University of California Santa Cruz

Data-Driven Computational Optimal Control for Uncertain Nonlinear Systems

Prof. Daniele Venturi and Prof. Qi Gong

PhD Students: Tenavi Nakamura-Zimmerer, Abram Rogers, Catherine Brennan,
and Panos Lambrianides.

DARPA Lagrange Program

Quarterly Progress Report (Q2)
July-September 2018

Contents

1 Executive summary

In this proposal, we address a very important research area in computational mathematics, namely the design and synthesis of optimal control strategies for high-dimensional stochastic dynamical systems. Such systems may be classical nonlinear systems evolving from random initial states, or systems driven by random parameters or processes. The first objective is to provide a validated new computational capability for optimal control of stochastic systems which will be achieved at orders of magnitude more efficiently than current methods based on spectral collocation or random sampling. To accomplish this goal, we will develop a new *data-driven* optimal control framework based on probability density function (PDF) equations. The new framework is built upon high-order numerical tensor methods, with no specific requirements on the structure of the continuous dynamics, cost function, or the type of uncertainties. The 18 months research plan is multidisciplinary and it involves multiple fields such as optimal control, large-scale optimization, and uncertainty quantification. It consists of theoretical and numerical developments, as well as a general software framework that will implement the proposed algorithms. The proposed research work will have a significant and broad impact in a wide range of engineering applications such as autonomous systems, environmental defense, and control of random networks.

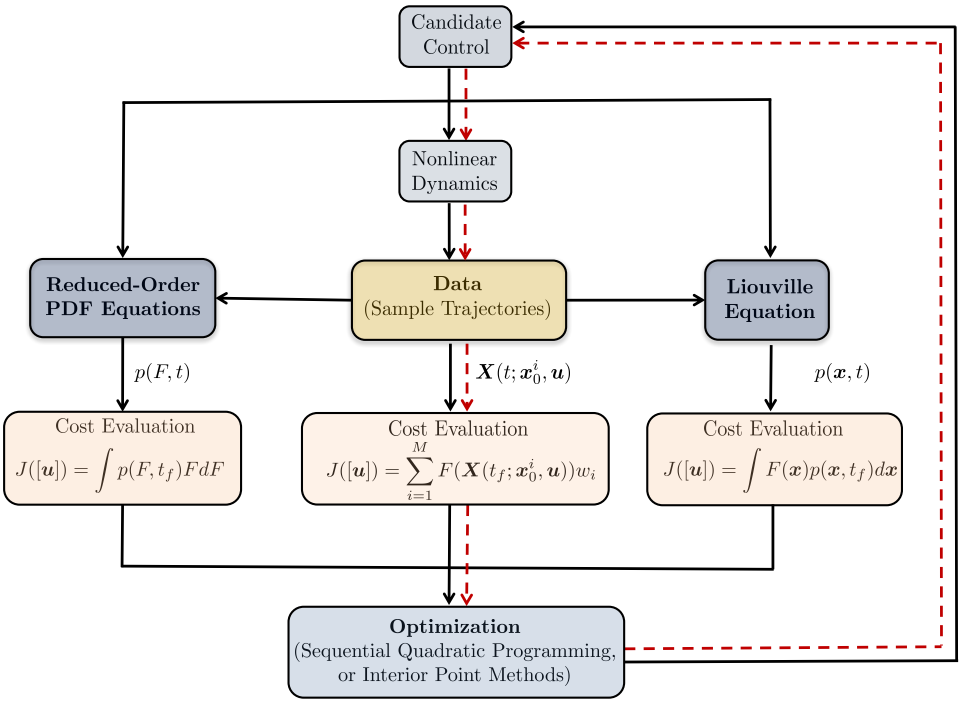


Figure 1: Sketch of the proposed data-driven optimal control architecture. Red dashed arrows indicate the flow of existing methods. The proposed optimal control architecture emphasizes the role of probability density function equations instead of nonlinear dynamics in the control loop. Such paradigm shift opens the possibility to integrate advanced numerical methods for high-dimensional PDF equations with optimization algorithms to mitigate the effects of uncertainty in high-dimensional nonlinear control systems.

Task I Develop scalable software and fast algorithms to compute the numerical solution to high-dimensional PDF equations.

- Parallel C++ class to perform operations between hierarchical tensor formats.
- Parallel sampling strategies and numerical tensor methods to approximate the flow map generated by high-dimensional dynamical systems.

Task II Develop stable algorithms to compute the numerical solution of data-driven PDF equations.

Task III Integrate the numerical algorithms developed in Task I and II into the computational optimal control framework.

Task IV Demonstrate the effectiveness of the proposed data-driven control strategies in applications to disease propagation models in random newtworks of interacting individuals, and swarms of attacking/defending/searching agents.

Figure 2: Research tasks.

2 Project report

In this section we discuss in detail the research activities and results we obtained in Q1 and Q2. The report is organized on a task-by-task basis.

2.1 Parallel C++ class to compute hierarchical tensor formats

As we promised in the quarter-by-quarter breakdown of the research activities we proposed, in Q1 we initiated the development of a parallel C++ class to compute hierarchical Tucker tensor formats. As of today we have available a working serial version of the C++ code, which we tested against the `htucker` Matlab softare developed at École Polytechnique Fédérale de Lausanne (EPFL - Switzerland), and available online at `https://anchp.epfl.ch/htucker`. The numerical results of such tests are summerized in section 2.1.5).

2.1.1 Brief overview of hierarchical tensor methods

Hierarchical tensor methods were originally introduced in [11] to mitigate the dimensionality problem and memory requirements in the numerical representation of high-dimensional functions. A key idea is to perform a sequence of Schmidt decompositions [24] (multivariate SVDs [8]) until the approximation problem is reduced to a product of one-dimensional functions/vectors. To illustrate the method in a simple way, consider a five-dimensional function $f(x_1, \dots, x_5)$. In a hierarchical Tucker tensor representation f is written as

$$f(x_1, \dots, x_5) = \sum_{i_1, \cdots, i_5 = 1}^{r} C[i_1, \dots, i_5] f_{i_1}^1(x_1) f_{i_2}^2(x_2) f_{i_3}^3(x_3) f_{i_4}^4(x_4) f_{i_5}^5(x_5). \tag{1}$$

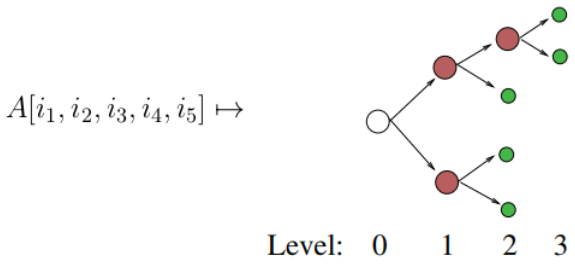


Figure 3: Graph representation of the hierarchical Tucker decomposition of a five-dimensional tensor.

where the 5-dimensional *core tensor* $C[i_1, \ldots, i_5]$ can be factored as a *product of at most three-dimensional tensors*. This is true in an arbitrary number of dimensions. The tensor components $f^k_{i_k}(x_k)$ and the factors of the core tensor can be computed by employing hierarchical SVDs [8, 15, 17] of suitable tensor matricizations, which we will describe in detail in section 2.1.3. Hierarchical tensor expansions can be conveniently visualized by *graphs* (see Figure 3). This is done by adopting the following standard rules: i) a node in a graph represents a tensor in as many variables as the number of the edges connected to it, ii) connecting two tensors by an edge represents a tensor contraction over a certain index. The 5-dimensional function (1) may be evaluated at grid points, e.g., defined within the 5-dimensional standardized hyper-cube $[0, 1]^5$. This basically converts $f(x_1, \ldots, x_5)$ into a 5-dimensional array (tensor), which we formally write as

$$A[i_1, \ldots, i_5] = f(x_1^{i_1}, \ldots, x_5^{i_5}), \qquad (x_1^{i_1}, \ldots, x_5^{i_5}) \in [0, 1]^5 \quad \forall i_j. \tag{2}$$

The basic problem we aim at overcoming with hierarchical tensor methods is the storage requirements of full tensor representations such as (2). To understand how serious such problem is, consider that in dimension 5 if we use 1000 evaluation nodes in each dimension then we need to store $1000^5 = 10^{15}$ floating point numbers (in double precision), which requires approximately 8000 *terabytes* of memory space. From an algorithmic viewpoint, hierarchical tensor methods can be seen as linear algebra techniques (multivariate SVD) to "compress" multivariate arrays of arbitrary dimension into arrays of manageable size. Due to the great practical potential of being able to compress (big) data, we find it a *high priority* to develop a high performance C++ code to implement such algorithms.

2.1.2 The C++ class

In Q1 we studied how the algebraic theory of tensors [11] can be effectively implemented in a C++ class. There are multiple perspectives with which one may view a tensor. A particularly effective one is to view tensors as multi-linear maps from sets of integers into the reals, i.e.,

$$A[i_1, i_2, \ldots, i_d] \in \mathbb{R}, \qquad i_k \in \mathcal{I}_k \qquad \forall k \in 1, \ldots, d.$$

Here, $A[i_1, \ldots, i_d]$ may correspond to the discretization of a multivariate function on a grid (see, e.g., equation (2)). The sets $\mathcal{I}_k$ are called index sets. For finite-cardinality index sets, we let each $i \in \mathcal{I}_k$ range from 0 to $\#(\mathcal{I}_k) - 1$ (where $\#$ denotes the cardinality) to match C++ indexing conventions. A multidimensional array may be stored in C++ in different ways. A conventional approach is to allocate arrays of memory addresses (called "pointer pointers"). Instead, we match the convention given by LAPACK: we allocate a single array of floating point numbers and then manage indexing through the use of "column major form" index weights. This allows lower level control of the memory and makes for more time spent computing and less time spent allocating and de-allocating memory.

Remark 2.1 A multi-index set $\mathcal{I}_1 \times \mathcal{I}_2 \times \ldots \mathcal{I}_d$ is also an index set. The elements of this set are tuples of integers. Since indexed sets of real numbers form a vector space, the notion of tensor described above satisfies the vector space axioms. i.e., tensors are vectors. Basic vector arithmetic operations one might do in Matlab or Numpy are now implemented in the tensor class.

Remark 2.2 (Matlab-like C++ environment) We used the memory management written for the tensor class to implement a column major form matrix class. Moreover, since the data is stored column major, we can now use LAPACK to add in common Matlab-like commands. Some of those included are QR factorization, singular value decomposition (SVD), and matrix multiplication. Combined with operator overloading, this implementation of a C++ matrix object allows for programming in a Matlab-like way, since memory management is taken care of entirely within constructors and destructors. Additional useful routines added in are the Kronecker and Hadamard products, which are not available in LAPACK as of the writing of this report.

2.1.3 Matricizations

Matricization is the process of taking a tensor, and generating a matrix with the same entries. To illustrate this, consider the following steps:

1. Start with a full tensor with an entry like so

$$A[i_1, i_2, \ldots, i_d]$$

2. Group the indexes by permuting them around so that

$$[i_1, i_2, \ldots, i_d] \mapsto \Big[[r_1, r_2, \ldots, r_m], [c_1, c_2, \ldots, c_{d-m}]\Big]$$

   Now we let assign each $[r_1, r_2, \ldots, r_m]$ a natural number, say $r \in \mathbb{N}$. Do the same for $[c_1, c_2, \ldots, c_{d-m}]$, getting the index $c \in \mathbb{N}$.

What we have done is giving each element of the multi-index an order pair of integers. Pairs of integers indexing a set of real numbers is called a matrix. Let's call that matrix B. We have described a map from a tensor to a matrix:

$$A[i_1, i_2, \ldots, i_d] \mapsto B[r, c].$$

The choice of permutation gives us the type of matricization we have done. For example, suppose we have a full tensor with entries $A[i_1, i_2, i_3, i_4]$. If we want to matricize on indexes 1 and 3, then we permute $A[i_1, i_2, i_3, i_4] \mapsto B\big[[i_1, i_3], [i_2, i_4]\big]$, then we count all of the indexes one at a time, in the first two indexes and second two indexes independently. This gives us the matrix $B[r, c]$. Matricization is denoted by the subset of row index labels in a superscript. So in this case, we discussed the $A^{(13)}$ matricization. This is also called the $(1, 3)$-mode matricization.

Remark 2.3 Matricization requires addressing every element of a tensor and allocating multiple arrays to define indexing, floating point storage, and the maximal bounds of an index. Doing this many times can get computationally very expensive. Matlab/Octave get around this by using the built in command `reshape()`. From the Octave documentation on this command, this calls the built in Fortran command `RESHAPE()`, which is a very low level implementation of array memory shape manipulation. In order to beat scaling performance of Matlab and Octave on a desktop computer, it is required to implement a highly efficient array reshaping function for row column form multi-dimensional arrays. An alternative may be to implement a Fortran call into C++, like LAPACK.

2.1.4 Brief description of the HT algorithm

Our goal is to take a full tensor, say $A[i_1, \ldots, i_d]$, and then generate a tree at each node containing smaller tensors which can be used to compute individual entries of A. The tree has one leaf for each of the $1, 2, \ldots, d$ indexes of A (see Figure 3). Each leaf contains a set of basis vectors corresponding to the $1, 2, \ldots, 5$ mode matricizations. The internal (non-root and non-leaf) nodes contain a 3-tensor with projection information for generating a basis corresponding to that node's matricization. So if node has children 1 and 2, then the 3-tensor contains coordinates for generating a basis corresponding to the $(1, 2)$ mode matricization. The exact formula is given in [8] and it is hereafter summarized. Let B_t denote the 3-tensor at node t and U_{s1}, U_{s2} be the matrices containing the basis vectors of the children of node t. Then the i column of the basis U_t is given by:

$$U_t[:, i] = \sum_j \sum_l B_t[i, j, l] \cdot (U_{s1}[:, j] \otimes U_{s2}[:, l]),$$

where the operator : has the Matlab/Octave meaning of "all entries in this index" and $\otimes$ is the Kronecker matrix product. Then (see [8]),

$$B[i, j, k] = \Big\langle U_t[:, i], \big(U_{s1}[:, j] \otimes U_{s2}[:, l]\big) \Big\rangle.$$

All bases are generated by taking the singular value decomposition of a matricization of the full tensor A along the indexes which correspond to a particular node. The orthogonality of the bases from a SVD is what allows us to use relatively simple projections to generate all reduced-order tensors on the tree. To adjust the multi-linear rank of a HT tensor, we simply take fewer left singular vectors to generate the matrices U_t at the leaves of the tree. Lastly, we discuss the root node (white node in Figure 3). This node is similar to the internal nodes, but instead it is only a matrix (2-Tensor), rather than a 3-Tensor. This is because there is no parent node of the root. The projection for the B_t array at the root is the same as the one given above, but the i index has a maximum index of 0, making the expression $B[0, j, k]$. In addition, U_t is a single column vector listing every entry of the full tensor A.

Remark 2.4 Described here is the "root-to-leaves" method for computing a HT decomposition. There is a much faster "leaves-to-root" approach which does successive products onto a "core tensor". It has the same error bounds (see Theorem 2.1) as the approach we just discussed. In particular, the following theorem holds for both HT tensor approximation algorithms.

Theorem 2.1 (HT approximation error [8]). Let A be a real valued tensor of dimension d. Let k be the max prescribed rank on each node of the tree and $\varepsilon > 0$. If there exists a tensor A_{best} of the same rank and $||A - A_{best}|| \leq \varepsilon$, then the singular values of $A^{(t)}$, denoted by σ_i for each node t can be estimated by

$$\sum_{i>k} \sigma_i^2 \leq \varepsilon^2.$$

On the other hand, if the singular values fulfill the theoretical bound $\sum_{i>k} \sigma_i^2 \leq \varepsilon^2/(2d-3)$, then the truncation yields a HT tensor $A_{\mathcal{H}}$ such that $||A - A_{\mathcal{H}}|| \leq \varepsilon$. Thus, the overall accuracy depends on how many singular values we keep in the matrix representation at the leaves. If we drop none, then we obtain an exact representation of the full tensor in the HT form.

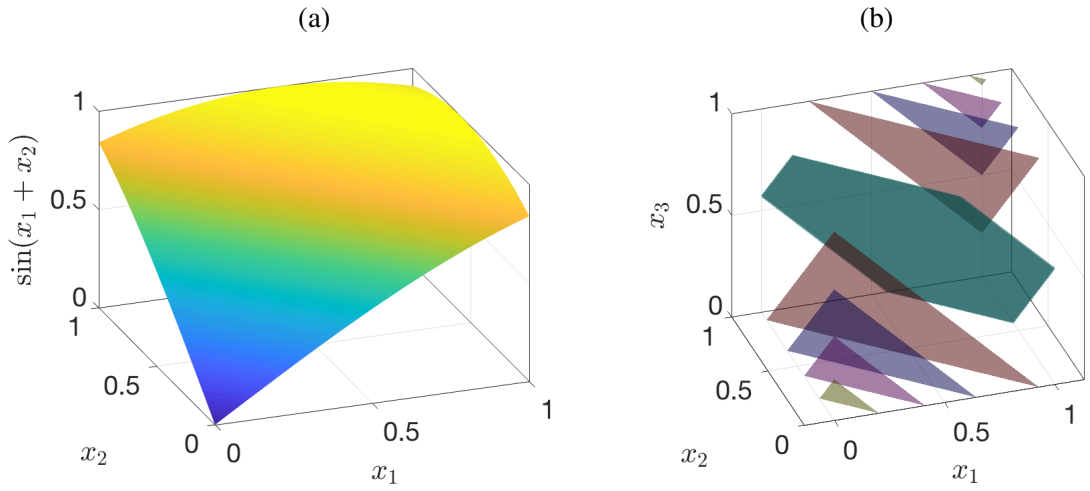


Figure 4: Sine function (3) in 2D (a) and 3D (b). In (b) we show the level sets (iso-surfaces) corresponding to $g = 0$ (green), $g = 0.2$ (magenta), $g = 0.4$ (blue), $g = 0.6$ (purple) and $g = 0.8$ (light green).

2.1.5 Numerical results: serial C++ code

As a first test for the C++ code we developed, we generated a tensor which has entries given by sampling the following scalar function on a uniform grid in the unit hyper-cube $[0, 1]^d$.

$$g(x_1, x_2, \ldots, x_d) = \sin\left(\sum_{i=1}^{d} x_i\right). \tag{3}$$

It was shown in [18] that $g(x_1, x_2, \ldots, x_d)$ can be written as

$$g(x_1, ..., x_d) = \sum_{j=1}^{d} \sin(x_j) \prod_{\substack{i=1 \\ i\neq j}}^{d} \frac{\sin(x_i + \chi_i - \chi_j)}{\sin(\chi_i - \chi_j)}, \tag{4}$$

for any d-tuple of distinct numbers $\{\chi_1, \ldots, \chi_d\}$. Therefore, in principle, $g(x_1, x_2, \ldots, x_d)$ can be written as a fully diagonal HT decomposition with separation rank equal to $r = d$. In Figure 4 we plot the function (3) in two and three dimensions (iso-surfaces).

Next, we perform an analysis of the performance of the HT leaves-to-root decomposition algorithm. In particular, we consider d-dimensional functions of the form (3) and compute the HT decomposition by using both the `htucker` Matlab software available online at `https://anchp.epfl.ch/htucker`) and our newly developed C++ code. Our results are summarized in Figure 5. It is seen that the two implementations yield nearly identical error plots, differing only on the order of machine accuracy. This suggests that our C++ code is mathematically correct, and relatively efficient. In fact, as easily seen from the plots of Figure 5, our code outperforms the Matlab code by 10 times in speed for small dimensions. However, as d increases and we need to perfom more costly matricizations, the built-in Matlab `reshape()` function outperforms our current implentation of the C++ matricization. We are currently investigating possible approaches to overcome this difficulty as discussed above in Remark 2.3.

