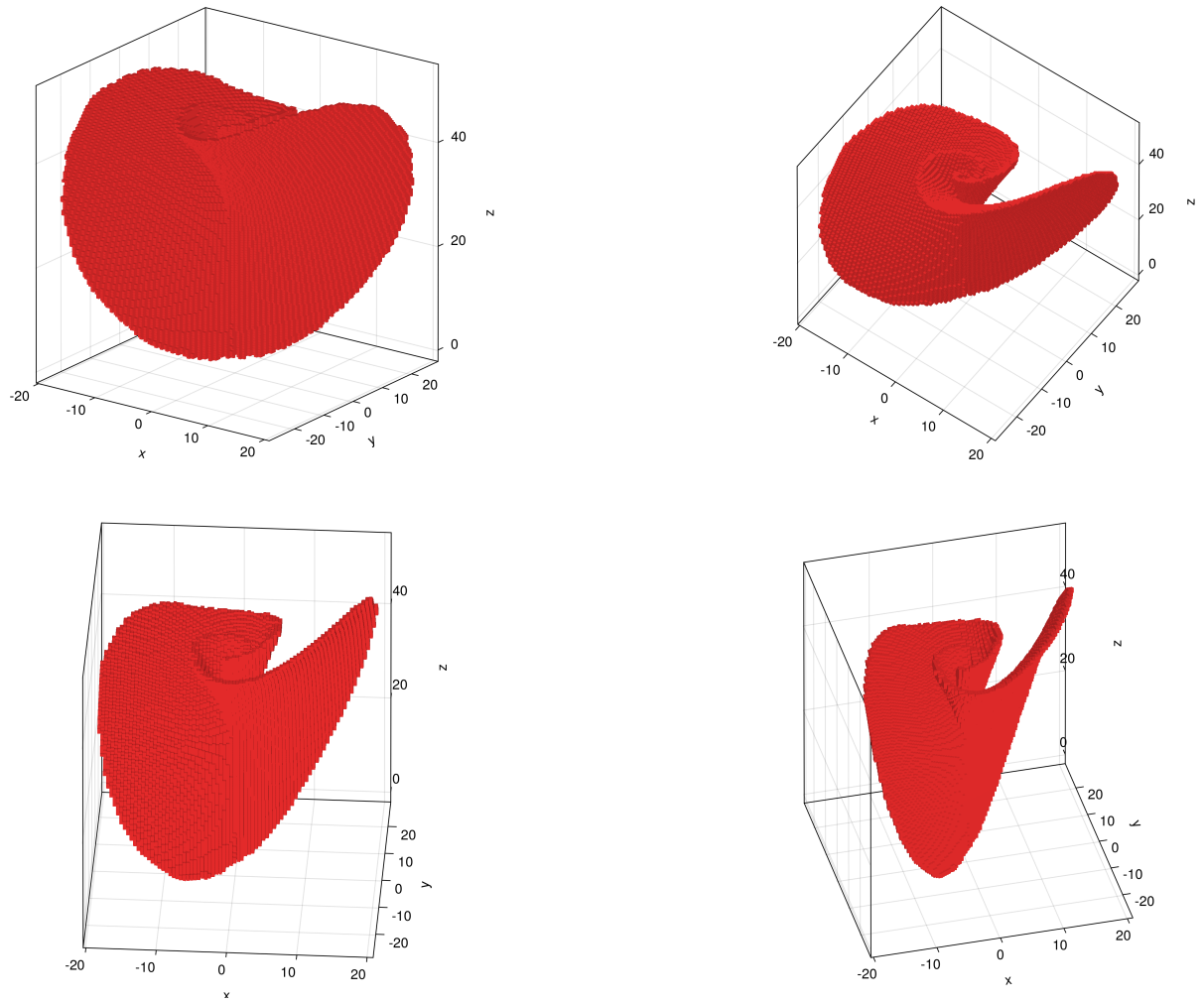
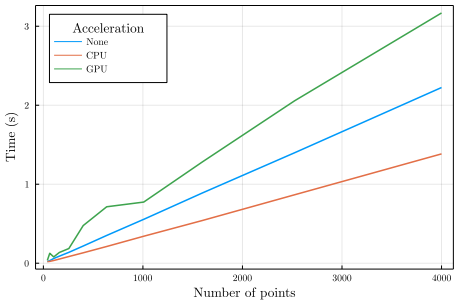
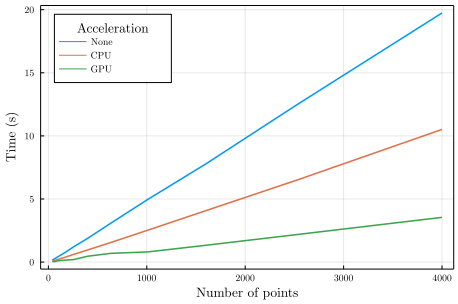