2.1.6 Parallelization of the HTucker C++ class

To parallelize the algebraic routines of tensor arithmetic we used both OpenMP and MPI communication protocols. This allowed us to store each node of the HT tree sketched in Figure 3 in its own compute

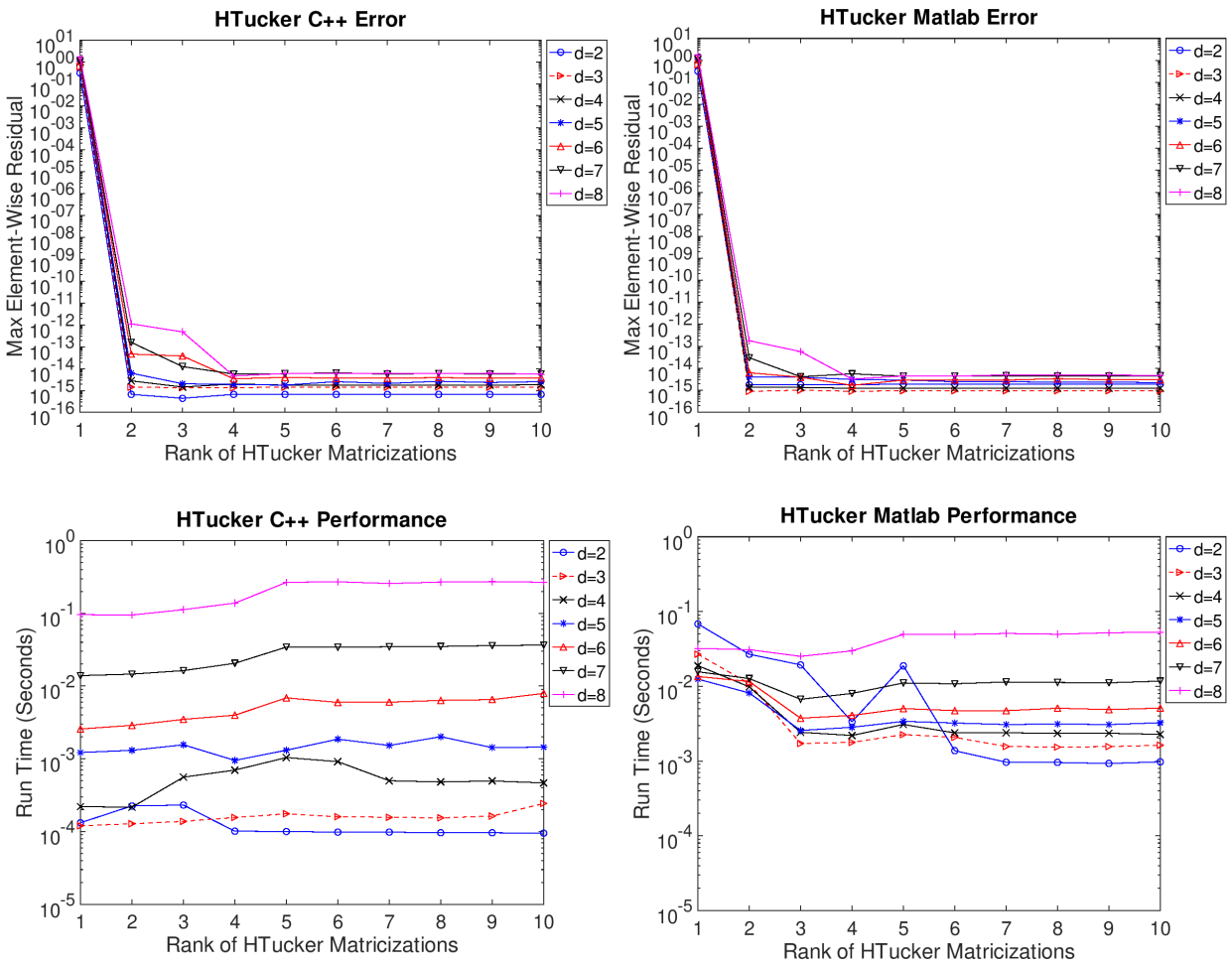
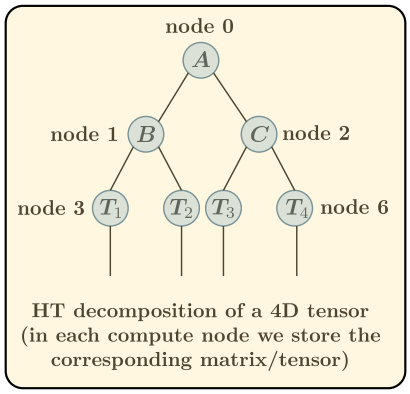


Figure 5: Hierarchical HT decomposition of the sine function (3). Comparison between the `htucker` Matlab software available online at `https://anchp.epfl.ch/htucker` and the serial version our newly developed C++ code in terms of accuracy and execution time.

node as in Figure 6. OpenMP is used so that whatever cores are active on each compute node can perform parallelized linear algebra operations through the use of LAPACK and ScaLAPACK.

Distributed memory implementation

It is natural to attempt to place one tensor or matrix in each compute node of a parallel computer (see Figure 6). This is the approach discussed in the recent paper [9]. In this section we will explicitly state how such a distributed memory implementation can be done using MPI. By distributed memory computer, here we mean a computer which has multiple instances of the same program running. Each instance has an ID number and can send or receive data from any other instance. We will be using these IDs to define what each node does in computing an HTucker decomposition, and how nodes communicate when performing computations on an HTucker tensor. A standard tree data structure consists of nodes containing some data and which point to 2 children nodes, or NULL if no children. In the context of a distributed memory machine, we replace the concept of "pointing to do different memory locations" with storing a set of integers indicating which compute nodes refer to the left child, right child and parent. To illustrate the concept, consider the simple



$$f(x_1, x_2, x_3, x_4) = \sum_{l,m=1}^{r} A_{lm} \varphi_l(x_1, x_2) \psi_m(x_3, x_4)$$
$$= \sum_{l,m,i,j,p,q}^{r} A_{lm} B_{lij} C_{mpq} T_1^i(x_1) T_2^j(x_2) T_3^p(x_3) T_4^q(x_4)$$

$\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$, and $\{\boldsymbol{T}_1, \ldots, \boldsymbol{T}_4\}$, are computed by hierarchical SVD

Figure 6: Parallel implementation of the HTucker C++ class. Dimension tree corresponding to a 4-tensor.

example shown in Figure 6 of a tree corresponding to a 4-tensor. Iterating from left to right in each layer, we correspond an index to each tree node. Each node contains a data structure with 3 integers and the relevant tensor objects for HTucker decomposition. For example, node 1 contains a parent ID of 0, a left child ID of 3, and a right child ID of 4. For the root, the parent ID is set to 0, which is the same as its own ID. For the leaves, the children ID numbers are set to -1. An algorithm for assigning unique ID numbers for all nodes for arbitrary dimension trees using the scheme outlined here is implemented as a dependency for the HTuckerMPI C++ object. The algorithm for computing the HTucker decomposition on a distributed tree is largely the same as the method we described in Q1. The difference lies in how the data is stored and transferred in the computer. Any time where a matrix, tensor, or some related data (e.g. number of components in an array) is required from a parent/child node, an MPI message is passed. Using this, we can initialize a tensor on node 0, and then send data to the rest of the tree. To this effect, we compute all the required matricizations simultaneously. Then SVDs are all done simultaneously and the left singular vectors are sent to the respective parent nodes. As for how this is accomplished in C++, we store an HTuckerMPI object on each compute node. Each object contains either the root matrix, a transfer tensor, or the leaf basis matrices. Each node using this object as an interface to communicate with all other nodes on the tree. Addition is accomplished by concatenating tensors as is described by the Matlab HTucker manual. The only operations required at the time taken to copy two summands into a new HTuckerMPI object. Truncation of an HTucker tensor to another HTucker tensor is similar to going from full tensor to HTucker (see [15]). First, we generate a set of matrices called Gramians for each node which are roughly equivalent to the matricizations. Then we use these matrices to generate new matrices containing left singular vectors. Finally we only the child frames in similar manner as stated in (2.1.3). As mentioned above, OpenMP is also used in the parallelization process. Each compute node is also given the capability to compute with shared memory in parallel. This is to say that we can take advantage of the parallelizations used in LAPACK for computing, e.g., the SVD and the QR factorization. On the workstation used in the tests below, we have a Intel i9-7980XE with 18 CPU cores with Hyper-Threading of 2 processes per core. So our Linux operating system registers a total of 36 ”logical cores.” If we are to use the 4 dimensional tensor example above, then we need to used 7 MPI compute nodes (one for each tree node). This number tells us how many cores we can allocate to parallelizing with OpenMP. The allocation is simple, diving 36 by 7 we have maximally 5 OpenMP processes per MPI node and then one left over.

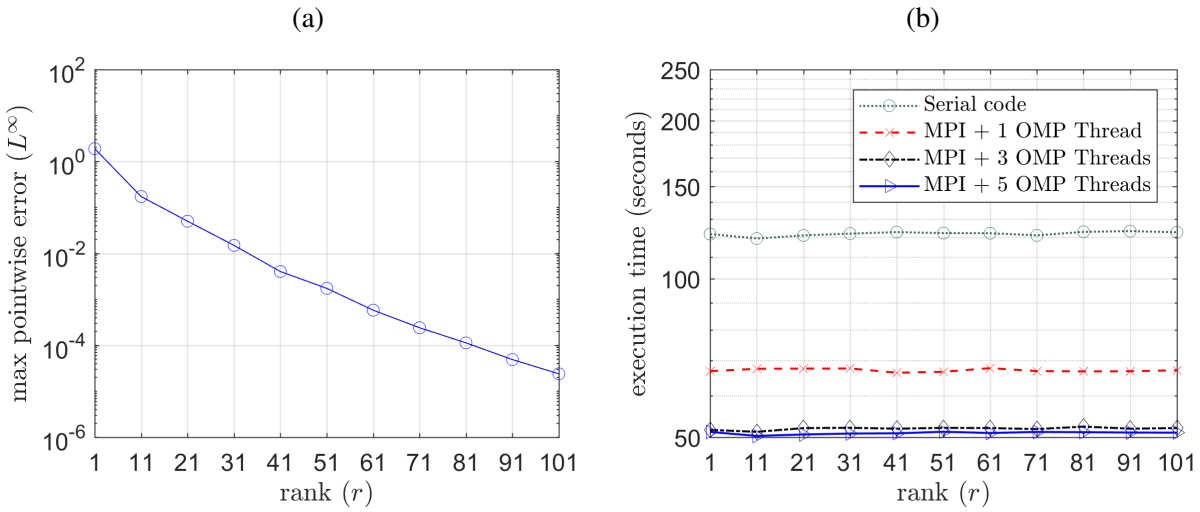


Figure 7: Performance of the parallel HTucker C++ class with "root-to-leaves" [15] truncation in computing the tensor decomposition of the function (5). Here we use 7 MPI nodes, with up to 5 OpenMP threads in each node.

2.1.7 Numerical results: parallel C++ code

We consider the following non-separable function to study parallel versus serial performance of the C++ code we developed

$$g(x_1, x_2, x_3, x_4) = \exp\left[\frac{\sin(5x_1x_2)\cos(5x_3)}{1+\cos(10x_1x_4)^2}\right]. \tag{5}$$

We sample g on a $60 \times 60 \times 60 \times 60$ grid in $[0,1]^4$. This yields a 4D numerical tensor with 12.36 million entries, which requires 103.68 Mb of storage if we use double precision floating point numbers. We tested accuracy and computational time for several different separation ranks. We start at rank 1 and then increase the rank by 10 every iteration, until 101. Our results are summarized in Figure 7. In particular, Figure 7(a) shows that the maximum pointwise error decays more or less exponentially fast with the separation rank r. Such error decay, is not obviously affected by the number or OpenMP threads within each compute node. In Figure 7(b) it is seen that the parallel HTucker code indeed outperforms the serial version by a significant margin. Specifically, we can see a reduction by $1/2$ in execution time. This is not the full $1/7$ one would expect since a large overhead is introduced by telling different processing nodes to send data back and forth. We are currently working on *optimizing (minimize) communication between the compute nodes*. Even so, with this overhead we make significant gains in speed. Also, the parallel code is more suited to larger and larger problems. If the code spends more time computing on independent cores than passing data between cores, then we see better performance. For this particular tensor, the two implementations scale constant with rank. This is because the last step, which actually depends on rank, is the series of projections explained in section 2.1.3 . However, this step takes far less time than computing all of the matricizations and singular value decompositions, which are not currently programmed to scale with rank. Observing the performance of the parallel code, we also see that adding more cores per compute node has a significant impact on compute time, going down from around 60 seconds to 50 seconds.

2.1.8 Application to a 4D Liouville equation

Consider the following four-dimensional initial/boundary value problem for the Liouville equation on the periodic cube $\mathcal{D} = [-1, 1]^4$

$$\begin{cases} \dfrac{\partial p(t,\boldsymbol{x})}{\partial t} + \boldsymbol{G} \cdot \nabla p(t,\boldsymbol{x}) = 0 \qquad t \geq 0, \quad \boldsymbol{x} \in \mathcal{D}, \\ p(0,\boldsymbol{x}) = p_0(\boldsymbol{x}) = \dfrac{1}{(4\pi^2\sigma^4)} \exp\left[-\dfrac{x_1^2 + x_2^2 + x_3^2 + x_4^2}{2\sigma^2} \right], \qquad \sigma = 2. \end{cases} \tag{6}$$

By using the method of characteristics, it is straightforward to obtain the following exact solution (with constant $\boldsymbol{G}$)

$$p(t,\boldsymbol{x}) = p_0(\boldsymbol{x} - \boldsymbol{G}t). \tag{7}$$

Taking partial derivatives of $p(t,\boldsymbol{x})$ in a discrete form is done through the use of a "μ-mode" product $\circ_\mu$, which is performed by taking the μ-mode matricization, applying the differentiation matrix to the resulting operator. For example, the partial derivative in x_1 is:

$$\left.\frac{\partial p}{\partial x_1}\right|_{(t,[x_1^i,x_2^j,x_3^k,x_4^w])} \approx (\boldsymbol{D} \circ_1 \boldsymbol{P}(t))[i,j,k,l], \tag{8}$$

where we denoted by $\boldsymbol{D}$ the one-dimensional pseudospectral (Fourier) differentiation matrix, and with $\boldsymbol{P}(t)$ the full tensor (with all indexes) at time t. The semi-discrete form of the initial/boundary value problem 6 can be compactly written in an HTucker form as

$$\frac{d\boldsymbol{P}(t)}{dt} = -\sum_{k=1}^{4} G_k \boldsymbol{D} \circ_k \boldsymbol{P}(t). \tag{9}$$

To integrate the ODE system (9) in time, we use the the second-order explicit Adams-Bashforth scheme

$$\boldsymbol{P}(t_{n+1}) = \boldsymbol{P}(t_n) - \frac{\Delta t}{2} \sum_{k=1}^{4} G_k \boldsymbol{D} \circ_k \left(3\boldsymbol{P}(t_n) - \boldsymbol{P}(t_{n-1})\right). \tag{10}$$

By the properties of pseudo-spectral methods, we expect that accuracy depends on differentiability in space. In particular, since the initial condition is infinitely differentiable and numerically zero on the boundary for sufficiently small σ, we expect exponential convergence in space. As for using HTucker to solve this problem, it can be shown that multiplying a matrix into the μ leaf in the HTucker decomposition is equivalent to taking the μ-mode product with the full tensor; summing is simply concatenation; and scalar multiplication can be accomplished by scaling the root node's matrix by a real number. Since the data stored at each step grows with concatenation if we do not truncate, we truncate to a given max rank at the end of every iteration. In Figure 8 we plot a few sections of the solution to (6) in the x_1x_2-plane at different times. The rank of the HTucker decomposition is chosen to be 1, 2, and 3. Note that since a the Gaussian initial condition is fully separable, it can be represented exactly with a rank 1 tensor format. Thus, raising rank does not improve accuracy, but increases computation time since more data copying for each addition and also more vectors operations. We see all this in Figure 9, where we plot the execution time needed to advect the solution for 10^6 time steps (one cycle) This number was chosen because after this many iterations the maximum pointwise error is of order $O(10^{-4})$, small enough to be a fair estimate of the solution. Next, we study scaling with with the number of grid points, to see how the different algorithms handle growing problem size. In Figure 9 we see the execution time of the serial C++ algorithms grows roughly with power $1/2$. On the other hand, the execution time of HTucker grows much slower. This is because essentially we don't need to

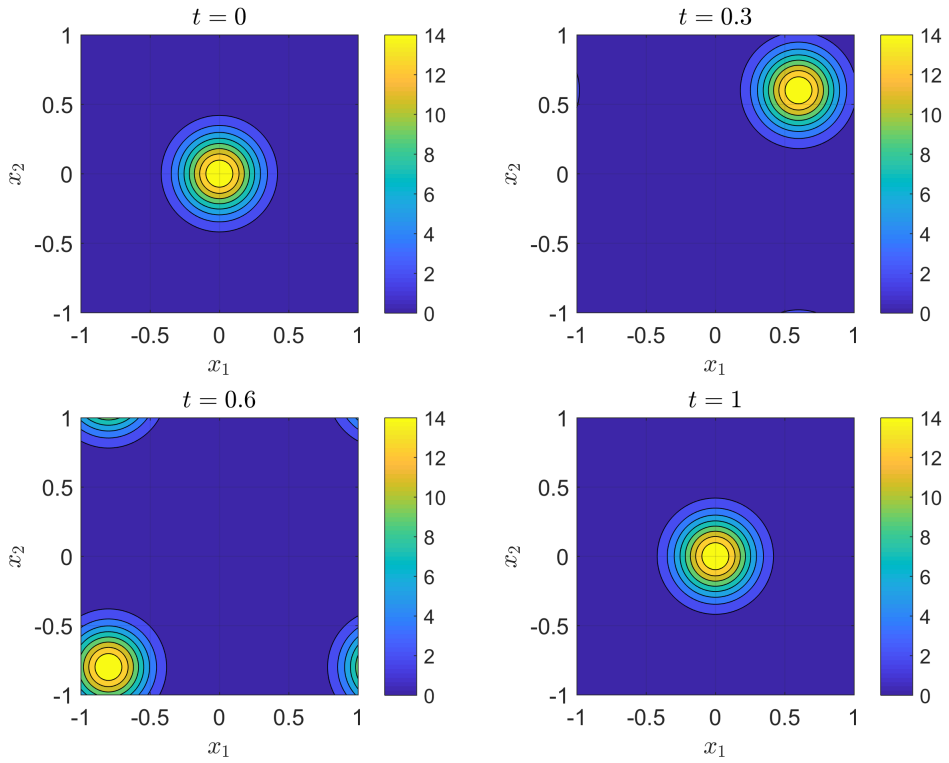


Figure 8: Solution of the initial/boundary value problem (6). Shown are sections of the solution in the x_1x_2-plane at different times. Here x_3 and x_4 are both set equal to $2t$.

compute any of the additions, and the modal products need only to be applied to the leaves. We see that the computing time levels off entirely for a rank 1 representation. For more complicated problems, the optimal rank in general depends on time, suggesting that the solution may increase or reduce its separability as time integration proceeds. In this case, we can adaptively compute such optimal rank on-the-fly based on fast error estimators.

2.2 Data-driven methods to compute PDFs and flow maps

Consider the n-dimensional system of autonomous first-order ordinary differential equations,

$$\dot{\boldsymbol{x}} = \boldsymbol{G}(\boldsymbol{x}(t)), \qquad \boldsymbol{x}(0) = \boldsymbol{x}_0 \sim p_0(\boldsymbol{x}), \tag{11}$$

where $p_0(\boldsymbol{x})$ is a given probability density function (PDF). For any fixed initial condition, the solution is determined by the *flow map*

$$\boldsymbol{x} = \boldsymbol{\Phi}(\boldsymbol{x}_0, t), \tag{12}$$

which is a function of both the initial condition $\boldsymbol{x}_0$ and time t. It can be shown [2] that the *forward flow map* satisfies the flow map equation

$$\frac{\partial \boldsymbol{\Phi}(\boldsymbol{x}_0, t)}{\partial t} - (\boldsymbol{G}(\boldsymbol{x}_0) \cdot \nabla)\, \boldsymbol{\Phi}(\boldsymbol{x}_0, t) = 0 \qquad \boldsymbol{\Phi}(\boldsymbol{x}_0, 0) = \boldsymbol{x}_0. \tag{13}$$

Similarly, the *inverse flow map* satisfies the initial value problem

$$\frac{\partial \boldsymbol{\Phi}_0(\boldsymbol{x}, t)}{\partial t} + (\boldsymbol{G}(\boldsymbol{x}) \cdot \nabla)\boldsymbol{\Phi}_0(\boldsymbol{x}, t) = 0, \qquad \boldsymbol{\Phi}_0(\boldsymbol{x}, 0) = \boldsymbol{x}. \tag{14}$$

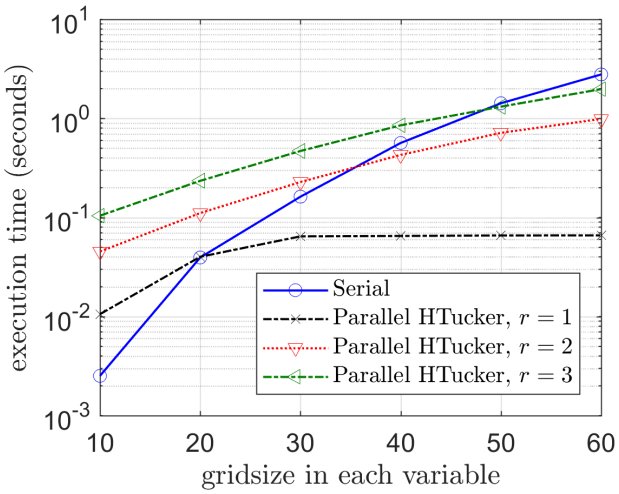


Figure 9: Computational time needed to advect the solution back to its initial position (reached at $t = 1$) versus the number of collocation point in each variable x_i. To get from $t = 0$ to $t = 1$ we perform 10^6 time steps.