Figure 6.1: Box Covering of the unstable set for the Lorenz system around the equilibrium point described in Eq. 6.4

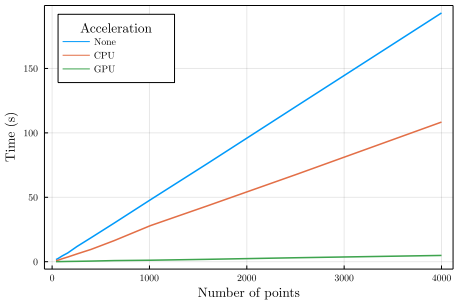
Execution time without acceleration and with CPU acceleration both show linear trends w.r.t. both k and N. CPU acceleration consistently performs faster by a factor of 2 across all trials. One might expect that the 256 bit vector registers, which can fit 4 double-precision floating point numbers each, should provide a speedup factor of 4. A set of 4 double precision floating point multiplcations can be sped up by slightly less than a factor of 4 by replacing it with one packed SIMD multiplication. However, this does not



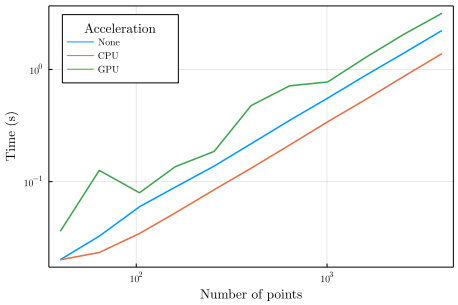
(a) Times for a "simple" map, $k = 10$.



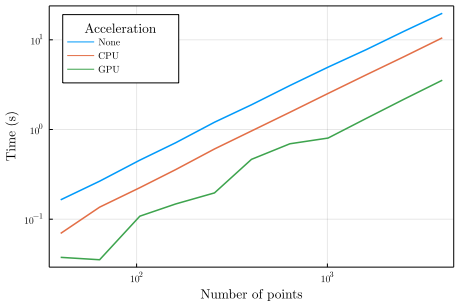
(b) Times for an "average" map, $k = 100$.



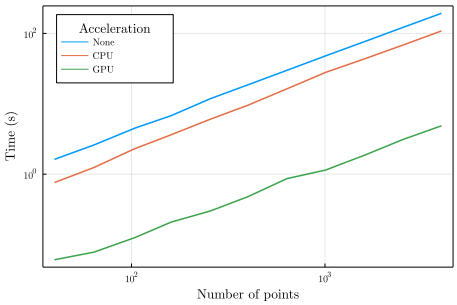
(c) Times for a "complex" map, $k = 1000$.



(d) Log-Log plot of 6.2a



(e) Log-Log plot of 6.2b



(f) Log-Log plot of 6.2c

Figure 6.2: Execution time for Algorithm 2 using various steps k and test points N.

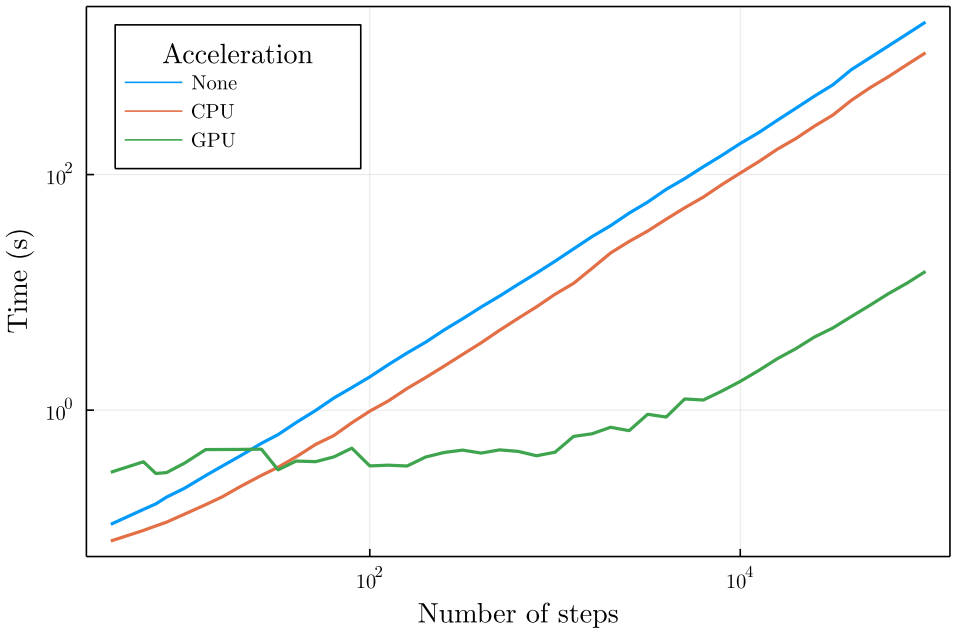


Figure 6.3: Log-Log plot of number of steps k vs time (s), where $N = 400$ is constant.

factor in the time taken to convert the points into a vector stored in the YMM register (the gather and scatter operations). Furthermore, the bookkeeping required to manage a

set of d-dimensional test vectors adds further complexity. Consider inlining the function `tuple_vscatter!` (Listing 4) into the CPU-accelerated method for `map_boxes` (Listing 6). The code length grows by nearly one third and includes an extra `for` loop.

More interesting are the performance results of the GPU-accelerated code. Glancing at Fig. 6.2a it would seem that the GPU has actually hindered performance, though this is no longer the case in Fig. 6.2b. This is because the calculation for the "simple" map can be performed so quickly that the execution time is dominated by the time it takes to transfer the box set to the GPU memory. A more detailed examination of this phenomenon can be seen in Fig. 6.3. In this case $k = 26$ steps was required before the GPU-accelerated version beat out the version without acceleration, and $k = 32$ steps were required to beat the CPU-accelerated version. However, once this threshold has been passed the GPU performs significantly better. Moreover, the speedup factor does not remain bounded, as it did with the CPU version. By $k = 7944$ steps, the GPU version has surpassed a 100-fold speedup, and continues to climb. Due to the limits of the testing hardware and time, steps were only tested up to $k = 10000$. At this point the rate of increase of the speedup factor had slowed, though notably had not stopped.

The results of the present paper solidify that GAIO.jl is an ideal candidate for parallel programming due to its high proportion of data-independent computations. As of the writing, there are still regions of active development. These include endowing the matrix transfer operator P^n from Eq. 2.5 with a structure that allows it to act as the operator $Q_n P$ from Eq. 2.6, as well as efficiently porting adaptive test point sampling techniques to the GPU.

References

[1] Andreas Abel and Jan Reineke. "uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Apr. 2019. DOI: 10.1145/3297858.3304062. URL: https://doi.org/10.1145%2F3297858.3304062.

[2] *Achieved Occupancy*. https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm. 2021.

[3] Hans Wilhelm Alt. *Linear Functional Analysis*. Springer Berlin, 2016. ISBN: 978-1-4471-7279-6. DOI: 10.1007/978-1-4471-7280-2.

[4] Takafumi Arakaki et al. *FLoops.jl*. https://github.com/JuliaFolds/FLoops.jl.git. 2022.

[5] Takafumi Arakaki et al. *Transducers.jl*. https://github.com/JuliaFolds/Transducers.jl.git. 2022.

[6] Tim Besard, Christophe Foket, and Bjorn De Sutter. "Effective Extensible Programming: Unleashing Julia on GPUs". In: *IEEE Transactions on Parallel and Distributed Systems* (2018). ISSN: 1045-9219. DOI: 10.1109/TPDS.2018.2872064. arXiv: 1712.03112 [cs.PL].