When considering uncertainty, the PDF of the state vector $\boldsymbol{x}$ at time t can be found by solving the Liouville equation

$$\frac{\partial p(\boldsymbol{x}, t)}{\partial t} + \nabla \cdot [p(\boldsymbol{x}, t)\boldsymbol{G}(\boldsymbol{x})] = 0. \tag{15}$$

The analytical solution to (15) can be expressed with the method of characteristics as

$$p(\boldsymbol{x}, t) = p_0(\boldsymbol{\Phi}_0(\boldsymbol{x}, t)) \exp\left[-\int_0^t \nabla \cdot \boldsymbol{G}(\boldsymbol{\Phi}(\boldsymbol{x}_0, \tau))d\tau\right], \tag{16}$$

where $\boldsymbol{\Phi}_0(\boldsymbol{x}, t)$ is the *inverse flow map* [7] satisfying (14). From (16), we see that if the system is volume-preserving, i.e., if $\nabla \cdot \boldsymbol{G} = 0$ then we have

$$p(\boldsymbol{x}, t) = p_0(\boldsymbol{\Phi}_0(\boldsymbol{x}, t)). \tag{17}$$

This means, in particular, that the level sets of p_0 are preserved throughout the dynamics. This allows us to track the support of the joint PDF $p(\boldsymbol{x}, t)$ by propagating forward in time the almost-zero level set.

Remark 2.5 For a large class of control systems, e.g., control affine systems, it is possible to design state feedback control to make the system (11) divergence-free. Such property can be explored to design optimal closed-loop controls that leverage divergence-free dynamics.

2.2.1 Data-driven approximation of probability density functions using deep neural nets

Machine learning offers an efficient way to compute data-driven solutions of partial differential equations [23]. In Q1 we implemented several algorithms that leverage deep neural networks (designed in TensorFlow [1]) to approximate the PDF of prototype low-dimensional dynamical systems. The algorithms are built upon two different types of neural nets

- Data-driven neural nets;
- Physics-informed data-driven neural nets (PINN).

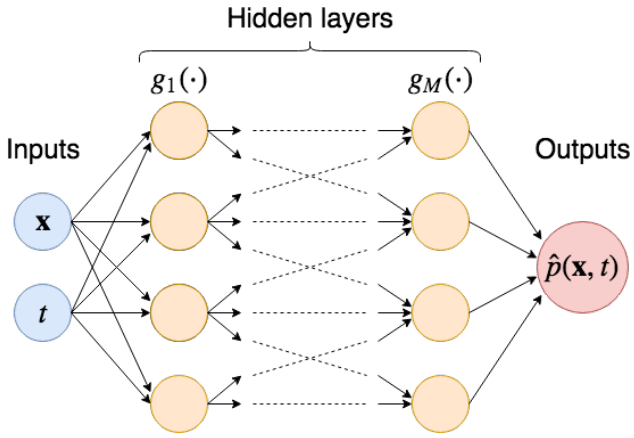


Figure 10: Architecture of a feed-forward neural net for approximating $p(\boldsymbol{x}, t)$ by a composition of functions: $\hat{p}(\boldsymbol{x}, t) = g_M \circ g_{M-1} \circ \ldots g_1(\boldsymbol{x}, t)$. Each layer (function) g_j consists of a set of parameters $\boldsymbol{\theta}_j = \{\boldsymbol{W}_j, \boldsymbol{b}_j\}$, weights and biases on the outputs of the previous layer g_{j-1}, along with a nonlinear *activation function*, e.g., the sigmoid function or the hyperbolic tangent. For example, $g_2 \circ g_1(\boldsymbol{x}, t) = \tanh[\boldsymbol{W}_2 \cdot g_1(\boldsymbol{x}, t) + \boldsymbol{b}_2] = \tanh[\boldsymbol{W}_2 \cdot \tanh(\boldsymbol{W}_1 \cdot [\boldsymbol{x}, t] + \boldsymbol{b}_1) + \boldsymbol{b}_2]$. The parameters are optimized during model training so that the output $\hat{p}\left(\boldsymbol{x}^{(i)}, t^{(i)}\right)$ is as close as possible, in some norm, to the training data $p\left(\boldsymbol{x}^{(i)}, t^{(i)}\right)$.

In the first case, the PDF of the system is estimated by training the neural net sketched in Figure 10 entirely with sample paths[1] of (11). In practice, we minimize a cost function of the form

$$MSE_{\text{data}}(\boldsymbol{\theta}_1, ..., \boldsymbol{\theta}_M, t) = \frac{1}{N_d} \sum_{k=1}^{N_d} \left[\log\left(p(\boldsymbol{x}^{(k)}, t)\right) - \log\left(\hat{p}(\boldsymbol{x}^{(k)}, t)\right) \right]^2, \tag{18}$$

where $p(\boldsymbol{x}^{(k)}, t)$ is obtained by solving (15) with the method of characteristics, $\hat{p}(\boldsymbol{x}^{(k)}, t)$ is the neural net representation

$$\hat{p}(\boldsymbol{x}, t) = g_M \circ g_{M-1} \circ \ldots g_1(\boldsymbol{x}, t) \tag{19}$$

evaluated at $\boldsymbol{x} = \boldsymbol{x}^{(k)}$, N is the number of sample paths, and $\boldsymbol{\theta}_j = \{\boldsymbol{W}_j, \boldsymbol{b}_j\}$ are the free parameters in the j-th activation function. In (18), $\boldsymbol{x}^{(k)} = \boldsymbol{\Phi}(\boldsymbol{x}_0^{(k)}, t)$ denotes the the position of the particle $\boldsymbol{x}_0^{(k)}$ at time t, which can be easily determined by integrating system (11) from the initial condition $\boldsymbol{x}_0^{(k)}$.

In the second case, i.e., in the *physics-informed data-driven neural net (PINN)* setting, we augment the cost function with a penalty term that represents the magnitude of the residual we obtain when we substitute the neural net representation (19) into the Liouville equation (15), i.e.,

$$R(\boldsymbol{x}, t) = \frac{\partial \hat{p}(\boldsymbol{x}, t)}{\partial t} + \nabla \cdot (\boldsymbol{G}(\boldsymbol{x}) \hat{p}(\boldsymbol{x}, t)) \,. \tag{20}$$

In this case, the cost functional we consider is

$$MSE_{\text{PINN}}(\boldsymbol{\theta}_1, ..., \boldsymbol{\theta}_M, t) = MSE_{\text{data}}(\boldsymbol{\theta}_1, ..., \boldsymbol{\theta}_M, t) + \mu MSE_{\mathcal{L}}(\boldsymbol{\theta}_1, ..., \boldsymbol{\theta}_M, t), \tag{21}$$

[1] Training the neural net as shown in Figure 10 can be done at a specific time t, e.g., at final time or at an entire sequence of time instants between two prescribed times. In the latter case we aim at learning and the entire dynamics of the joint PDF $p(\boldsymbol{x}, t)$, i.e., from $t = 0$ to $t = t_f$.

where μ is penalty parameter and

$$MSE_{\mathcal{L}}(\boldsymbol{\theta}_1, ..., \boldsymbol{\theta}_M, t) = \frac{1}{N_c} \sum_{k=1}^{N_c} \Big[R(\boldsymbol{x}^{(k)}, t) \Big]^2 \tag{22}$$

is the mean square error associated with the residual of the Liouville equation. As before, $\boldsymbol{x}^{(k)} = \boldsymbol{\Phi}(\boldsymbol{x}_0^{(k)}, t)$ ($\boldsymbol{\Phi}$ is the flow map generated by (11)). The residual (20) can be easily evaluated by using automatic differentiation techniques applied to (19).

Prototype dynamical system In the following sections we study the effectiveness of PINN and other methods to predict the PDF and the flow map of the two-dimensional divergence-free nonlinear dynamical system

$$\begin{cases} \dot{x} = 2xy - 1 \\ \dot{y} = -x^2 - y^2 + \mu \end{cases} \tag{23}$$

The phase portraits of the system (23) are plotted in Figure 11 for different values of the parameter μ. We observe that the system undergoes two saddle node bifurcations at $\mu = 1$.

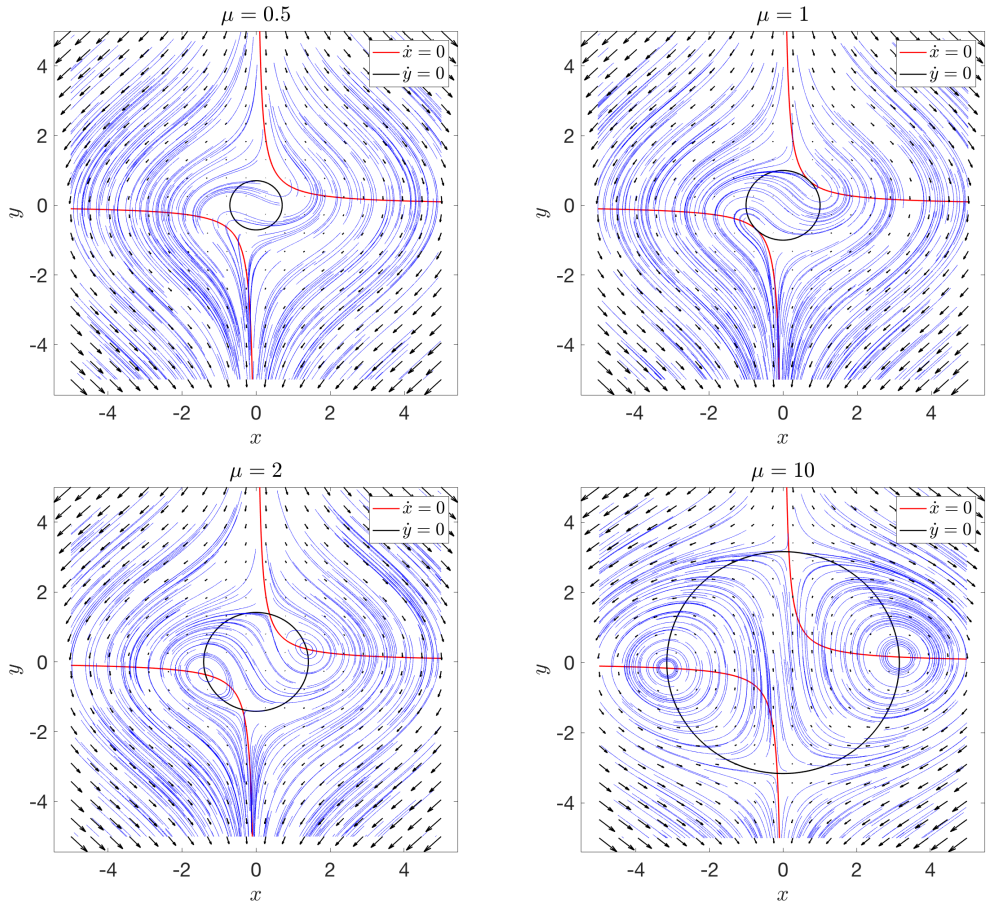


Figure 11: Phase portraits of the system (23) for different values of μ.

2.2.2 Brief description of the machine learning algorithms we implemented

In this section, we outline our first implementation of the data-driven machine learning algorithms to estimate the joint PDF of the solution to the dynamical system (11).

Data-driven machine learning This algorithm is purely based on data, i.e., sample trajectories of (11), *without* the PDE constraint represented by the Liouville equation. Specifically, we use the bare-bones feed-forward neural net sketched in Figure 10 with the cost function defined in (18).

Physics-informed data-driven machine learning This algorithm operates as follows:

1. Set up two deep neural nets using, e.g., TensorFlow [1].

   - The first net learns an approximation of PDF, $\hat{p}(\boldsymbol{x}, t) \approx p(\boldsymbol{x}, t)$. To this end, we generate a training data set $\left\{\left(\boldsymbol{x}^{(i)}, t^{(i)}\right), p\left(\boldsymbol{x}^{(i)}, t^{(i)}\right)\right\}$, $i = 1, \ldots, N_d$, by forward and/or backward integration of (11) from many different spatio-temporal points $\left(\boldsymbol{x}^{(i)}, t^{(i)}\right)$ and evaluation of $p\left(\boldsymbol{x}^{(i)}, t^{(i)}\right)$ by (16). This is a *supervised learning* scenario with inputs $\left\{\left(\boldsymbol{x}^{(i)}, t^{(i)}\right)\right\}$ and outputs $\left\{p\left(\boldsymbol{x}^{(i)}, t^{(i)}\right)\right\}$ (see Figure 10).
   - The second net is constructed using TensorFlow's built-in automatic differentiation to estimate the partial derivatives of $\hat{p}$. It has the same parameters as the first net, and penalizes approximations $\hat{p}$, which in general does not satisfy the Liouville equation (15).

2. The two nets are trained simultaneously with the cost function (21) (see Figure 12).

3. Once training is complete, we can use the first net to obtain fast approximations to the probability density function at any point (see Figure 13).

2.2.3 Generating training data

Generating training data for neural nets is not exactly a straightforward process. Backward integration from points $\boldsymbol{x}^{(i)}$ may yield initial points with rather arbitrary positions. In this case, numerical integration will take a very long time and may eventually fail. Forward integration from a set of points $\boldsymbol{x}_0^{(i)} \sim p_0(\boldsymbol{x})$ will always yield well-defined data with non-zero probability, so long as the dynamical system (11) meets some basic conditions. However, this data may not be well-structured for the purpose of representing $p(\boldsymbol{x}, t)$. For divergence-free systems, feed-forward sampling can provide a reasonable approximation of the support of $p(\boldsymbol{x}, t)$. Hence we can construct a convex hull around the points $\boldsymbol{x}^{(i)} = \boldsymbol{\Phi}\left(\boldsymbol{x}_0^{(i)}, t\right)$ to estimate this support. We can then "fill in" the rest of the convex hull by backward integration. We need to find other methods for systems with divergence, since for these it is harder to estimate the support of $p(\boldsymbol{x}, t)$ directly from $\boldsymbol{\Phi}\left(\boldsymbol{x}_0^{(i)}, t\right)$.

2.2.4 Numerical results

In this section we present the numerical results we obtained by training the feed-forward and PINN neural nets with sample trajectories of (23) for the purpose of predicting the joint PDF of the state vector. In particular, we tested the following different scenarios:

- Prediction of the joint PDF at final time with feed-forward neural nets;

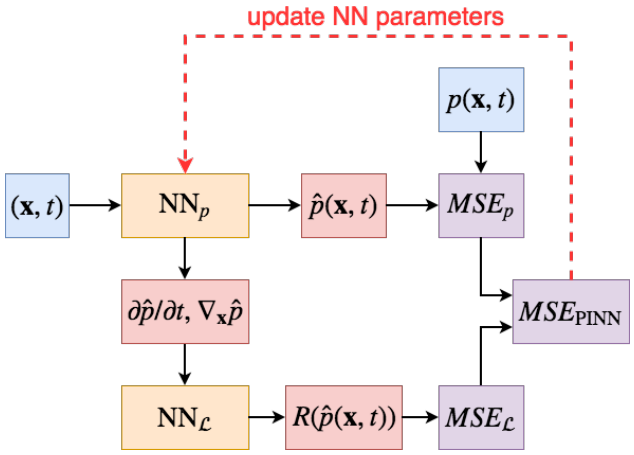


Figure 12: Training physics-informed neural nets (PINN). The neural net NN_p representing $\hat{p}(\boldsymbol{x}, t)$ is supplemented by a second neural net $\mathrm{NN}_{\mathcal{L}}$ which is a *derivation* of NN_p obtained by automatic differentiation and the Liouville equation (15). The free parameters of NN_p are obtained by minimizing the mean-square error MSE_{PINN} defined in (21).

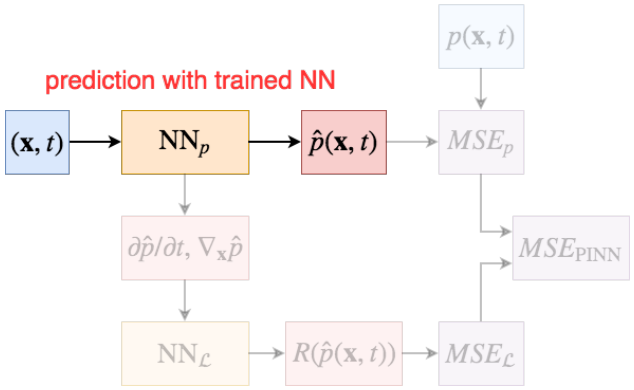


Figure 13: Predicting with trained physics-informed neural nets (PINN).

- Prediction of the full dynamics of the joint PDF with feed-forward neural nets;
- Prediction of the full dynamics of the joint PDF with physics-informed neural nets (PINN).

Hereafter, we analyze each case in detail, and discuss our numerical findings.

Prediction of the joint PDF at final time with feed-forward neural nets

By using the method of characteristics, we randomly generated PDF data points at time $t = t_f$ (t_f variable) for the two-dimensional divergence-free test system (23) with $\mu = 5$. We chose the initial PDF $p_0(x, y)$ to be the product of two independent Gaussians with means $\mu_x = \mu_y = 0.75$ and variances $\sigma_x^2 = \sigma_y^2 = 0.25$. We learned the final time PDF using standard Tensorflow [1] without any secondary physics-informed neural net [23]. We used an L-BFGS [3] optimizer and a $\tanh()$ activation function for the neural net, and varied the configurations of the hidden layers to increase performance. In addition, before feeding data to the neural net, we mapped spatial data (x, y) to $[-1, 1]$, where $\tanh()$ is steepest, and took the logarithm of the probability data. Learning the log probability ensured that the model would preserve positivity of the PDF.

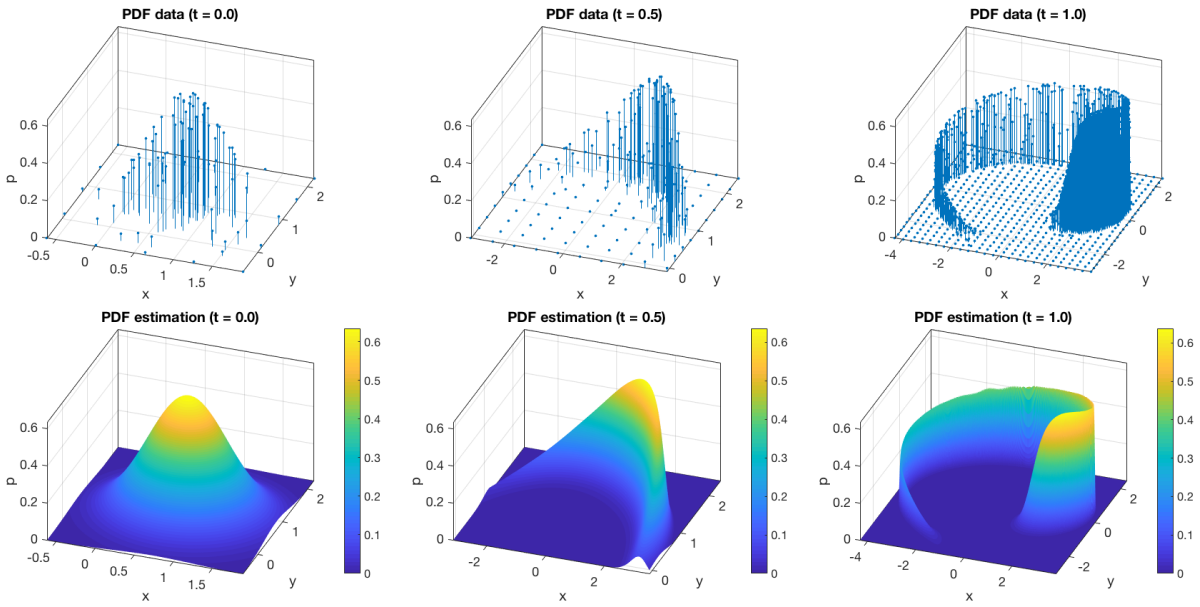


Figure 14: Training data (top, 100, 200, and 2000 points, left to right) and 1000×1000 reconstructions (bottom) of initial PDF $p_0(x, y)$, $p(x, y, t = 0.5)$, and $p(x, y, t = 1.0)$, left to right, using 8 hidden layers with 20 neurons each. Root mean squared error (RMSE) with respect to $N_v = 2500$ validation data on a 50×50 grid: 4.60 e–04, 4.89 e–04, and 2.56 e–04, left to right. RMSE is taken with respect to the actual probability, not the log probability.

Where the probability was too small, we set it to a minimum threshold $e^{-15} \approx 3.06 \times 10^{-7}$, so that there wouldn't be any numerical problems when we took the logarithm.

Data generation We generated initial conditions $\left(x_0^{(i)}, y_0^{(i)}\right) \sim p_0(x, y)$, $i = 1, \ldots, N_s$, and numerically integrated to $t = t_f$ to obtain $\left(x^{(i)}, y^{(i)}\right)$. The probability data $p\left(x^{(i)}, y^{(i)}, t = t_f\right)$ was obtained using the solution to the Liouville equation (16). We then enclosed these points in a rectangle and built an $N_m \times N_m$ uniform grid of points $\left(x^{(i)}, y^{(i)}\right)$, $i = 1, \ldots, N_m^2$, from which we propagated backwards and used the Liouville equation to get probability data. This provides a total of $N_d = N_s + N_m^2$ training data. For model validation, we checked the neural net predictions on a uniform grid over the same area as the training data.