[7] Jeff Bezanson et al. *julialang*. https://github.com/JuliaLang/julia.git. 2022.

[8] Jeff Bezanson et al. *julialang.org*. https://julialang.org/. 2022.

[9] Jeff Bezanson et al. "Julia: A fast dynamic language for technical computing." In: *arXiv* (2012). DOI: https://doi.org/10.48550/arXiv.1209.5145.

[10] Haim Brezis, ed. *Nonlinear Systems and Chaos*. Vol. 19. Progress in Nonlinear Differential Equations and their Applications. 1996. DOI: https://doi.org/10.1007/978-3-0348-7518-9.

[11] Sarah Day, Oliver Junge, and K. Mischaikow. "A Rigorous Numerical Method for the Global Analysis of Infinite-Dimensional Discrete Dynamical Systems". In: *SIAM Journal on Applied Dynamical Systems* 3.2 (2004). DOI: https://doi.org/10.1137/030600210.

[12] Michael Dellnitz and Adreas Hohmann. "The Computation of Unstable Manifolds Using Subdivision". In: *Nonlinear Systems and Chaos*. Ed. by Haim Brezis. Vol. 19. Progress in Nonlinear Differential Equations and their Applications. 1996, pp. 449–459. DOI: https://doi.org/10.1007/978-3-0348-7518-9.

[13] Michael Dellnitz and Andreas Hohmann. "A Subdivision Algorithm for the Computation of Unstable Manifolds and Global Attractors". In: *Numerische Mathematik* 75 (1997). DOI: https://doi.org/10.1007/s002110050240.

[14] Michael Dellnitz and Oliver Junge. "An Adaptive Subdivision Technique for the Approximation of Attractors and Invariant Measures". In: *Computing and Visualization in Science* 1.1 (2010). DOI: https://doi.org/10.1007/s007910050006.

[15] Michael Dellnitz and Oliver Junge. "An adaptive subdivision technique for the approximation of attractors and invariant measures. Part II: Proof of convergence". In: *Computing and Visualization in Science* 1.1 (2010). DOI: https://doi.org/10.1080/14689360109696233.

[16] Michael Dellnitz and Oliver Junge. "On the Approximation of Complicated Dynamical Behavior". In: *SIAM Journal on Numerical Analysis* 2.36 (1999).

[17] Michael Dellnitz, Oliver Junge, and Gary Froyland. "The Algorithms Behind GAIO - Set Oriented Numerical Methods for Dynamical Systems". In: *Ergodic Theory, Analysis, and Efficient Simulations of Dynamical Systems*. Ed. by Bernold Fiedler. Springer Berlin, 2001, pp. 145–174. DOI: https://doi.org/10.1007/3-540-35593-6.

[18] *EXPLORING THE GPU ARCHITECTURE*. Tech. rep. vmware, 2020. URL: https://images.core.vmware.com/sites/default/files/resource/exploring_the_gpu_architecture_noindex.pdf.

[19] Bernold Fiedler, ed. *Ergodic Theory, Analysis, and Efficient Simulations of Dynamical Systems*. Vol. 1. 2001. DOI: https://doi.org/10.1007/978-3-642-56589-2.

[20] Michael J. Flynn. "Very high-speed computing systems". In: *Proceedings of the IEEE*. Vol. 54. 1966, pp. 1901–1909. DOI: https://doi.org/10.1109/PROC.1966.5273.

[21] Alan Gray. "GPU Architecture". lecture notes. 2017. URL: https://www.archer.ac.uk/training/course-material/2017/11/gpu-daresbury/slides/GPU_Architecture.pdf.

[22] April Herwig. *Benchmarks*. https://github.com/April-Hannah-Lena/schoolwork/tree/main/Thesis/benchmarks. 2022.

[23] April Herwig. *UnicodeSetOperations.jl*. https://github.com/April-Hannah-Lena/UnicodeSetOperations.jl.git. 2022.

[24] Andreas Johann. "Nichtlineare Dinamik". lecture notes. 2021.

[25] Oliver Junge. *GAIO*. https://github.com/gaioguy/GAIO. 2020.