Model training In Figure 14, we visualize the training data and neural net reconstruction of the initial PDF $p_0(x, y)$ and the final time PDF $p(x, y, t = t_f)$ for $t_f = 0.5$ and $t_f = 1.0$. We observe that the dynamics (23) rapidly advect the smooth Gaussian into a thin curve. In Table 1 and 2 we present speed and accuracy results for estimating $p(x, y, t = 1.0)$ depending on the architecture of the neural net. For Table 3 and 4 we fixed the architecture and varied the amount of training data fed to the net. This let us test the sensitivity of the net to data availability and determine good ratios of forward propagated data to backward propagated data.

Discussion We immediately observe that the initial Gaussian is very easy to reconstruct. It can be learned to $O(10^{-4})$ accuracy in seconds with only a hundred or so training data points. At $t_f = 0.5$, we can still reliably reconstruct the PDF using only 200 points. As we advance time, however, the regression problem becomes more difficult as the approximate support of $p(x, y, t)$ advects into a thin curve with steep slopes. Thus we require more training data, and deeper neural nets are more reliable for learning the PDF. Table 1

		Neurons per layer			
		10	20	30	40
Hidden layers	2	3.27 e–02	1.29 e–02	1.68 e–02	6.66 e–03
	4	1.58 e–02	1.10 e–03	1.21 e–03	2.03 e–03
	6	1.42 e–03	8.18 e–04	7.50 e–04	4.75 e–03
	8	1.54 e–03	4.01 e–04	1.91 e–03	5.24 e–03
	10	1.08 e–03	2.33 e–03	5.44 e–03	2.21 e–03
	12	2.70 e–03	5.31 e–03	2.42 e–03	7.97 e–03

Table 1: Validation RMSE results for estimating the final time PDF $p(x, y, t = 1.0)$, depending on the structure of the neural net. The total amount of training data N_d is fixed at 2000 with $N_s = 1100$ points for the forward propagation and $N_m \times N_m = 30 \times 30$ uniform grid for backward propagation. Validation RMSE is measured for $N_v = 2500$ data points on a 50×50 uniform grid. Note that because training data are randomly generated, weights are randomly initialized, and the optimization problem is not convex, performance can vary each time the model is trained.

		Neurons per layer			
		10	20	30	40
Hidden layers	2	22 s	69 s	88 s	97 s
	4	35 s	60 s	78 s	90 s
	6	35 s	44 s	108 s	104 s
	8	22 s	82 s	99 s	172 s
	10	50 s	60 s	120 s	160 s
	12	46 s	76 s	158 s	195 s

Table 2: Training time of a neural net for estimating the final time PDF $p(x, y, t = 1.0)$, depending on the structure of the net. Training data generation ($N_d = 2000$ points) and validation data generation ($N_v = 2500$ points) by numerical integration take approximately 3 and 6 seconds each, while evaluating the trained net for 90000 outputs takes a fraction of a second. Listed runtimes are for a naïve single CPU implementation using TensorFlow 1.8 [1] on a 2012 MacBook Pro with 2.5 GHz Intel Core i5 processor and 4 GB RAM.

shows that deeper neural nets tended to be more accurate; in particular we should use at least 6 hidden layers for this problem. Meanwhile, there appears to be little benefit to increasing the number of neurons per layer, except for several models which achieved $O(10^{-4})$ RMSE on fortunate training sessions. Table 2 reveals that we can make nets deeper with minimal increase in training time, whereas increasing the width of nets is costly. At the same time, all the trained nets can produce *one million* outputs for plotting in a fraction of a second. Meanwhile, generating only $N_v = 2500$ validation data points by numerical integration took around 8 seconds on average. Thus, as we expect, neural nets are slow to train but incredibly fast once they are trained. In Table 1 we observe that, apart from a few outliers, increasing the number of data points generally improved accuracy. However, the kinds of training points used was also relevant. As discussed previously, using more forward propagation points provides more resolution of the PDF within the approximate support, while using more meshgrid points yields better boundaries for this region. There appeared to be a limit to the usefulness of increasing the fineness of the meshgrid, however. Perhaps using too many grid points gave the net too much weight on putting zeros outside of the approximate support, and not enough weight to learning the shape of the PDF within the approximate support. Increasing the amount of data points had some relation to increased training time, but not as much as one might expect.

		Forward propagation points						
		800	900	1000	1100	1200	1300	1400
Grid points	400	6.49 e–04	3.17 e–03	4.40 e–03	1.11 e–02	1.31 e–02	9.52 e–03	3.11 e–04
	625	1.85 e–02	1.16 e–03	4.45 e–02	1.28 e–03	6.66 e–04	6.69 e–03	1.37 e–02
	900	6.18 e–03	5.89 e–03	4.51 e–04	4.01 e–04	6.95 e–04	8.65 e–04	5.84 e–04
	1225	1.03 e–02	9.38 e–04	9.73 e–03	7.15 e–03	7.43 e–03	1.86 e–03	3.79 e–03
	1600	2.54 e–02	1.22 e–03	2.38 e–03	1.74 e–03	7.12 e–03	1.24 e–03	2.56 e–03

Table 3: Validation RMSE results for estimating the final time PDF $p(x, y, t = 1.0)$ depending on the availability of data. For all trials we use 8 hidden layers with 20 neurons each. Validation RMSE is measured for $N_v = 2500$ data points on a 50×50 uniform grid.

		Forward propagation points						
		800	900	1000	1100	1200	1300	1400
Grid points	400	58 s	67 s	67 s	50 s	52 s	94 s	71 s
	625	37 s	63 s	85 s	79 s	70 s	45 s	104 s
	900	140s	62 s	83 s	82 s	75 s	75 s	79 s
	1225	62 s	86 s	60 s	67 s	88 s	82 s	69 s
	1600	69 s	76 s	66 s	87 s	116 s	86 s	77 s

Table 4: Training time of a neural net for estimating thefinal time PDF $p(x, y, t = 1.0)$ depending on the availability of data. For all trials we used 8 hidden layers with 20 neurons each.

2.2.5 Prediction of the full dynamics of the joint PDF with feed-forward neural nets

Here we employed feed-forward deep neural nets to learn the *whole temporal evolution* of the joint PDF $p(x, y, t)$, within the time interval $t \in [0, 1]$.

Data generation We used another heuristic data generation algorithm based off the one we used for the final time PDF. This process is summarized in four steps below.

1. First we discretized the time interval $t \in [t_0, t_f]$ into N_t number of distinct snapshots. About one third of the snapshots t_k were from a uniform discretization of the interval, including the endpoints. The remaining two thirds were randomly sampled from a half-normal distribution and mapped to the interval so that they would cluster closer to t_f. That is, we constructed a time discretization which was coarser near t_0 and finer near t_f. Having higher-resolution data near t_f was helpful for our test system (23), as the dynamics advected the initially smooth Gaussian into a thin curve, and moreover, the speed of the advection increased with time. For other systems, the distribution of time snapshots may need to be adjusted to improve performance.

2. As before, we randomly sampled a set of N_s initial conditions $\left(x_0^{(i)}, y_0^{(i)}\right) \sim p_0(x, y)$, and propagated those forward to obtain data points at teach time step, $\left(x^{(i,k)}, y^{(i,k)}, t_k\right)$, $i = 1, \ldots, N_s$, $k = 0, \ldots, N_t - 1$. Since the system was divergence free, it was trivial to assign probability data $p^{(i,k)} = p_0\left(x_0^{(i)}, y_0^{(i)}\right)$ to each point, but even with divergence, fitting the solution to the Liouville equation (16) in here would not be difficult.

3. Next we used two rounds of backward integration from each time snapshot t_k. In the first round, we built an $N_m \times N_m$ uniform grid over the forward samples at each individual time snapshot. Backward

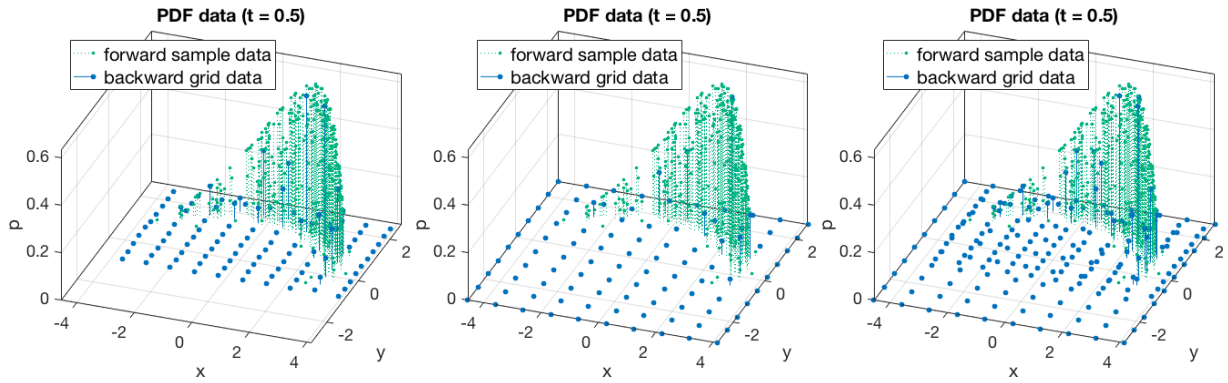


Figure 15: Example of the forward and backward sample strategy for generating data for the time dependent PDF $p(x, y, t)$. Here $t_k = 0.5$, we use $N_s = 100$ forward samples and two $N_m \times N_m = 10 \times 10$ grids of backward samples. Left: forward samples and first round of backward samples. Center: forward samples and second round of backward samples. Right: complete training data at time snapshot $t_k = 0.5$.

numerical integration and the solution to the Liouville equation again supplied probability values. This first grid encouraged the net to learn the boundaries of approximate support of the PDF at each time step.

4. In the second round, for each time snapshot we constructed an additional $N_m \times N_m$ grid over the whole spatial domain visited by the forward samples over *all time* $t \in [t_0, t_f]$, and integrated backward from there. This second grid supplied data further outside the approximate support at each time, which allowed for better interpolation in between time steps as the net saw the whole spatial domain over the whole time interval. This process yields $N_d = N_t \left(N_s + 2N_m^2\right)$ total training data. For a visual example, see Figure 15.

We did not, however, track the flows backward in time and save them at each timestep. The reason we avoided this extension is because backwards integration faces some numerical issues making these data unreliable. Some unlucky data points would be flung far away from the region of interest, and training on these outliers could cause problems. Validation data was taken from a uniform sampling on the space and time domains trained on by the model. This appeared to give a good indication of how well the model performed, as indicated by visually inspecting the plots of $\hat{p}(x, y, t)$ at various times t. Unfortunately, all these decisions are decidedly heuristic and based on the problem at hand (23). Data generation is undoubtedly the main component of this method that can be improved in future work.

Model training In Figure 16, we visualize the neural net reconstruction of training data of the PDF $p(x, y, t)$ at time snapshots $t_k = 0.0$, 0.5, and 1.0. This time, instead of requiring three separate training sessions, the neural net is trained once for the whole time interval $t \in [0.0, 1.0]$, and we plot data only from these points. Tables 5 and 6 include results on the training speed and accuracy of the neural net depending on the net architecture. Tables 7 and 8 include results on the training speed and accuracy of the neural net depending on the availability of training data.

Discussion Tables 5 and 7 show typical $O(10^{-3})$ accuracy over time and space for the time dependent model trained without a PINN. This is not as good as the $O(10^{-4})$ accuracy of the final time model. Even so, the time dependent error takes into account all time instances in the interval $[0.0, 1.0]$, thus the net is able to accurately estimate the PDF at times not represented in the training data. Moreover, we observe that the time dependent model is able to more reliably reconstruct the PDF at later times (i.e. $t \to 1.0$). That

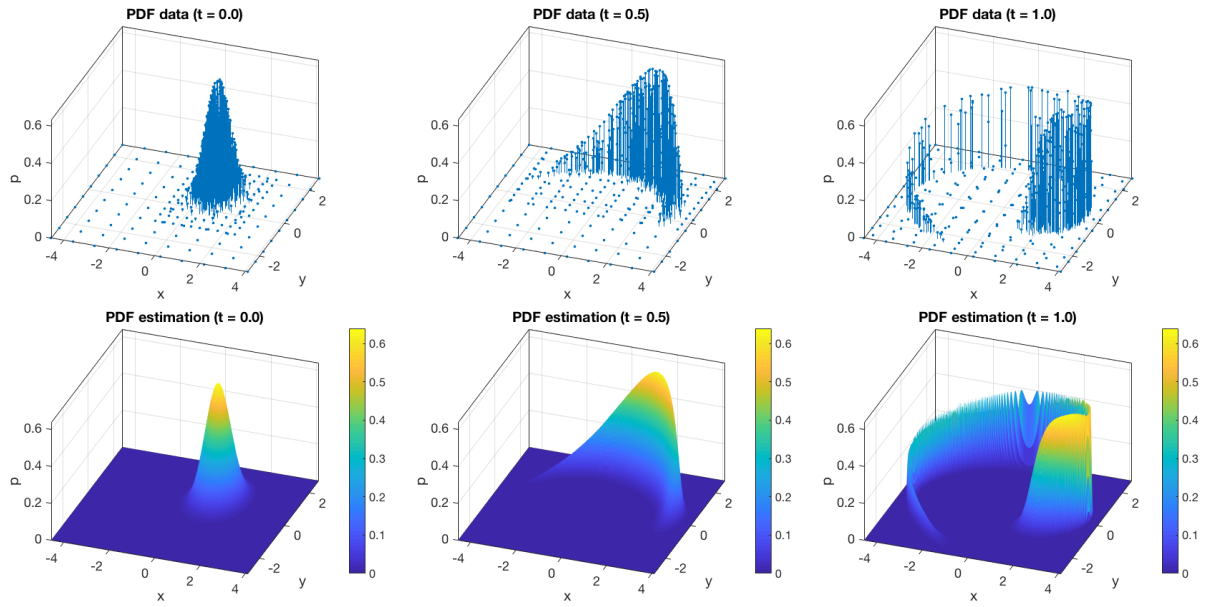


Figure 16: Training data (top) and model reconstruction (bottom) of the PDF $p(x, y, t)$ at time steps $t_k = 0.0$, 0.5, and 1.0, from top to bottom. We used 8 hidden layers with 20 neurons each and fed the net total of 5000 training data points distributed evenly among $N_t = 10$ time instances in $t \in [0.0, 1.0]$. The RMSE was 1.26 e–03 on training data and 1.88 e–03 on 2500 validation data. Each frame of the reconstruction has 90000 points. The rough edge on top of the surface at $t = 1.0$ is due to the coarseness of the plotting data, not the model prediction. We can use a 1000×1000 or finer grid to resolve this more smoothly.

is, often the final time PDF model would have to be re-trained once or twice to get $O(10^{-3})$ RMSE, while we typically obtained this kind of accuracy on the first try with the time dependent model. We suspect that having time as an additional input gave the net additional structure to learn from. The results in Table 5 indicate that accuracy is improved both by increasing the depth and width of the net. Of course, this came at the cost of increased training time, as seen in Table 6. Finally, Table 7 shows that the net can learn the time dependent PDF accurately with as few as around 3000 training data, which is not significantly more than was needed for the learning $p(x, y, t = 1.0)$. Increasing both the number of time snapshots and the data points per snapshot improved accuracy somewhat, up to $O(10^{-3})$. Increasing the number of time snapshots appeared to be more helpful than increasing the number of points in each snapshot.

2.2.6 Prediction of the full dynamics of the joint PDF with physics-informed neural nets (PINN)

Using the same training data generation algorithm as for the time-dependent model learned from data only, we implemented a physics-informed neural net [23] to approximate $p(x, y, t)$ as described previously. To this end, we leveraged TensorFlow's [1] automatic differentiation capabilities to compute the partial derivatives $\partial\hat{p}/\partial t$, $\partial\hat{p}/\partial x$, and $\partial\hat{p}/\partial y$ and added an MSE penalty to neural net (see equation (21)). This additional penalty was enforced on both the training data points and a set of collocation points randomly generated by Latin hypercube sampling from the spatial domain and time interval trained. In Table 9 we summarize accuracy results for the PINN depending on the net architecture. With the PINN, we can now reliably obtain $O(10^{-4})$ validation error with only half the number of training data we used before. These nets benefit from having at least 20 or 30 neurons per layer, and the performance tends to improve if we make the net deeper. Unfortunately, these nets also take around twice as long to train as the standard neural net (compare Table

		Neurons per layer			
		10	20	30	40
Hidden layers	2	7.81 e–02	3.84 e–02	4.00 e–02	7.48 e–02
	4	1.66 e–02	1.54 e–02	2.28 e–02	5.05 e–03
	6	1.66 e–02	2.07 e–03	3.04 e–03	3.01 e–03
	8	4.14 e–02	1.10 e–03	9.64 e–03	3.57 e–03
	10	7.28 e–03	3.13 e–03	2.40 e–03	5.48 e–03
	12	3.22 e–03	7.86 e–03	4.74 e–03	4.76 e–03

Table 5: Validation RMSE results for estimating the time dependent PDF $p(x, y, t)$ over the interval $t \in [0.0, 1.0]$ depending on the structure of the neural net. The total amount of training data N_d is fixed at 5000 samples spread over $N_t = 10$ time snapshots. Validation RMSE is measured for $N_v = 2500$ data points. Note that because training data and validation data are randomly generated, weights are randomly initialized, and the optimization problem is not convex, performance can vary each time the model is trained.

		Neurons per layer			
		10	20	30	40
Hidden layers	2	50 s	127 s	153 s	180 s
	4	74 s	146 s	182 s	246 s
	6	101 s	159 s	194 s	258 s
	8	187 s	143 s	214 s	261 s
	10	111 s	213 s	222 s	324 s
	12	164 s	219 s	357 s	400 s

Table 6: Training time of a neural net for estimating the time dependent PDF $p(x, y, t)$ over the interval $t \in [0.0, 1.0]$, depending on the structure of the net. Training data generation ($N_d = 5000$ points) and validation data generation ($N_v = 2500$ points) by numerical integration take approximately 5 and 2 seconds each, while evaluating the trained net for 1.8 million outputs takes only about 2 seconds.

10 with Table 6). With enough collocation points, we can also compensate for having only a few training data points, as seen in Table 11. These collocation points can be generated in a small fraction of a second, but adding more collocation points does increase the training time significantly (see Table 12), so overall there are no time savings with this method. The improved accuracy gained by using a PINN may be worth the increased training time, depending on the problem. This will be important moving forward, as some systems may be more poorly behaved than (23), especially once we address systems with divergence and control.

Discussion The physics-informed neural net looks to be superior to the standard deep neural net for the time-dependent problem, and has advantages over the neural net for estimating the final time PDF. These are, specifically, robustness to poor data, and, of course, the availability of $\hat{p}(x, y, t)$ for all $t \in [t_0, t_f]$. This could eventually open up the possibility of using control to steer the PDF around obstacles. The disadvantage here was increased training time (prediction was still order of magnitudes faster than numerical integration). We expect that deploying these programs on a GPU will close this gap. Before we can successfully use this method for control, we will need to address three challenges.

		Points per snapshot					
		200	300	400	500	600	700
Snapshots	6	7.30 e–02	6.08 e–02	6.35 e–02	2.52 e–02	1.77 e–02	8.81 e–03
	8	3.06 e–02	7.34 e–03	2.06 e–03	3.52 e–03	1.21 e–02	2.63 e–02
	10	7.82 e–03	2.90 e–03	2.51 e–03	1.10 e–03	2.78 e–03	3.02 e–03
	12	2.00 e–02	2.97 e–03	4.96 e–03	7.71 e–03	6.53 e–03	3.67 e–03
	14	4.19 e–03	2.73 e–03	2.61 e–03	5.71 e–03	2.39 e–03	1.42 e–03

Table 7: Validation RMSE results for estimating the time dependent PDF $p(x, y, t)$ over the interval $t \in [0.0, 1.0]$ depending on the availability of data. For all trials we use 8 hidden layers with 20 neurons each. Validation RMSE is measured for $N_v = 2500$ data points.

		Points per snapshot					
		200	300	400	500	600	700
Snapshots	6	106 s	101 s	84 s	141 s	139 s	125 s
	8	69 s	101 s	117 s	124 s	146 s	117 s
	10	86 s	108 s	133 s	143 s	159 s	131 s
	12	138 s	138 s	146 s	154 s	210 s	211 s
	14	109 s	128 s	193 s	310 s	256 s	248 s

Table 8: Training time of a neural net for estimating the time dependent PDF $p(x, y, t)$ over the interval $t \in [0.0, 1.0]$, depending on the availability of data. For all trials we used 8 hidden layers with 20 neurons each.

1. The current heuristic data generation scheme can be improved. Time snapshots, for example, could be chosen adaptively to reflect the speed of the time evolution of the system (more snapshots where the evolution is faster). Moreover, we need not generate only a fixed number of time snapshots and grids at each snapshot – data can be distributed throughout the whole space-time domain. Also, we may be able to improve the first round of backward sampling by choosing points in a concentrated way around the forward samples. Specifically, we could build a convex hull around the forward samples and do backward sampling within that hull, assigning more samples near those points identified to be close to the boundary of the hull. This could provide both better resolution data within the approximate support of $p(\boldsymbol{x}, t)$, as well as a better resolution of the boundary. A method for constructing the hull and identifying points close to the boundary was proposed in [28].

2. Aside from the PINN, the neural net architecture we used here was very bare-bones. Many sophisticated improvements to this basic architecture have been developed, which we can test. For example, we can regularize the parameters $\boldsymbol{\theta}$ of the net by adding an $\ell_1(\boldsymbol{\theta})$ or $\ell_2(\boldsymbol{\theta})$ penalty term to the training cost function (21). This will reduce the likelihood of having large parameters which can cause unexpected behavior.

3. Lastly, we point out that PINNs are still rather underdeveloped methods. Here we used a direct application of the method described in [23], which imposes the governing PDE on the model with a single penalty term (22). Since the problem is highly non-convex, minimizing this term once is in no way guaranteed to yield a satisfying solution. To address this, we will develop a way to enforce the Liouville equation (15) as a *constraint*, which could further improve the efficacy of the PINN.

    - Explicit constraints are difficult to implement in neural nets. To get around this, one possibility is to manually encode a sort of penalty method by training in epochs. That is, we retrain the

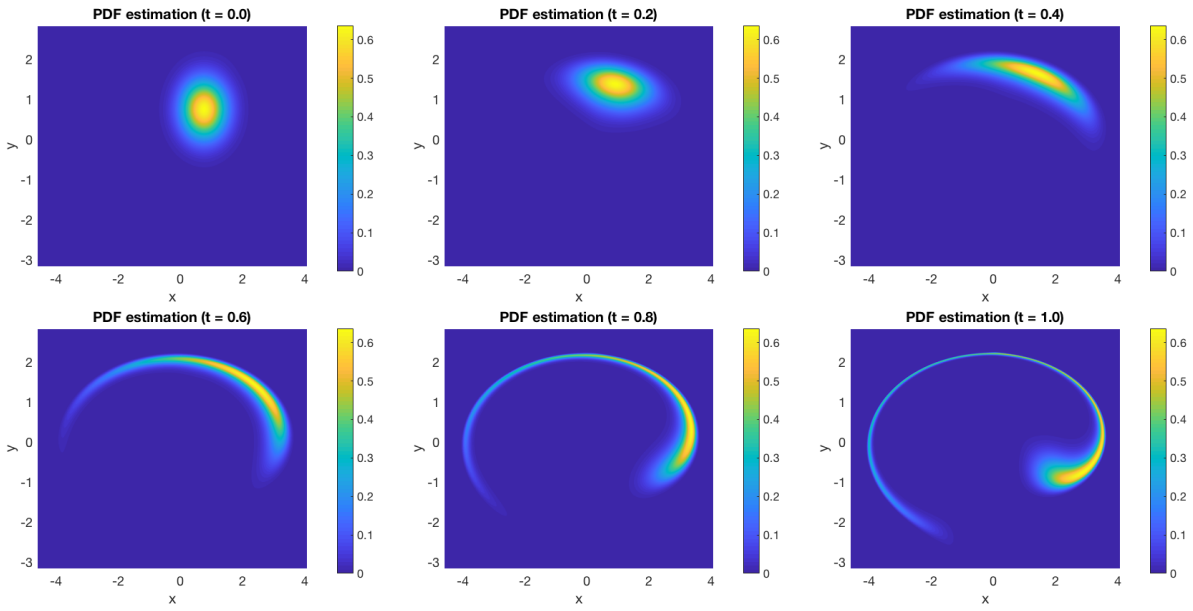


Figure 17: Time evolution of $p(x, y, t)$ for the system (23) where the initial PDF is the product of two indepent Gaussians with means $\mu_x = \mu_y = 0.75$ and variances $\sigma_x^2 = \sigma_y^2 = 0.25$. Predictions are made using a physics-informed neural net with 8 hidden layers, each with 20 neurons, trained on $N_d = 2000$ data points and $N_c = 8000$ total collocation points. The final RMSE was 6.98×10^{-4} with respect to $N_v = 2500$ validation points.

net multiple times, each time increasing the weight of the second term (22) in the training cost function (21). If the optimization converges, then intuitively we expect that in the limit, the Liouville equation will be enforced as a constraint (we will need to prove this conjecture). We can increase the weight of the constraint term either by simply scaling it, or by increasing the number of collocation points fed to the net. The second option (or a combination of the two) may be preferable, since using fewer points at the start will reduce computation time.

2.2.7 Refining physics-informed deep neural networks

This physics-informed neural network algorithms we described in section 2.2.1 do not consider the relative magnitudes of the two error terms, nor does it seek to satisfy the governing Liouville equation (15) as a constraint. In particular, since it only imposes a penalty, we often see that the training optimization gets caught in a local minimum, which yields unsatisfying solutions. To overcome this difficulty, we tested two refinements (and their combination) of the way $MSE_{\mathcal{L}}$ in included in the optimization process.

- The first method consists in updating the weight of the penalty term $MSE_{\mathcal{L}}$, i.e., the parameter μ in (21), as the neural network training progresses.
- The second method consists in randomizing which of the collocation points were used, and re-sampling points as the neural network training progresses.

For both methods, we opted for a simple implementation and considered a defined number of iterations (or "epochs") in which we increase the penalty weight and/or re-sample the collocation points.

		Neurons per layer			
		10	20	30	40
Hidden layers	2	1.19 e–01	1.48 e–02	1.01 e–02	3.49 e–03
	4	2.02 e–02	2.79 e–03	5.22 e–04	9.01 e–04
	6	1.51 e–02	5.85 e–04	6.22 e–04	4.92 e–04
	8	2.32 e–03	5.13 e–04	8.57 e–04	4.51 e–04
	10	3.25 e–03	8.19 e–04	7.55 e–04	4.48 e–04
	12	1.57 e–03	1.36 e–03	5.90 e–04	4.91 e–04

Table 9: Validation RMSE results for estimating the time dependent PDF $p(x, y, t)$ over the interval $t \in [0.0, 1.0]$ with a physics-informed neural net, depending on the structure of the net. The amount of training data N_d is fixed at 2500 samples and the number of collocation points N_c is fixed to 5000 (including the training data points). Validation RMSE is measured for $N_v = 2500$ data points. Note that because training data, collocation points, and validation data are randomly generated, weights are randomly initialized, and the optimization problem is not convex, performance can vary each time the model is trained.

		Neurons per layer			
		10	20	30	40
Hidden layers	2	53 s	188 s	419 s	403 s
	4	163 s	232 s	439 s	593 s
	6	64 s	368 s	538 s	676 s
	8	286 s	385 s	618 s	742 s
	10	270 s	578 s	635 s	907 s
	12	294 s	598 s	670 s	1565 s

Table 10: Training time of a physics-informed neural net for estimating the time dependent PDF $p(x, y, t)$ over the interval $t \in [0.0, 1.0]$, depending on the structure of the net. Training data generation ($N_d = 2500$ points) and validation data generation ($N_v = 2500$ points) by numerical integration each take approximately 2 seconds, while evaluating the trained net for 1.8 million outputs also takes only 2 seconds.

Increasing the penalty weight Increasing the penalty weight μ in (21) on-the-fly during the optimization progresses turned out to not be much of an improvement at all. At the beginning we thought that increasing the relative magnitude of the penalty term by scaling it in each iteration/epoch would lead to better solutions of the governing PDE (15). However, the sequences of weights we tried typically made the solution *worse* over time [20]. We ran 10 simulations, each with two different sequences of penalty weights, i.e.,

$$\{\mu_E\} = \left\{\sqrt{i_{E-1} + E}\right\} \qquad \text{and} \qquad \{\mu_E\} = \{i_{E-1} + E\}, \tag{24}$$

where $E = 1, 2, \ldots, 10$ is the epoch number, and i_{E-1} is total number of training iterations over all the previous epochs. In Figure 18 we plot the results we obtained. In each iteration/epoch we used the full set of $N_c = 10000$ collocation points and $N_d = 2500$ data points, and trained a neural net with 8 hidden layers of 20 neurons each.

We found that the mean error for the unrefined PINN was 1.98 e–02, with a standard deviation of 5.66 e–02. Mean training time was 562 s, with a standard deviation of 248 s. For $\{\mu_E\} = \left\{\sqrt{i_{E-1} + E}\right\}$ the mean error was 3.38 e–03, with a standard deviation of 4.00 e–03. Mean training time was 794 s, with a standard deviation of 141 s. Finally for $\{\mu_E\} = \{i_{E-1} + E\}$ the mean error was 4.88 e–02 with a standard deviation of 3.66 e–02. Mean training time was 812 s, with a standard deviation of 211 s. Of course, the unrefined PINN is quicker to train since it requires only one iteration/epoch. Most of the time, it also

		Collocation points						
		4000	5000	6000	7000	8000	9000	10000
Training data	500	1.72 e–03	1.38 e–02	1.40 e–02	3.83 e–02	2.25 e–03	1.69 e–03	1.41 e–02
	1000	1.46 e–03	6.04 e–03	2.23 e–03	1.82 e–03	1.29 e–03	9.74 e–04	1.78 e–03
	1500	8.33 e–03	3.20 e–03	1.99 e–03	1.40 e–03	7.91 e–04	8.39 e–04	1.19 e–02
	2000	6.81 e–03	4.30 e–03	7.62 e–04	5.97 e–04	6.33 e–04	8.51 e–04	6.21 e–04
	2500	1.06 e–03	5.13 e–04	6.42 e–04	8.05 e–04	8.62 e–04	8.56 e–04	9.88 e–04
	3000	1.03 e–03	7.22 e–04	1.04 e–03	6.89 e–04	8.46 e–04	5.87 e–04	1.02 e–03

Table 11: Validation RMSE results for estimating the time dependent PDF $p(x, y, t)$ over the interval $t \in [0.0, 1.0]$ with a physics-informed neural net, depending on the total number of training data N_d and number of collocation points N_c, including the training data points. For all trials we used 8 hidden layers with 20 neurons each. Validation RMSE is measured for $N_v = 2500$ data points.

		Collocation points						
		4000	5000	6000	7000	8000	9000	10000
Training data	500	197 s	422 s	403 s	258 s	487 s	494 s	836 s
	1000	240 s	320 s	353 s	269 s	484 s	404 s	489 s
	1500	335 s	352 s	351 s	387 s	360 s	558 s	425 s
	2000	362 s	428 s	404 s	495 s	563 s	515 s	517 s
	2500	360 s	385 s	380 s	371 s	318 s	602 s	640 s
	3000	287 s	319 s	321 s	390 s	535 s	706 s	529 s

Table 12: Training time of a neural net for estimating the time dependent PDF $p(x, y, t)$ over the interval $t \in [0.0, 1.0]$ using a physics-informed neural net, depending on the total number of training data N_d and number of collocation points N_c, including the training data points. For all trials we used 8 hidden layers with 20 neurons each.

achieves better accuracy: when we ignore the two outliers at the top left of Figure 18, the mean error is 1.52 e–03 with standard deviation 1.03 e–03, mean training time is 623 seconds with standard deviation 171 s. On the other hand, while giving the penalty term an increasing weight (in particular $\{\mu_E\} = \{\sqrt{i_{E-1} + E}\}$) tends to make the final error worse on most runs, it might also make the optimization more robust to poor initializations. Thus, it may be possible to tune the sequence $\{\mu_E\}$ to improve the training robustness for particular problems. Even if the weight sequence is constant over all epochs, it will be necessary to adjust the magnitude relative to the data loss term so as to improve training.

Randomizing collocation points Randomly choosing a subset of the collocation points to train on in each iteration/epoch served us to both

- Decrease training time;
- Regularize the training against bad initializations.

Specifically, we defined suitable sequences $\{N_{c,E}\} = N_{c,1}, N_{c,2}, \ldots, N_c$, where N_c is the total number of pre-generated collocation points, while $N_{c,E}$ is the number of points used in iteration/epoch E. By training on fewer collocation points at the beginning, we obtained an *approximate solution in far less time than training on the full set*. Such solution can be subsequently refined and regularized by re-sampling

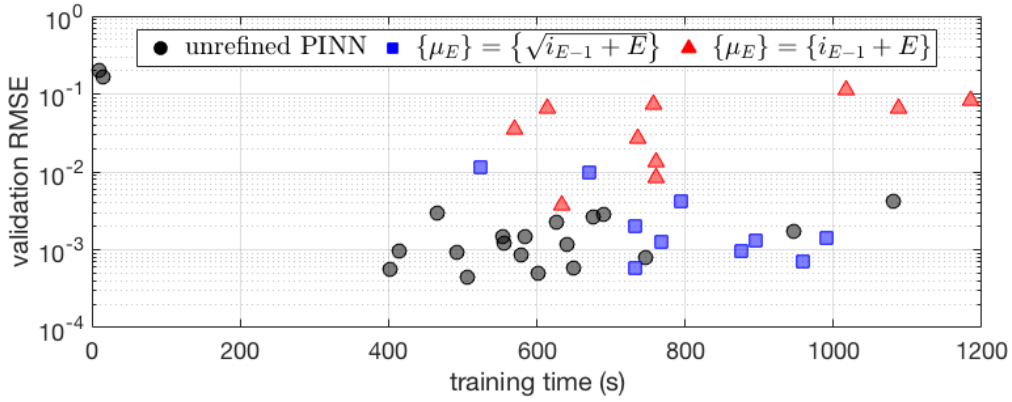


Figure 18: Performance of physics-informed neural nets with cost function (21) and fixed $\mu = 1$ (unrefined PINN) versus neural nets where we increase the penalty parameter $\{\mu_E\}$ on-the-fly as optimization proceeds. Here used $N_d = 2500$ training data, $N_c = 10000$ collocation points, $E = 1, \ldots, 10$ iteration/epochs, and 8 hidden layers with 20 neurons each.

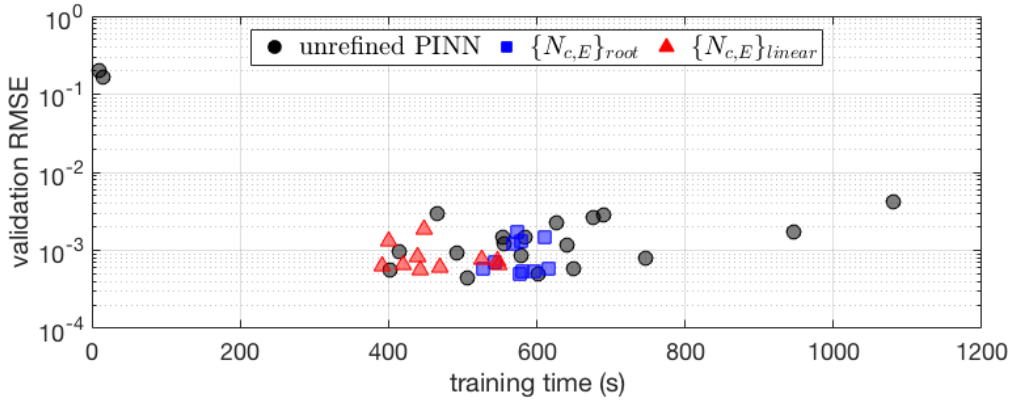


Figure 19: Performance of physics-informed neural nets with cost function (21) and fixed $\mu = 1$ (unrefined PINN) versus neural nets where we randomly select collocation points $\{N_{c,E}\}$ according to the sequences in (25)-(26). Here we use $N_d = 2500$ training data, $N_c = 10000$ total collocation points, $E_{\max} = 10$ epochs, and 8 hidden layers with 20 neurons each.

collocation points used in each iteration/epoch. We tested two sequences,

$$\{N_{c,E}\}_{\text{root}} = \left\{ N_c \sqrt{\frac{E/E_{\max} + 1}{2}} \right\}, \tag{25}$$

$$\{N_{c,E}\}_{\text{linear}} = \left\{ N_c \frac{E/E_{\max} + 1}{2} \right\}. \tag{26}$$

With $N_c = 10000$ collocation points, the first epoch was trained with $N_{c,1} = 7416$ (root sequence) and $N_{c,1} = 5500$ (linear sequence), respectively. These were completely re-sampled in each epoch, meaning that some points used in previous epochs were not necessarily included in following epochs. Overall we obtained a similar maximum accuracy (that is, the best accuracy obtained over all trial runs), while driving *the mean and variance of the error down significantly*. In addition, training was reliably faster than for the unrefined PINN. In Figure 19 we summarize the performance of the randomized algorithm. Specifically, we used 10 trials for each sequence (25)-(26), and compared the results to the unrefined PINN results used before. We

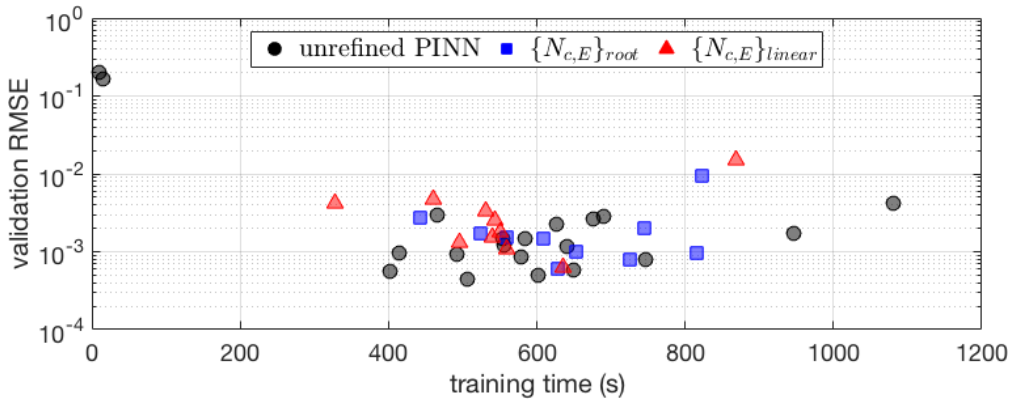


Figure 20: Performance of physics-informed neural nets with cost function (21) and fixed $\mu = 1$ (unrefined PINN) versus neural nets where we randomly select collocation points $\{N_{c,E}\}$ in each iteration/epoch, along with increasing scalar penalty weights $\{\mu_E\} = \{\sqrt{i_{E-1} + E}\}$. Here we use $N_d = 2500$ training data, $N_c = 10000$ total collocation points, $E_{\max} = 10$ epochs, and 8 hidden layers with 20 neurons each.

notice that for the root sequence (25), the mean error is 9.15 e–04, with a standard deviation of 4.59 e–04. The mean training time is 577 s, standard deviation was 27 s. For the linear sequence (26), the mean error is 8.64 e–04 with a standard deviation of 4.17 e–04. The mean training time is 463 s with a standard deviation of 58 s. Randomizing collocation points over iterations/epochs made the optimization of the cost function (21) significantly more robust, and with the sequence (26) also significantly quicker. Since this appears to already provide good regularization, we do not attempt to make ℓ_1 or ℓ_2 weight regularization work: a quick prototype already indicated that these added computational burden without making the training more reliable.

Combining randomized collocation points with evolving penalty weights To reduce the validation mean squared error further, we tried to combine randomization of collocation points with evolving penalty terms. This approach yields some improvements in training robustness and speed, but not as much as using the randomized collocation points with constant penalty weights. Specifically, we perfomed the same numerical experiment with both series of collocation point numbers, combined with the first set of weights $\{\mu_E\} = \{\sqrt{i_{E-1} + E}\}$. Figure 20 shows the results. Randomizing the collocation points is the clear winner here, though scalar penalty weights may need to be added and tuned depending on the problem at hand. Note that if we use the randomized root sequence (25) with $\{\mu_E\} = \{\sqrt{i_{E-1} + E}\}$, the mean error is 2.22 e–03, with a standard deviation of 2.57 e–03. This mean training time is 653 s, with a standard deviation of 125 s. On the other hand, if we use the linear sequence (26) with $\{\mu_E\} = \{\sqrt{i_{E-1} + E}\}$, then we obtain a mean error of 3.58 e–03 with a standard deviation of 4.18 e–03. In this case the mean training time is 551 s and the standard deviation is 137 s.

Training physics-informed neural nets with log probability data and different cost functions Previously, we made the decision to train on log probability data so that the net would predict only positive probability values, since there is no other way to neatly enforce this constraint. The downside is, however, that when $p \approx 0$, $|\log p| >> 1$. Consequently in log scale, the difference between small probability values is magnified, and the difference between large probability values is made insignificant in comparison. It follows that minimizing a mean square error term on log probabilities, as we do in (18), encourages the model

to accurately learn minute differences where p is small, but ignore relatively large differences between large probability values. Thus it makes sense to consider different error metrics for the training data. We consider and compare three different error metrics: the MSE on the true probability, a weighted MSE on the log probability, and the symmetric Kullback-Leibler divergence.

- **Mean squared error.** This is possibly the most intuitive error metric we can use. Instead of (18), we calculate

$$MSE_{\text{data, true}}(\boldsymbol{\theta}) \coloneqq \frac{1}{N_d} \sum_{i=1}^{N_d} \left[\exp \log \hat{p} \left(\boldsymbol{x}^{(i)}, t \right) - p \left(\boldsymbol{x}^{(i)}, t \right) \right]^2. \tag{27}$$

  This immediately provides an error term which scales with the magnitude of the probability.