[26] Oliver Junge, April Herwig, Lukas Mayrhofer, et al. *GAIO.jl*. https://github.com/gaioguys/GAIO.jl.git. 2022.

[27] R. Z. Khas'minskii. "Principle of Averaging for Parabolic and Elliptic Differential Equations and for Markov Processes with Small Diffusion". In: *Theory of Probability & Its Applications* 8.1 (1963), pp. 1–21. DOI: 10.1137/1108001. eprint: https://doi.org/10.1137/1108001. URL: https://doi.org/10.1137/1108001.

[28] Yuri Kifer. *Random Perturbations of Dynamical Systems*. Progress in Probability. Birkhäuser Boston, MA, 1988. DOI: https://doi.org/10.1007/978-1-4615-8181-9.

[29] W. Kutta. "Beitrag zur näherungsweisen Integrationtotaler Differentialgleichungen". In: *Zeit. Math. Phys.* 46 (1901), pp. 435–53.

[30] Andrzej Lasota and Michael C. Mackey. *Chaos, Fractals, and Noise. Stochastic Aspects of Dynamics*. Springer New York, NY, 1994. DOI: https://doi.org/10.1007/978-1-4612-4286-4.

[31] Richard B. Lehoucq, Danny C. Sorensen, and Chao Yang. *ARPACK Users' Guide: Solution of Large-scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*. SIAM, 1998.

[32] Tien-Yien Li. "Finite approximation for the Frobenius-Perron operator. A solution to Ulam's conjecture". In: *Journal of Approximation Theroy* 17.2 (1976), pp. 177–186. DOI: https://doi.org/10.1016/0021-9045(76)90037-X.

[33] Edward N. Lorenz. "Deterministic Nonperiodic Flow". In: *Journal of the Atmospheric Sciences* 20 (2 1963), pp. 130–141. DOI: https://doi.org/10.1175/1520-0469(1963)020%3C0130:DNF%3E2.0.CO;2.

[34] *MATLAB version 9.3.0.713579 (R2017b)*. The Mathworks, Inc. Natick, Massachusetts, 2017.

[35] Dániel Nagy, Lambert Plavecz, and Ferenc Hegedűs. *Solving large number of non-stiff, low-dimensional ordinary differential equation systems on GPUs and CPUs: performance comparisons of MPGOS, ODEINT and DifferentialEquations.jl*. 2020. DOI: 10.48550/ARXIV.2011.01740. URL: https://arxiv.org/abs/2011.01740.

[36] Andreas Noack et al. *Arpack.jl*. https://github.com/JuliaLinearAlgebra/Arpack.jl.git. 2021.

[37] *NVIDIA A100 Tensor Core GPU Architecture*. Tech. rep. nvidia, 2020. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf.

[38] *NVidia Occupancy Calculator*. https://docs.nvidia.com/cuda/cuda-occupancy-calculator/CUDA_Occupancy_Calculator.xls. 2021.

[39] George Osipenko. "Construction of Attractors and Filtrations". In: *Banach Center Publications* 47 (1999).

[40] George Osipenko. *Dynamical Systems, Graphs, and Algorithms*. Springer Berlin, 2007. DOI: https://doi.org/10.1007/3-540-35593-6.

[41] Jacob Palis and Wellington Melo. *Geometric Theory of Dynamical Systems*. Springer New York, 1982. DOI: https://doi.org/10.1007/978-1-4612-5703-5.

[42] Christopher Rackauckas, David Widmann, et al. *MuladdMacro.jl*. https://github.com/SciML/MuladdMacro.jl.git. 2022.

[43] C. Runge. *Ueber die numerische Auflösung von Differentialgleichungen*. 1895. DOI: https://doi.org/10.1007/bf01446807.

[44] Erik Schnetter, Kristoffer Carlsson, et al. *SIMD.jl*. https://github.com/eschnett/SIMD.jl.git. 2022.

[45] Christopher P. Stone, Andrew T. Alferman, and Kyle E. Niemeyer. "Accelerating finite-rate chemical kinetics with coprocessors: Comparing vectorization methods on GPUs, MICs, and CPUs". In: *Computer Physics Communications* 226 (2018), pp. 18–29. ISSN: 0010-4655. DOI: https://doi.org/10.1016/j.cpc.2018.01.015.