- **Weighted mean squared error.** Here we simply weight each term of the MSE on log probability by the true probability value. Since this true probability is already available (it is pre-calculated), this only requires N_d additional multiplications each time the loss is computed.

$$MSE_{\text{data, weighted}}(\boldsymbol{\theta}) \coloneqq \frac{1}{N_d} \sum_{i=1}^{N_d} p \left(\boldsymbol{x}^{(i)}, t \right) \left[\log \hat{p} \left(\boldsymbol{x}^{(i)}, t \right) - \log p \left(\boldsymbol{x}^{(i)}, t \right) \right]^2. \tag{28}$$

- **Symmetric Kullback-Leibler divergence.** The last error metric we test is the Kullback-Leibler divergence, a well-known metric for comparing two probability distributions. So that we do not favor large $\hat{p}$, we use the symmetric version as originally proposed in [16]:

$$\begin{aligned} \mathrm{sD}_{\mathrm{KL}}(p \| \hat{p}(\boldsymbol{\theta})) &= \mathrm{D}_{\mathrm{KL}}(p \| \hat{p}(\boldsymbol{\theta})) + \mathrm{D}_{\mathrm{KL}}(\hat{p}(\boldsymbol{\theta}) \| p), \\ &= \int p(\boldsymbol{x}) \left[\log p(\boldsymbol{x}) - \log \hat{p}(\boldsymbol{x}) \right] d\boldsymbol{x} + \int \hat{p}(\boldsymbol{x}) \left[\log \hat{p}(\boldsymbol{x}) - \log p(\boldsymbol{x}) \right] d\boldsymbol{x}. \end{aligned} \tag{29}$$

  Next we assume that the training data is a representative sample from the training domain and approximate the integrals by Monte Carlo integration. We let V be the volume of the training domain (including time) so that

$$\begin{aligned} \mathrm{sD}_{\mathrm{KL}}(p \| \hat{p}(\boldsymbol{\theta})) &\approx V \frac{1}{N_d} \sum_{i=1}^{N_d} p \left(\boldsymbol{x}^{(i)}, t \right) \left[\log p \left(\boldsymbol{x}^{(i)}, t \right) - \log \hat{p} \left(\boldsymbol{x}^{(i)}, t \right) \right] \\ &\quad + V \frac{1}{N_d} \sum_{i=1}^{N_d} \exp \left(\log \hat{p} \left(\boldsymbol{x}^{(i)}, t \right) \right) \left[\log \hat{p} \left(\boldsymbol{x}^{(i)}, t \right) - \log p \left(\boldsymbol{x}^{(i)}, t \right) \right] \\ &= \frac{V}{N_d} \sum_{i=1}^{N_d} \left[p \left(\boldsymbol{x}^{(i)}, t \right) - \exp \left(\log \hat{p} \left(\boldsymbol{x}^{(i)}, t \right) \right) \right] \left[\log p \left(\boldsymbol{x}^{(i)}, t \right) - \log \hat{p} \left(\boldsymbol{x}^{(i)}, t \right) \right]. \end{aligned} \tag{30}$$

  In this formulation we need to take N_d exponents, N_d additional subtractions, and one additional multiplication, making each cost evaluation about as expensive as the MSE on the true probability (27).

Hereafter, we train a physics-informed neural network 10 different times with each different training data error metric described above. This gives us an indication of how well they perform. We train for only on one iteration/epoch using a constant weight of $\mu_1 = 1$ on the Liouville term, and the same number of data points and architecture as before. In other words, we examine the effects of the metric we use in the cost function (21) for the so-called "unrefined PINN". For validation, we still use the root mean square error (RMSE) with

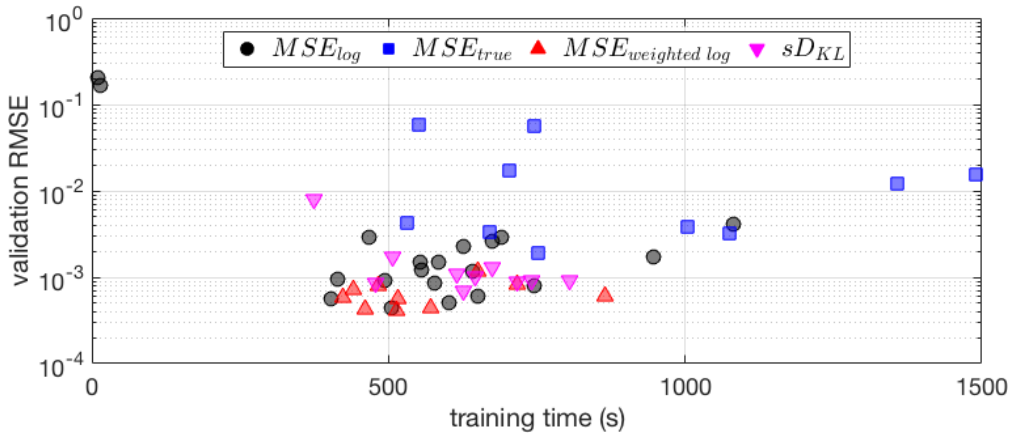


Figure 21: Performance of physics-informed neural nets with cost function (21) and fixed $\mu = 1$ trained with different training data loss metrics. We train for one epoch with $N_d = 2500$ training data, $N_c = 10000$ total collocation points, and 8 hidden layers with 20 neurons each.

respect to the *benchmark probability*. This is so that we can compare results of these trials to prior results, and because this appears to be a good metric for validation (though not necessarily training). Figure 21 summarizes all results we obtained. We see that if we use the cost (27), we obtain a mean validation RMSE (on benchmark probability) of 1.76 e–02 with standard deviation 2.17 e–02. The mean training time is 888 s while the standard deviation is 332 s. On the other hand, with the cost functional (28) the mean validation error is 6.49 e–04 with standard deviation was 2.34 e–04. The mean training time is 564 s with standard deviation 141 s. Lastly with the symmetric Kullback-Leibler divergence (30) the mean error is 1.73 e–03 with standard deviation was 2.17 e–03, while the mean training time is 619 s with a standard deviation of 131 s. The results of Figure 21 suggest that training on the benchmark probability data (27) takes longer and is significantly less accurate than training on the log probability data (18). This is somewhat surprising because minimizing this error metric is equivalent to minimizing the RMSE, from which we evaluate our trained models. Since evaluating the cost metric alone is not significantly more expensive than for a the MSE on log probability, the long training times must be caused by the difficulty in training on this metric. On the other hand, both the weighted MSE (28) and the symmetric Kullback-Leibler divergence (30) compare well to MSE_{data} (18). The symmetric Kullback-Leibler divergence performs about as well MSE_{data} *without* the outliers. This implies that it is somewhat more robust. The weighted MSE is the outstanding performer here: it is reliably more accurate than all the other error metrics, without taking longer to train.

2.2.8 Application to the Van der Pol oscillator

So far we tested our algorithms to the divergence-free system (23). In this section we study *non*-divergence-free systems. In particular, we consider the Van der Pol oscillator as a model test problem. The governing equations are

$$\begin{cases} \dot{x} = y, \\ \dot{y} = \mu \left(1 - x^2\right) y - x. \end{cases} \tag{31}$$

Since trajectories for this system are bounded (they are drawn to the limit cycle), data generation is less complicated than for possibly unbounded systems. Typically we set the damping parameter to $\mu = 1.0$. The phase portrait that corresponds to this value of μ is shown in Figure 22. While the trajectories themselves are integrated quite easily, the density can become quite large at some points on the limit cycle. This means that

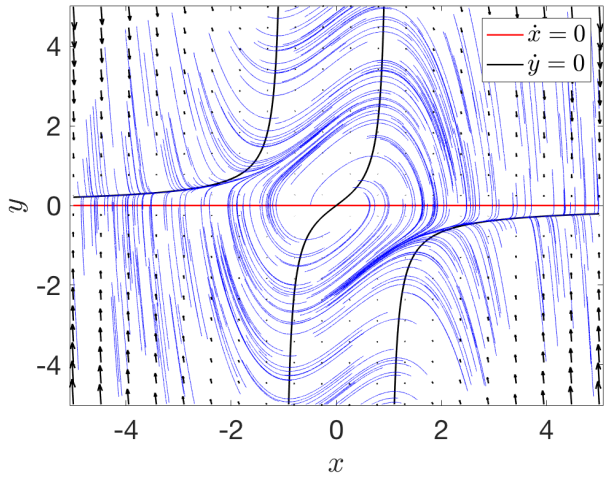


Figure 22: Phase portrait of the Van der Pol oscillator (31) for $\mu = 1$.

depending on where we place the initial distribution, the density can reach numerical infinity in finite time. Thus the success of the PDF estimation is highly dependent on the particular choices of p_0 and t_f. Working with PDFs with densities on different scales, in particular if the scales change over time, will be a central challenge when we implement this method with control. This is because, in general, the ideal control will push the PDF into a tall, thin spike around some target point. That is, in general we would like the variance to be small, which means the density over the desired expectation will be quite large. It is possible that learning the flow map and inverse flow map, then using equation (16), will alleviate this problem.

Data generation We still use the existing data generation algorithm, largely unchanged. The only difference is that we temporarily save the solutions at all time values output by the adaptive time stepping in SciPy's`sdop853` ODE integrator [12, 13]. We then used simple trapezoidal integration over the resulting time series to approximate (16), giving us $p(\boldsymbol{x}, t)$ at all the time steps $t = t_k$. This additional data was *not* saved for training, though we could have done so. Saving all trajectory data might have slightly improved accuracy, at the cost of more training time. The only reason to do this was to get better approximations of $p(\boldsymbol{x}, t = t_k)$.

Normalized validation error To facilitate accuracy comparison between different problems, we normalize the validation RMSE in the following way

$$NRMSE_{\text{val}} = \frac{\sqrt{\dfrac{1}{N_v}\displaystyle\sum_{i=1}^{N_v}\left[p\left(\boldsymbol{x}^{(i)}\right) - \hat{p}\left(\boldsymbol{x}^{(i)}\right)\right]^2}}{\max\limits_i p\left(\boldsymbol{x}^{(i)}\right) - \min\limits_i p\left(\boldsymbol{x}^{(i)}\right)}. \tag{32}$$

For example, one trial on the two-dimensional divergence-free test system (23) with the usual problem parameters and hyperparameters yielded a validation RMSE of 5.88 e–04, which translated to a normalized RMSE of 9.26 e–04. For this system and initial distribution we had a normalization factor of $\max p_{\text{val}} - \min p_{\text{val}} \approx 0.63$, so all previous results can be similarly scaled for comparison.

t_f	training time	RMSE	NRMSE
1	286 s	5.66 e–04	9.05 e–04
2	303 s	1.08 e–03	1.76 e–03
3	481 s	1.94 e–03	3.48 e–03
4	699 s	3.71 e–03	5.96 e–03

Table 13: Van der Pol oscillator (31) with initial PDF centered at the origin. Training time of the physics-informed neural network with randomized samples and increasing penalty weights.

Numerical results

In this section we study the random initial value problem for the Van der Pol equation (31). We consider three different cases

- Initial PDF centered at the origin of the phase plane (see Figure 22);
- Initial PDF centered inside the limit cycle;
- Initial PDF centered outside the limit cycle.

These cases yield similar dynamics but different time-evolving PDFs. Hereafter, we study each case in detail.

Initial PDF centered at the origin First we let $p_0(x, y)$ be the product of two independent Gaussians centered at $(\mu_x, \mu_y) = (0, 0)$, with standard deviations $(\sigma_x, \sigma_y) = (0.5, 0.5)$. We approximate the time-dependent solution with final times $t_f = 1, 2, 3,$ and 4 using different neural networks. Each model had 8 hidden layers of 20 neurons each and was trained using weighted log probabilities (28) for the training cost and randomized collocation points according to the formula (26) with $E_{\text{max}} = 10$ iteration/epochs. We chose a simple penalty weight sequence $\{\mu_E\} = \{E\}$. We train the network on $N_d = 2500$ training data split among $N_t = 10$ time snapshots ($N_s = 200$ forward samples), along with $N_c = 10000$ collocation points. Finally, each model was validated against $N_v = 2500$ data points with $t \in [0, t_f]$. In Table 13 we summarize the training times for the final PDF at different times. In Figures 23 and 24 we plot the PDF we obtain with two neural nets over the interval $t \in [0.0, 4.0]$. The first net is trained only with data for $t \in [0.0, 3.0]$, so *the two last frames are extrapolated.* In both sets of predictions, we see that the initial PDF spreads out and is attracted to the familiarly shaped limit cycle. In addition, most of the probability mass tends to accumulate at opposite corners of the limit cycle. The extrapolation for the first net is still fairly accurate at $t = 3.5$, which we confirm by visually comparing it to the second net's prediction, and estimating the total probability mass at that time to be 1.0449.

Initial PDF centered inside the limit cycle (not at the origin) Now we let $p_0(x, y)$ be the product of two independent Gaussians centered at $(\mu_x, \mu_y) = (0.5, 0)$ with standard deviations $(\sigma_x, \sigma_y) = (0.5, 0.5)$. This places almost all of the initial probability mass inside the limit cycle, slightly off center. We approximate the time-dependent solution with final times $t_f = 1, 2,$ and 3 using different networks. The hyper-parameters were chosen as before, except for the Liouville penalty weight which we fixed at $\{\mu_E\} \equiv 1$. In Figures 25 and 26 we plot the outputs of two neural nets over the interval $t \in [0.0, 3.0]$. The first net is trained only with data for $t \in [0.0, 2.0]$, so *the final three frames are extrapolated.* With just this small change in the initial PDF, the density grows much faster on the limit cycle, making the problem significantly harder than

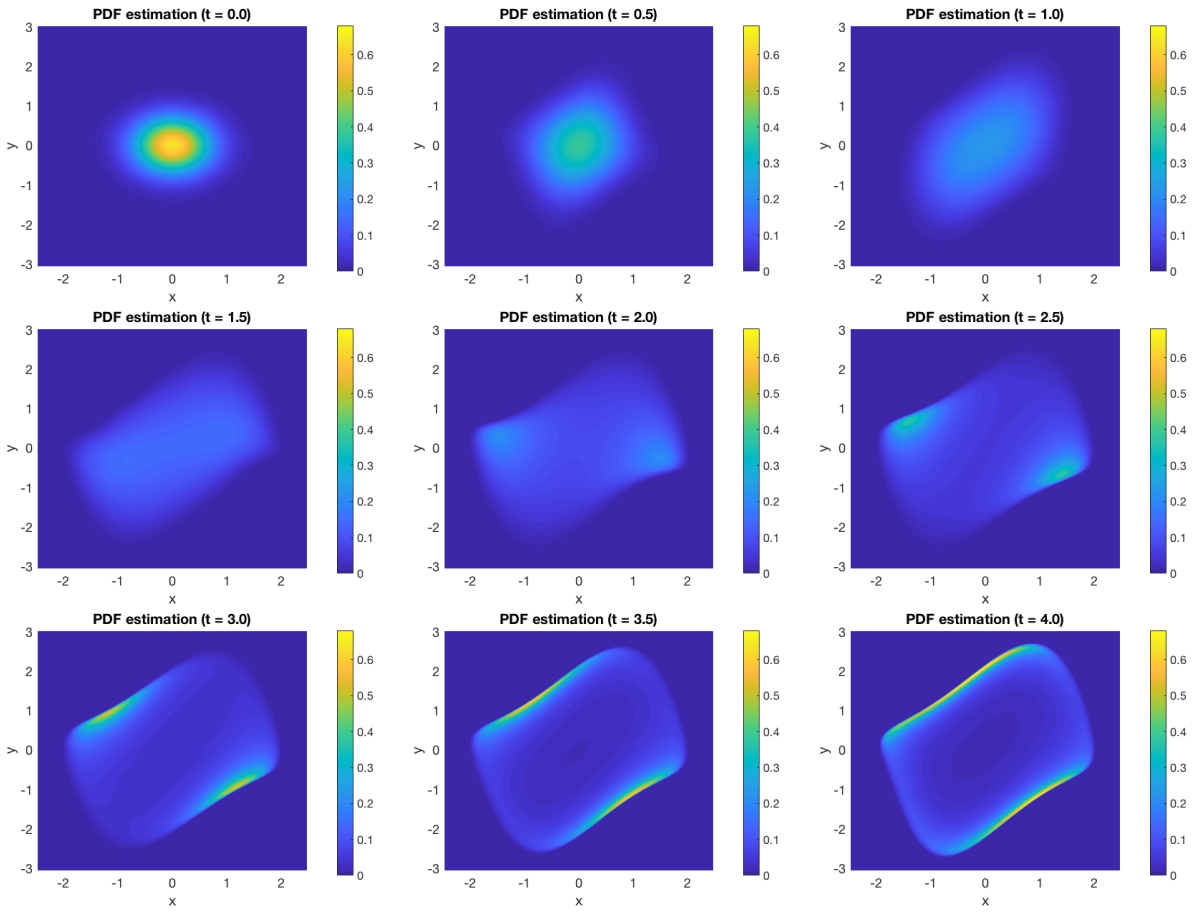


Figure 23: Time evolution of $p(x, y, t)$ for the Van der Pol oscillator (31) where the initial PDF is the product of two indepent Gaussians with means $\mu_x = \mu_y = 0$ and variances $\sigma_x^2 = \sigma_y^2 = 0.25$. Here the net is trained with data in the interval $t_0 = 0.0$ to $t_f = 3.0$, so the final two frames are extrapolated.

that with the initial PDF at the origin. The extrapolation with the first net is surprisingly accurate even at $t = 2.6$, but starts to break down after this point.

Initial PDF centered outside the limit cycle Lastly, we let $p_0(x, y)$ be the product of two independent Gaussians centered at $(\mu_x, \mu_y) = (0, 3)$ with standard deviations $(\sigma_x, \sigma_y) = (0.5, 0.5)$. This places the initial PDF outside of the limit cycle. This problem is harder since the density grows much faster than the other problems, so we train only up to $t_f = 1$. Here we use $N_t = 15$ time snapshots (still with $N_s = 200$ forward samples, yielding $N_d = 3750$ training data) since the temporal evolution of the PDF is much quicker, along with $N_c = 15000$ total collocation points. Other hyper-parameters are the same. This neural net took 534 s to train, reached a validation RMSE of 1.12 e–02, which scales to a NRMSE of 3.06 e–03. Qualitatively, the probability quickly moves to the corner of the limit cycle and then accumulates there quite rapidly. Trajectory sampling beyond this time interval (to say $t_f = 3$) shows that the probability mass then moves slowly along the limit cycle, with ever-increasing density reaching over 800 by $t_f = 3$. This experiment shows that differently scaled probability densities are very difficult for these neural nets to learn. Improving the behavior when faced with large density spikes will likely be a central challenge when we add control, since most desired controls will push the density into a thin, tall spike.

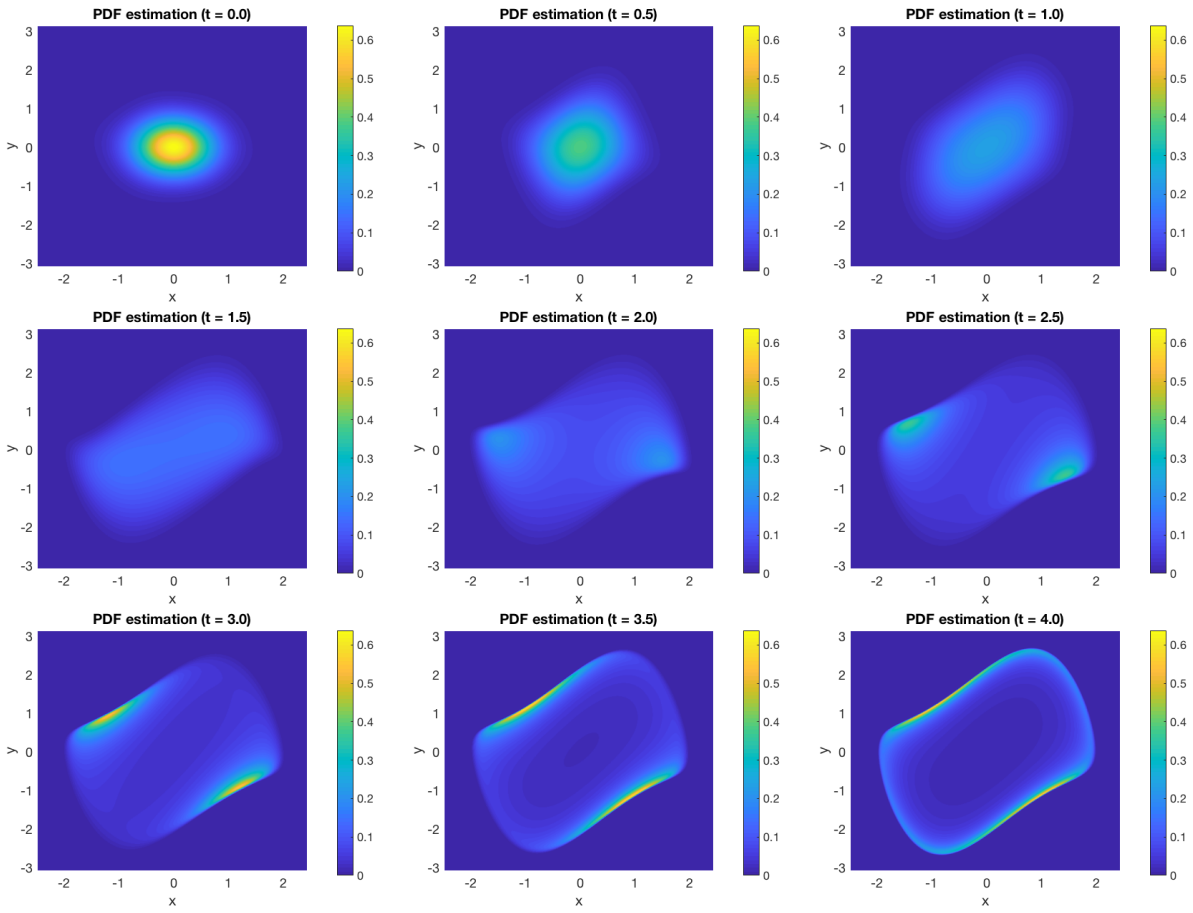


Figure 24: Time evolution of $p(x, y, t)$ for the Van der Pol oscillator (31) where the initial PDF is the product of two indepent Gaussians with means $\mu_x = \mu_y = 0$ and variances $\sigma_x^2 = \sigma_y^2 = 0.25$. Here the net is trained with data in the interval $t_0 = 0.0$ to $t_f = 4.0$.

2.2.9 Flow map prediction with deep neural networks

In this section we develop neural network models to predict the forward and the inverse flow maps of arbitrary nonlinear dynamical systems. With the flow map available, we can quickly predict solutions $\boldsymbol{x}(t) = \boldsymbol{\Phi}(\boldsymbol{x}_0, t)$ for any $(\boldsymbol{x}_0, t)$ in the training domain – without any numerical integration. Moreover, the flow map allows us to compute *any statistical property* of the system, including correlation between different phase variables and temporal correlations. For instance, we have

$$\mathbb{E}\left\{x_1(t)x_3(s)\right\} = \int_{-\infty}^{\infty} \Phi_1(\boldsymbol{x}_0, t)\Phi_3(\boldsymbol{x}_0, s)p_0(\boldsymbol{x}_0)d\boldsymbol{x}_0. \tag{33}$$

On the other hand, the solution to the Liouville equation (16) allows us to compute statistical properties at a specific time, i.e., but it does not include information on joint statistics at different times, e.g., temporal correlations.

Neural net architecture How do we build a neural network to predict the forward and the inverse flow maps? What inputs and outputs should the neural net take? Hereafter we discuss two architectures we propose, which we will study in detail in subsequent sections.

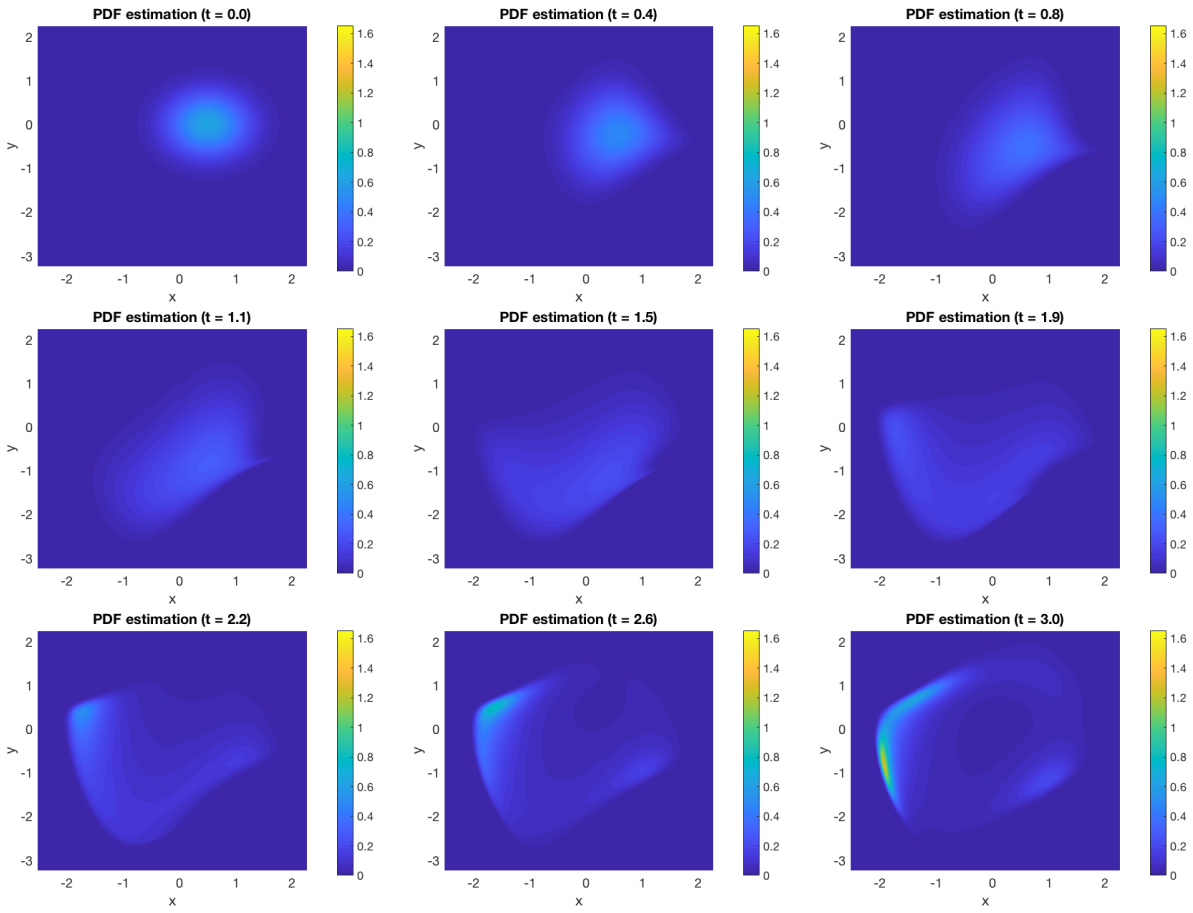


Figure 25: Time evolution of $p(x, y, t)$ for the Van der Pol oscillator (31) where the initial PDF is the product of two independent Gaussians with means $\mu_x = 0.5$, $\mu_y = 0$ and variances $\sigma_x^2 = \sigma_y^2 = 0.25$. Here the net was trained with data in the interval $t_0 = 0.0$ to $t_f = 3.0$.

- **"Jump" flow map estimator.** The first possibility is to learn it exactly in the form defined in equation (12), i.e., $\boldsymbol{x}(t) = \boldsymbol{\Phi}(\boldsymbol{x}_0, t)$. That is, we feed the net an initial condition $\boldsymbol{x}_0$ and a desired (final) time t. The net learns to output the state $\boldsymbol{x}(t)$ which evolved from that initial condition, so it "jumps" from the initial condition to the desired time. Similarly for the inverse flow map $\boldsymbol{x}_0 = \boldsymbol{\Phi}_0(\boldsymbol{x}, t)$, we feed in a state $\boldsymbol{x}$ and time t, and the net learns to output the initial condition $\boldsymbol{x}_0$ which would generate that state when the system is evolved to time t.
- **"Step" flow map estimator.** Another possibility is to discretize the time interval of interest $[t_0, t_f]$ into N_t time steps and have the net learn to step forward to the next time step. Thus, we feed the net inputs $\boldsymbol{x}(t)$ and t, and the net outputs $\boldsymbol{x}(t + \Delta t)$. Since we will soon add open-loop control, we will need to consider non-autonomous systems, which is why we also include time-dependence in the input. The net for learning the inverse map is much the same, with inputs $\boldsymbol{x}(t)$ and t, but the output is a step back: $\boldsymbol{x}(t - \Delta t)$.

The advantages of the first option are its simplicity and flexibility in the output: once trained, getting the map to final or initial state requires only a single evaluation of the appropriate neural net. The second option, on the other hand, seems to be easier to train (as we will discuss shortly). Furthermore, for the application of obtaining probability values at final time, we only need to train *one* neural net – i.e., that for approximating

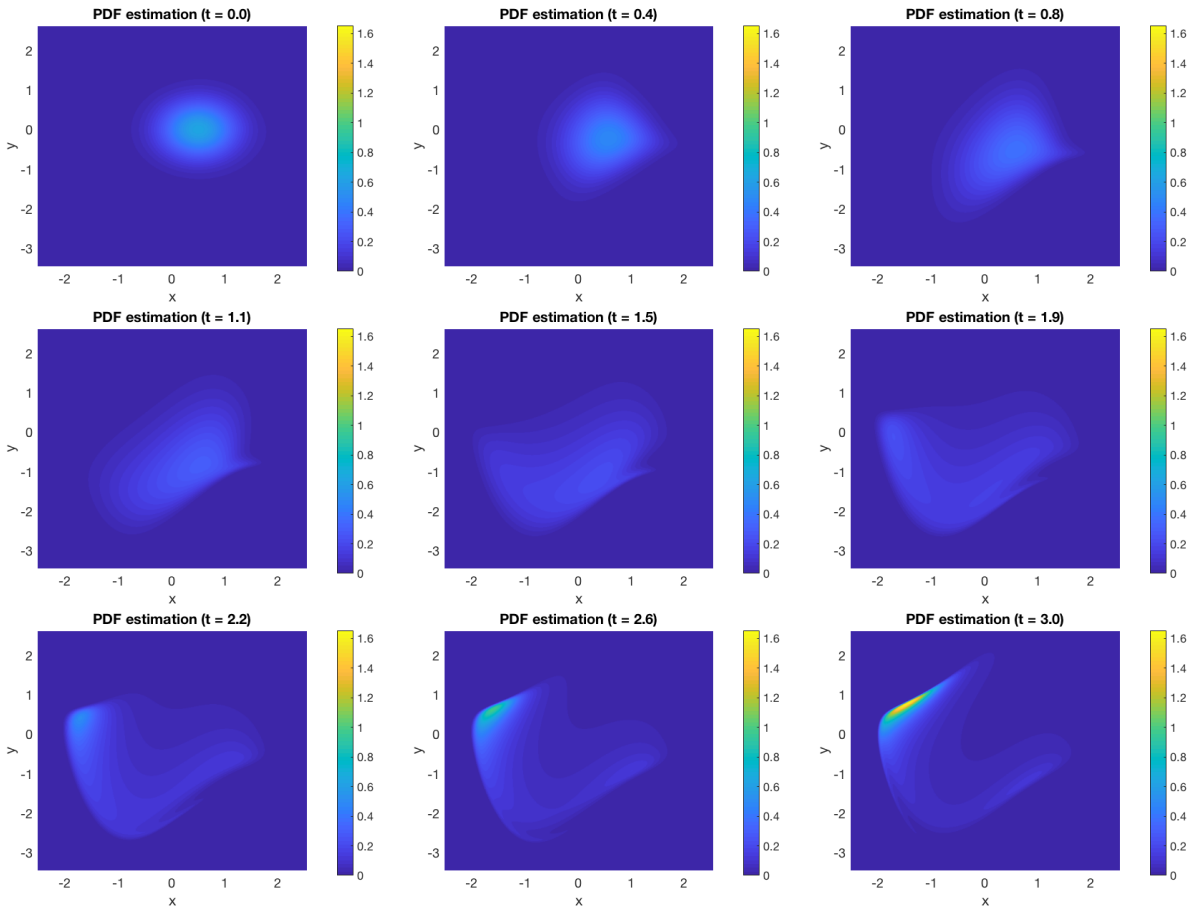


Figure 26: Time evolution of $p(x, y, t)$ for the Van der Pol oscillator (31) where the initial PDF is the product of two independent Gaussians with means $\mu_x = 0.5$, $\mu_y = 0$ and variances $\sigma_x^2 = \sigma_y^2 = 0.25$. Here the net was trained with data in the interval $t_0 = 0.0$ to $t_f = 3.0$.

the inverse flow map. Recall that evaluating eq. (16) for $p(\boldsymbol{x}, t_f)$ requires knowledge of $\boldsymbol{x}_0 = \boldsymbol{\Phi}_0(\boldsymbol{x}, t_f)$, as well as $\boldsymbol{x}(t) = \boldsymbol{\Phi}(\boldsymbol{x}_0, t)$ for enough values of $t \in [t_0, t_f]$ to accurately approximate the integral. A neural net which learns to step backwards in time automatically obtains this time series as we repeatedly evaluate it to reach $\boldsymbol{x}_0$.

Forward and inverse flow map approximation using independent neural nets We rewrote our data generation algorithms and physics-informed neural net for PDF approximation to *separately* approximate the forward and the inverse flow maps. The architecture for the physics-informed neural net approximating the forward flow map with a "jump" is sketched in Figure 28. If we use the physics-informed neural net here (often we do not need to, see discussion below), the total loss (for the forward flow map approximator) is the sum of two terms:

$$\text{loss}_{\text{fwd}}(\boldsymbol{\theta}; E) := \text{loss}_{\text{data}}(\boldsymbol{\theta}) + \mu_E \text{loss}_{\text{constraint}}(\boldsymbol{\theta}), \tag{34}$$

where $\boldsymbol{\theta}$ are the shared parameters of both nets and μ_E is a weight which depends on the training epoch E.

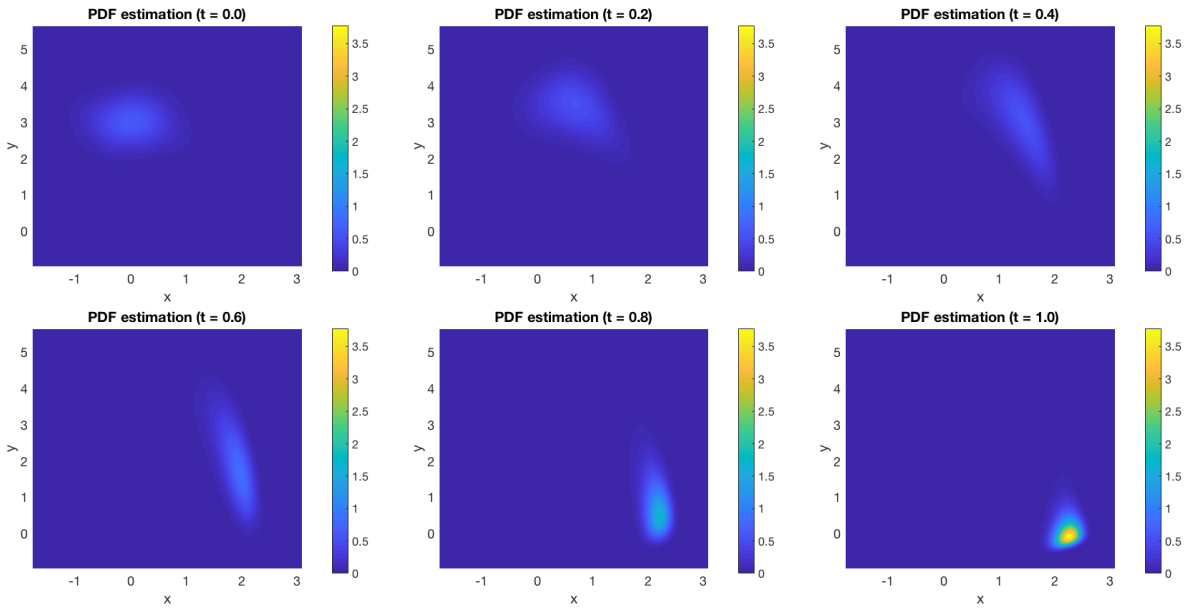


Figure 27: Time evolution of $p(x, y, t)$ for the Van der Pol oscillator (31) where the initial PDF is the product of two independent Gaussians with means $\mu_x = 0$, $\mu_y = 3$ and variances $\sigma_x^2 = \sigma_y^2 = 0.25$. Here the net was trained with data in the interval $t_0 = 0.0$ to $t_f = 1.0$.

Here the training data loss is a simple mean square error loss given by

$$\text{loss}_{\text{data}} = MSE_{\text{data}}(\boldsymbol{\theta}) \coloneqq \frac{1}{N_d} \sum_{j=1}^{N_d} \left[\boldsymbol{x}^{(j)} \left(t^{(j)} \right) - \widehat{\boldsymbol{\Phi}} \left(\boldsymbol{x}_0^{(j)}, t^{(j)} \right) \right]^2, \tag{35}$$

or a "logcosh" loss given by

$$\text{loss}_{\text{data}} = \text{logcosh}_{\text{data}}(\boldsymbol{\theta}) = \frac{1}{N_d} \sum_{j=1}^{N_d} \log \left[\cosh \left(\boldsymbol{x}^{(j)} \left(t^{(j)} \right) - \widehat{\boldsymbol{\Phi}} \left(\boldsymbol{x}_0^{(j)}, t^{(j)} \right) \right) \right]. \tag{36}$$

For the constraint loss we usually use a sum of MSE losses, with one term in the sum for each dimension in the state-space:

$$\text{loss}_{\text{constraint}}(\boldsymbol{\theta}; E) = \sum_{i=1}^{n} \frac{1}{N_{c,E}} \sum_{j=1}^{N_{c,E}} \left[R \left(\widehat{\Phi}_i \left(\boldsymbol{x}_0^{(j)}, t^{(j)} \right) \right) \right]^2, \tag{37}$$

where $N_{c,E}$ is the number of collocation points used in the epoch E. We also have the option of a sum of logcosh losses on these residuals. Like before, we define R as the residual when the neural net prediction $\widehat{\boldsymbol{\Phi}}(\boldsymbol{x}_0, t)$ is inserted into the set of flow map equations (13), i.e. $R(\widehat{\Phi}_i) \coloneqq \partial \widehat{\Phi}_i / \partial t - \boldsymbol{G} \cdot \nabla \widehat{\Phi}_i$, for $i = 1, \ldots, n$. The neural net architectures and loss terms for the "jump" inverse flow map estimator, as well as for the "step" flow map estimator, are similarly defined, with small and obvious changes in the inputs and outputs.

Bi-directional autoencoders to learn simultaneously the forward and inverse flow maps We also propose a method to join the two independent neural nets which estimate the forward and inverse flow maps, hence deriving neural net capable of estimating simultaneously both maps. As we will discuss later, the

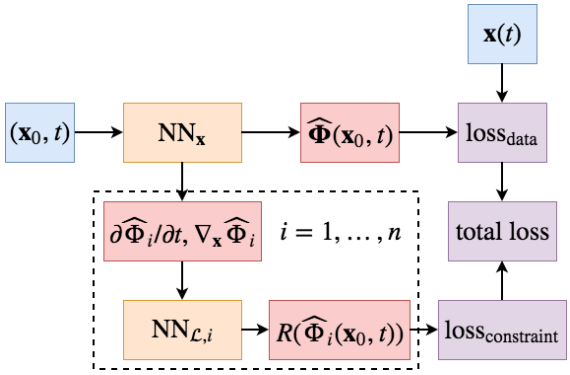


Figure 28: Architecture of the PINN for training the neural net $NN_{\boldsymbol{x}}$ using a set of supplementary neural nets $NN_{\mathcal{L},i}$, $i = 1, \ldots, n$ (n is the dimension of the system) to penalize predictions $\hat{\boldsymbol{x}}(t)$ which deviate from the flow map equations (13).

inverse flow map is typically much more challenging to approximate than the forward flow map. This is associated with well-known challenges in backward numerical integration of dynamical systems. As we will show hereafter, by using the bi-directional autoencoders we will be able to mitigate some difficulties in the approximation of the inverse flow map, by using the forward flow map.