[46] Harmen Stoppels et al. *ArnoldiMethod.jl*. https://github.com/JuliaLinearAlgebra/ArnoldiMethod.jl.git. 2021.

7 Appendix

Number of Points	Time, $k = 10$	Time, $k = 100$	Time, $k = 1000$
40	0.020134	0.16445	1.62278
64	0.0326548	0.2653	2.60109
104	0.059709	0.455421	4.50348
160	0.0892529	0.71049	6.76705
256	0.137585	1.20524	11.8306
400	0.217655	1.88588	18.6102
636	0.351438	3.09114	29.8942
1008	0.55545	4.9555	47.9885
1596	0.888532	7.75978	76.2201
2524	1.39837	12.4615	121.395
4000	2.22419	19.7415	193.073

Table 1: Results using no acceleration

Number of Points	Time, $k = 10$	Time, $k = 100$	Time, $k = 1000$
40	0.0201105	0.0694606	0.761066
64	0.0233355	0.136251	1.24381
104	0.034471	0.224639	2.3031
160	0.0525481	0.356561	3.62152
256	0.0846066	0.606809	6.04962
400	0.131712	0.962795	9.5455
636	0.211706	1.55601	16.3179
1008	0.341492	2.52285	27.9455
1596	0.540294	4.06358	43.3757
2524	0.866405	6.49313	68.151
4000	1.38411	10.5114	108.385

Table 2: Results using CPU acceleration

Number of Points	Time, $k = 10$	Time, $k = 100$	Time, $k = 1000$
40	0.0360844	0.0375707	0.0607846
64	0.125934	0.0353227	0.0780206
104	0.0795413	0.108104	0.125333
160	0.135748	0.147764	0.207934
256	0.185914	0.196439	0.300255
400	0.476413	0.464716	0.47896
636	0.714487	0.69321	0.863987
1008	0.774053	0.80273	1.14469
1596	1.2813	1.32692	1.8528
2524	2.05627	2.17883	3.10034
4000	3.16808	3.54294	4.86421

Table 3: Results using GPU acceleration

```
function map_boxes(g::SampledBoxMap, source::BoxSet)
    P = source.partition

    # Use multithreaded iteration over the box set
    @floop for box in source
        c, r = box.center, box.radius
        domain_points = g.domain_points(c, r)

        # Map each test point
        for p in domain_points
            fp = g.map(p)
            hitbox = point_to_box(P, fp)

            # Skip to next iteration if `fp` lands outside `P`
            isnothing(hitbox) && continue

            # Add perturbations to `fp`
            r = hitbox.radius
            image_points = g.image_points(c, r)
            for ip in image_points
                hit = point_to_key(P, ip)
                isnothing(hit) && continue

                # Use @reduce syntax to initialize an empty
                # key set and add hits during iteration
                @reduce(image = union!(Set{keytype(P)}(), hit))
            end
        end
    end
    return BoxSet(P, image)
end
```

Listing 6: Function to calculate $f(\mathcal{B})$

```
function map_boxes(
        g::SampledBoxMap{C,N}, source::BoxSet
    ) where {simd,C<:BoxMapCPUCache{simd},N}

    # Preallocated temporary storage objects.
    # `temp_vec` and `temp_points` are reinterprets of
    # the same array, `temp_points` is an array of tuples.
    # `idx_base` is a SIMD Vec of indices used for `tuple_vscatter!`
    idx_base, temp_vec, temp_points = g.acceleration

    P = source.partition
    @floop for box in source

        # Grab a separate section of the
        # temporary storage for each thread
        tid  = (threadid() - 1) * simd
        idx  = idx_base + tid * N
        mapped_points = @view temp_points[tid+1:tid+simd]

        domain_points = g.domain_points(box.center, box.radius)
        for p in domain_points
            fp = g.map(p)

            # Scatter packed `fp` into temporary storage
            tuple_vscatter!(temp_vec, fp, idx)

            # continue as with normal `map_boxes`
            for q in mapped_points
                hitbox = point_to_box(P, q)
                isnothing(hitbox) && continue
                image_points = g.image_points(q,  hitbox.radius)
                for ip in image_points
                    hit = point_to_key(P, ip)
                    isnothing(hit) && continue
                    @reduce(image = union!(Set{keytype(P)}(), hit))
                end
            end
        end
    end
    return BoxSet(P, image)
end
```

Listing 7: Function to calculate $f(\mathcal{B})$ with CPU acceleration

```
function map_boxes(
        g::SampledBoxMap{C}, source::BoxSet
    ) where {SZ,C<:BoxMapGPUCache{SZ}}
    # BoxMapGPUCache holds a max size `SZ` of box indices
    # which can fit in GPU memory

    # We use `Stateful` to iterate through
    # test points `SZ` points at a time
    P, keys = source.partition, Stateful(source.set)

    # Normalized test points
    points = g.domain_points(P.domain.center, P.domain.radius)
    np = length(points)

    image = Set{keytype(P)}()
    while !isnothing(keys.nextvalstate)
        stride = min(SZ, length(keys))

        # Allocate input/output indices on GPU
        in_keys = CuArray{Int32,1}(collect(Int32, take(keys, stride)))
        nk = length(in_keys)
        out_keys = CuArray{Int32,1}(undef, np * nk)

        # Compile the GPU kernel and calculate occupancy
        args = (g.map, in_keys, points, out_keys, P)
        compiled_kernel! = @cuda launch=false map_boxes_kernel!(args...)
        config  = launch_configuration(compiled_kernel!.fun)
        threads = min(np * nk, config.threads)
        blocks  = cld(np * nk, threads)

        # Launch the compiled kernel then wait until the task to complete
        compiled_kernel!(args...; threads, blocks)
        CUDA.synchronize()

        # Transfer output indices to CPU memory
        out_cpu = Array{Int32,1}(out_keys)
        union!(image, out_cpu)

        # Manually deallocate GPU arrays since julia has
        # more difficulty garbage collecting GPU memory
        CUDA.unsafe_free!(in_keys); CUDA.unsafe_free!(out_keys)
    end

    # remove out-of-bounds index
    delete!(image, 0i32)

    return BoxSet(P, image)
end
```

Listing 8: Function to calculate $f(\mathcal{B})$ with GPU acceleration

Number of Steps	Time, no accel.	Time, CPU accel.	Time, GPU accel.
4	0.10733	0.0774621	0.296883
6	0.143625	0.0952406	0.364458
7	0.159921	0.103962	0.289842
8	0.182721	0.111939	0.294995
10	0.217213	0.130934	0.35477
13	0.278128	0.157542	0.462502
16	0.335391	0.183797	0.463087
20	0.40894	0.224452	0.463877
26	0.522386	0.279074	0.466465
32	0.619818	0.32773	0.311194
40	0.783703	0.400696	0.369541
51	0.993318	0.511517	0.365205
64	1.26785	0.608839	0.400862
80	1.554	0.780857	0.475648
100	1.91483	0.979949	0.336051
126	2.4347	1.19774	0.340938
159	3.05702	1.52709	0.335395
200	3.77057	1.88865	0.39955
252	4.79369	2.36452	0.436983
317	5.9642	2.97117	0.457834
399	7.50092	3.72863	0.432015
502	9.30319	4.76645	0.459764
631	11.6748	6.0041	0.446623
795	14.5952	7.5396	0.409987
1000	18.3667	9.63419	0.439264
1259	23.3917	11.9185	0.598132
1585	29.741	16.0015	0.62983
1996	36.8385	21.5996	0.715319
2512	47.0677	27.0529	0.669515
3163	58.3913	33.1406	0.922343
3982	74.9216	41.7995	0.873487
5012	92.3616	52.2589	1.24426
6310	116.477	64.3065	1.218
7944	145.294	82.0594	1.44954
10000	183.667	103.063	1.75376
12590	227.309	128.813	2.17254
15849	289.176	164.242	2.73257
19953	365.261	202.171	3.32561
25119	463.006	257.573	4.1719
31623	578.231	320.201	4.98578
39811	778.834	427.457	6.23462
50119	980.618	545.352	7.76651
63096	1237.09	678.236	9.75043
79433	1560.56	857.923	11.9667
100000	1970.96	1080.9	15.0591

Table 4: Detailed results comparing map complexity and execution time