To accomplish this, we first *pre-train both nets independently*. Next we train them simultaneously using a cost function which is the weighted sum of the cost functions (34) of the forward and inverse maps, and two autoencoding terms which encourage the pair of neural nets to be left and right inverses of one another. We call this pair of neural nets a *bi-directional autoencoder*. The total loss function for the bi-directional autoencoder can be written as

$$\begin{aligned}\mathrm{loss}_{\mathrm{total}}(\boldsymbol{\theta}_{\mathrm{fwd}}, \boldsymbol{\theta}_{\mathrm{inv}}; E) = \mathrm{loss}_{\mathrm{fwd}}(\boldsymbol{\theta}_{\mathrm{fwd}}; E) + \lambda_1 \cdot \mathrm{loss}_{\mathrm{inv}}(\boldsymbol{\theta}_{\mathrm{inv}}; E) \\ + \lambda_2 \cdot \mathrm{loss}_{\boldsymbol{x}_0\ \mathrm{autoencode}}(\boldsymbol{\theta}_{\mathrm{fwd}}, \boldsymbol{\theta}_{\mathrm{inv}}) + \lambda_3 \cdot \mathrm{loss}_{\boldsymbol{x}\ \mathrm{autoencode}}(\boldsymbol{\theta}_{\mathrm{fwd}}, \boldsymbol{\theta}_{\mathrm{inv}}).\end{aligned} \tag{38}$$

Here we denoted by $\boldsymbol{\theta}_{\mathrm{fwd}}$, $\boldsymbol{\theta}_{\mathrm{inv}}$ and $\mathrm{loss}_{\mathrm{fwd}}$, $\mathrm{loss}_{\mathrm{inv}}$ as the parameters and loss functions (see eq. 34) of the forward and inverse flow map approximators, respectively. Further, we let λ_1, λ_2, and λ_3 be constant scalar weights, and

$$\mathrm{loss}_{\boldsymbol{x}_0\ \mathrm{autoencode}}(\boldsymbol{\theta}_{\mathrm{fwd}}, \boldsymbol{\theta}_{\mathrm{inv}}) \coloneqq \frac{1}{N_d}\sum_{j=1}^{N_d}\left[\boldsymbol{x}_0^{(j)} - \widehat{\boldsymbol{\Phi}}_0\left(\widehat{\boldsymbol{\Phi}}\left(\boldsymbol{x}_0^{(j)}, t^{(j)}\right), t^{(j)}\right)\right]^2. \tag{39}$$

We have written the inverse autoencoding term with MSE loss here, but a logcosh loss can also work. Similarly, the forward autoencoding term is given by

$$\mathrm{loss}_{\boldsymbol{x}\ \mathrm{autoencode}}(\boldsymbol{\theta}_{\mathrm{fwd}}, \boldsymbol{\theta}_{\mathrm{inv}}) \coloneqq \frac{1}{N_d}\sum_{j=1}^{N_d}\left[\boldsymbol{x}^{(j)} - \widehat{\boldsymbol{\Phi}}\left(\widehat{\boldsymbol{\Phi}}_0\left(\boldsymbol{x}^{(j)}, t^{(j)}\right), t^{(j)}\right)\right]^2. \tag{40}$$

Figure 29 helps make these loss terms more intuitive. Note that eqs. (39) and (40) are for the "jump" flow map estimator – analogous terms for the "step" estimator can be easily defined.

Data generation for the "Jump" flow map estimator We started the data generation process by defining a region of interest for the initial condition, in these tests we used a hypercube for convenience. We then

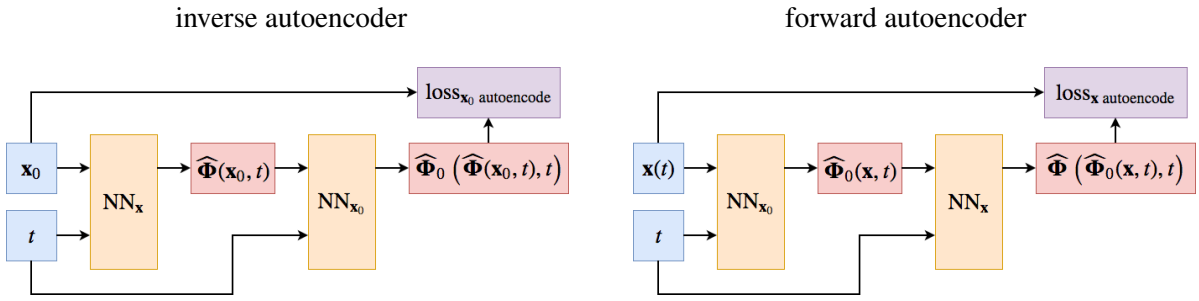


Figure 29: Diagrams representing the inverse (eq. (39)) and forward (eq. (40)) autoencoders for flow map approximation.

randomly sampled (by e.g. Latin hypercube sampling) from this region, giving us a set of points $\left\{\boldsymbol{x}_0^{(j)}\right\}$, $j = 1, \ldots, N_s^f$. We then randomly selected $2/3$ of these points for training, and $1/3$ for validation. Finally from each point we numerically integrated forward to time t_f, saving a set of points $\left(\boldsymbol{x}_0^{(j)}, \boldsymbol{x}^{(j)}\left(t^{(k)}\right), t^{(k)}\right)$ for each time step $t^{(k)} \in [t_0, t_f]$ used by the adaptive timestepping in the chosen integrator. For all tests here we used SciPy's `dop853` [12, 13] ODE integrator. While this forward data can also be used for training the inverse approximator, we reasoned that by creating additional backward samples we could fill in gaps in the final time data. For simplicity we used Latin hypercube sampling from the hypercube enclosing the data points at final time to obtain a set of points $\left\{\boldsymbol{x}^{(j)}(t_f)\right\}$, $j = 1, \ldots, N_s^b$. We randomly selected $2/3$ of these for training data and the remaining $1/3$ for validation data. Next we numerically integrated backwards to t_0, again saving the outputs at each adaptive time step, and used $\boldsymbol{x}(t_0)$ to complete the backward data: $\left(\boldsymbol{x}^{(j)}(t_0), \boldsymbol{x}^{(j)}\left(t^{(k)}\right), t^{(k)}\right)$. Note that while forward integration is typically successful for well-behaved systems, backward integration can yield trajectories which escape in finite time and often fails for these same systems. Thus we only saved trajectories for which the integration was successful and $\boldsymbol{x}(t_0)$ was not too far outside of the original sampling region of $\boldsymbol{x}_0$. Collocation points $\{\boldsymbol{x}_0^{(j)}, t^{(j)}\}$, $j = 1, \ldots, N_c$, for enforcing the flow map equations (13) consist of training data, supplemented by additional points drawn by Latin hypercube sampling from the domain of $\boldsymbol{x}_0$ and time interval $[t_0, t_f]$. Collocation points $\{\boldsymbol{x}^{(j)}, t^{(j)}\}$, $j = 1, \ldots, N_c$, for enforcing the inverse flow map equations (14) consist of the training data, plus additional points generated by Latin hypercube sampling from the entire space-time domain of the training region. As we will demonstrate below, using the PINN often makes results *less* accurate. We suspect the collocation point sampling method is to blame: it is easy to imagine that Latin hypercube sampling (or uniform sampling, etc.) yields many points which will be far away from the main cloud of trajectories, thus introducing numerous outliers into the collocation data. It is possible that sampling inside a (nearly) convex hull around the trajectory tube could make the PINN more accurate.

Data generation for the "Step" flow map estimator The major difference between the two data generation algorithms is simply where – or more precisely, when – we save the data. For the networks which learn to step forward or backward in time, we discretize the time interval $[t_0, t_f]$ into N_t time snapshots $t_0, t_1 = t_0 + \Delta t, \ldots, t_k = t_0 + k\Delta t, \ldots, t_{N_t} = t_f$, where $\Delta t := (t_f - t_0)/N_t$, and have the ODE integrator output results at those time snapshots. The kind of data we save is, of course, different. We now save data for the forward map in the form $(\boldsymbol{x}(t), \boldsymbol{x}(t + \Delta t), t + \Delta t)$, so that each input $\boldsymbol{x}(t)$ is paired with its value at the next time step, $\boldsymbol{x}(t + \Delta t)$, and the time at output $t + \Delta t$. Recall that we include time-dependence so that we can generalize to non-autonomous systems. The form of the backward map data is simply $(\boldsymbol{x}(t + \Delta t), \boldsymbol{x}(t), t)$. For collocation points, the form of the flow map equations differs slightly. The

components of the forward flow map satisfy[2]

$$\frac{\partial \Phi_i(\boldsymbol{x}(t), t+\Delta t)}{\partial t} - \boldsymbol{G}(\boldsymbol{x}(t), t+\Delta t) \cdot \nabla \Phi_i(\boldsymbol{x}(t), t+\Delta t) = 0. \tag{41}$$

Similarly, the components of the inverse flow map satisfy

$$\frac{\partial \Phi_i^{-1}(\boldsymbol{x}(t+\Delta t), t)}{\partial t} + \boldsymbol{G}(\boldsymbol{x}(t+\Delta t), t) \cdot \nabla \Phi_i^{-1}(\boldsymbol{x}(t+\Delta t), t) = 0. \tag{42}$$

Here we have written $\Phi_i^{-1}(\cdot)$ to denote the ith component of inverse flow map which maps back one time step, distinguishing it from the ith component of the full ("jump") inverse flow map, $\Phi_{0,i}$. Thus, the collocation points for the forward map look like $\left\{\boldsymbol{x}^{(j)}\left(t^{(j)}\right), t^{(j)} + \Delta t\right\}$, for $j = 1, \ldots, N_c$. These points again consist of the training data, augmented with spatial data drawn from the training region, and time values picked from the discrete time instances $t_0, t_1, \ldots, t_{N_t-1}$. For the inverse map we have $\left\{\boldsymbol{x}^{(j)}\left(t^{(j)} + \Delta t\right), t^{(j)}\right\}$, $j = 1, \ldots, N_c$, with time values also picked from $t_0, t_1, \ldots, t_{N_t-1}$.

Numerical results

In this section we present numerical results of flow map estimation using the algorithms described in the previous sections. For consistency with previous error metrics, here we we use the normalized root mean square validation error defined as

$$NRMSE_{\text{val}} = \frac{\sqrt{\dfrac{1}{N_v}\displaystyle\sum_{j=1}^{N_v}\sum_{i=1}^{n}\left[x_i^{(j)}\left(t^{(j)}+\Delta t\right) - \widehat{\Phi}_i^{(j)}\left(x_i^{(j)}\left(t^{(j)}\right), t^{(j)}\right)\right]^2}}{\sqrt{\dfrac{1}{n}\displaystyle\sum_{i=1}^{n}\left[\max_j x_i^{(j)}\left(t^{(j)}+\Delta t\right) - \min_j x_i^{(j)}\left(t^{(j)}+\Delta t\right)\right]^2}}. \tag{43}$$

We tested our algorithms by approximating the flow maps generated by the Van der Pol oscillator (31) in the time interval $[t_0, t_f] = [0, 4]$. We conducted one set of tests with initial conditions sampled from the region $x_0, y_0 \in [-2, 2]$, i.e., within and outside the limit cycle (see Figure 22). Backward trajectories for which $x(t_0)$ or $y(t_0)$ were not in $[-3, 3]$ were simply discarded. We conducted another set of tests with initial conditions sampled from the region $x_0 \in [-1, 3]$, $y_0 \in [-2, 2]$. Backward trajectories for which $x(t_0)$ were not in $[-2, 4]$ or $y(t_0) \notin [-3, 3]$ were discarded. For the "jump" neural net estimator, we used 100 forward samples for training and 50 for validation, and 150 backward samples for training and 75 for validation. Not all samples integrated successfully, unsuccessful integrations were not resampled. For the "step" flow map estimator, we used 40 forward samples for training and 20 for validation, and 60 backward samples for training and 30 for validation. Each trajectory was evaluated at $N_t = 50$ time snapshots. Not all samples integrated successfully, unsuccessful integrations were not resampled. For all tests we used 8 hidden layers with 20 neurons each. Each test was conducted 5 times, using the same 5 random seeds to enable better comparison. Other parameters are discussed where relevant.

Forward "jump" flow map estimator results We first tested the forward "jump" flow map estimator. We conducted one set of tests without the physics-informed neural net, i.e., using a plain feed-forward neural

[2]Note that equations (41) and (42) hold for non-autonomous systems of the form $\dot{\boldsymbol{x}} = \boldsymbol{G}(\boldsymbol{x}, t)$, $\boldsymbol{x}(0) = \boldsymbol{x}_0$.

Plain neural net (no PINN) with $N_c = 5000$ total collocation points					
trial	N_d	N_v	training time	RMSE	NRMSE
1	2492	1199	115 s	4.90 e–03	8.81 e–04
2	2506	1223	130 s	4.57 e–03	7.99 e–04
3	2523	1218	110 s	7.29 e–03	1.38 e–03
4	2570	1262	96 s	6.11 e–03	1.12 e–03
5	2622	1150	124 s	7.12 e–03	1.27 e–03
mean	2543	1210	115 s	6.00 e–03	1.09 e–03
SD	53	41	13 s	1.24 e–03	2.48 e–04

Table 14: Accuracy and training time of a plain neural net for approximating the forward "jump" flow map of the Van der Pol oscillator (31) with $x_0, y_0 \in [-2, 2] \times [-2, 2]$. Listed runtimes are for a naive single CPU implementation using TensorFlow 1.8 [1] on a 2012 MacBook Pro with 2.5 GHz Intel Core i5 processor and 4 GB RAM.

PINN with $N_c = 5000$ total collocation points					
trial	N_d	N_v	training time	RMSE	NRMSE
1	2492	1199	519 s	4.01 e–03	7.22 e–04
2	2506	1223	530 s	2.38 e–03	4.16 e–04
3	2523	1218	494 s	3.36 e–03	6.34 e–04
4	2570	1262	502 s	1.83 e–03	3.34 e–04
5	2622	1150	463 s	3.58 e–03	6.37 e–04
mean	2543	1210	506 s	3.03 e–03	5.49 e–04
SD	53	41	26 s	8.99 e–04	1.65 e–04

Table 15: Accuracy and training speed of a PINN for approximating the forward "jump" flow map of the Van der Pol oscillator (31) with $x_0, y_0 \in [-2, 2] \times [-2, 2]$.

net, and another with the PINN using $N_c = 5000$ total collocation points. When we used the PINN, we kept a constant scalar penalty weight of $\mu_E = 1.0$, and randomly selected collocation points according to (26), with $E_{\text{max}} = 10$ iterations/epochs. All loss terms were modeled as mean square errors. In all trials, data generation took approximately 1 second. In Tables 14 and 15 we summarize results of the tests with $x_0, y_0 \in [-2, 2]$. In particular, results of Table 15 are obtained with physics-informed "jump" forward flow map estimators. Similarly, Tables 16 and 17 summarize results with $x_0 \in [-1, 3]$ and $y_0 \in [-2, 2]$, with and without PINN.

It is seen that the "jump" flow map estimator is consistent and relatively accurate even without the PINN. When we use a PINN, the accuracy almost doubles, but training takes about four times as long. Some additional tests (data not shown here) indicated that increasing the number of collocation points N_c did not, in general, further improve accuracy, but did increase the training time. For visualization, in figure 30 we plot the validation data and corresponding neural net predictions from one trial of each region of initial conditions.

Inverse "jump" flow map estimator results We conducted one set of tests without the PINN, and another with the PINN using $N_c = 5000$ total collocation points. When we used the PINN, we kept a constant scalar penalty weight of $\{\mu_E\} \equiv 1.0$, and randomly selected collocation points according to (26) with $E_{\text{max}} = 10$ iteration/epochs. Latin hypercube sampling created many outliers in the collocation data. We sought to blunt

Plain neural net (no PINN)					
trial	N_d	N_v	training time	RMSE	NRMSE
1	2512	1285	123 s	2.74 e–03	5.10 e–04
2	2485	1228	122 s	4.45 e–03	8.07 e–04
3	2579	1213	156 s	5.37 e–03	9.89 e–04
4	2568	1249	110 s	4.13 e–03	7.53 e–04
5	2654	1191	100 s	4.24 e–03	8.09 e–04
mean	2560	1223	122 s	4.17 e–03	7.74 e–04
SD	66	36	21 s	9.44 e–04	1.72 e–04

Table 16: Accuracy and training speed of a plain neural net for approximating the forward "jump" flow map of the Van der Pol oscillator (31) with $x_0, y_0 \in [-1, 3] \times [-2, 2]$.

PINN with $N_c = 5000$ total collocation points					
trial	N_d	N_v	training time	RMSE	NRMSE
1	2512	1285	428 s	2.35 e–03	4.38 e–04
2	2485	1228	491 s	1.95 e–03	3.54 e–04
3	2579	1213	508 s	3.17 e–03	5.84 e–04
4	2568	1249	518 s	2.76 e–03	5.03 e–04
5	2654	1191	559 s	2.52 e–03	4.81 e–04
mean	2560	1223	501 s	2.55 e–03	4.72 e–04
SD	66	36	48 s	4.55 e–04	8.47 e–05

Table 17: Accuracy and training speed of a PINN for approximating the forward "jump" flow map of the Van der Pol oscillator (31) with $x_0, y_0 \in [-1, 3] \times [-2, 2]$.

the impact of these outliers by using a logcosh loss on the inverse flow map penalty term, since logcosh can be less sensitive to outliers than MSE [19]. For the data loss we still used MSE. In Tables 18 and 19 we summarize results of the tests we performed with $x_0, y_0 \in [-2, 2]$. In particular, Table 19 refers to case where we use a physics-informed neural network (residual of the forward flow map equation added in the cost function). In Tables 20 and 21 we summarize the same kind of results, but with initial conditions $x_0 \in [-1, 3]$ and $y_0 \in [-2, 2]$.

These results have two clear implications. First, that the inverse flow map is much harder to approximate than the forward flow map, as we expected. It is likely that we need more and better data, and perhaps a differently structered neural net. Second, the physics-informed neural net contributes nothing in these implementations. Moreover, when we trained the without using the PINN, it was still possible to compute the constraint loss term (37). If we chose to do this extra computation, we saw that for the forward map the constraint loss decreased naturally as we optimized with respect to the data only. However, in the case of the inverse map, the constraint loss was largely unaffected by minimizing the data loss. This indicates that there is a problem with the data generation algorithm and choices of hyperparameters. As discussed previously, the underlying cause here could be outliers in the collocation points. In the next quarter we will take steps to understand this behavior and experiment with different solutions.

"Jump" flow map estimator with bidirectional autoencoder results We conducted one set of tests of the bidirectional autoencoder sketched in Figure 29 using no PINN, and another using a PINN enforcing the *forward* flow map equations (13) *only*, with $N_c = 5000$ collocation points. We didn't go through the trouble

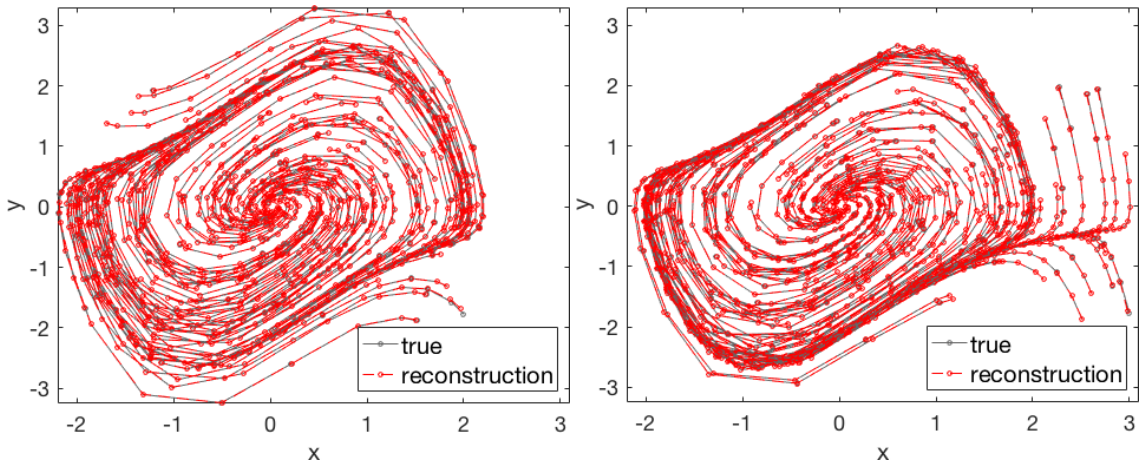


Figure 30: Validation data and corresponding neural net reconstructions by a PINN which approximates the forward "jump" flow map of the Van der Pol oscillator (31). Left: $x_0, y_0 \in [-2, 2] \times [-2, 2]$ from the model trained in trial 1 of Table 15, for which there were 96 successful validation integrations. Right: $x_0, y_0 \in [-1, 3] \times [-2, 2]$ from the model trained in trial 1 of Table 17, for which there were 100 successful validation integrations.

Plain neural net (no PINN) with $N_c = 5000$ total collocation points					
trial	N_d	N_v	training time	RMSE	NRMSE
1	2492	1199	108 s	3.96 e–02	1.01 e–02
2	2506	1223	114 s	3.02 e–02	7.69 e–03
3	2523	1218	127 s	8.29 e–02	2.19 e–02
4	2570	1262	76 s	5.20 e–02	1.39 e–02
5	2622	1150	83 s	4.41 e–02	1.14 e–02
mean	2543	1210	102 s	4.98 e–02	1.30 e–02
SD	53	41	21 s	2.01 e–02	5.46 e–03

Table 18: Accuracy and training speed of a plain neural net for approximating the inverse "jump" flow map of the Van der Pol oscillator (31) with $x_0, y_0 \in [-2, 2] \times [-2, 2]$.

of testing it extensively with the inverse flow map PINN, since a few initial tests already indicated this was ineffective. When using the PINN we kept a constant scalar penalty weight $\mu_E = 1$, and randomly selected collocation points according to (26) with $E_{\text{max}} = 10$. For the tests with $x_0, y_0 \in [-2, 2] \times [-2, 2]$ we used $\lambda_1 = 10^{-1}$, $\lambda_2 = 10^{-2}$, and $\lambda_3 = 10^{-2}$. For the tests with $x_0, y_0 \in [-1, 3] \times [-2, 2]$ we used $\lambda_1 = 10^{-3}$, $\lambda_2 = 10^{-4}$, and $\lambda_3 = 10^{-4}$. Note that λ_1 is the weight on the inverse loss term in the cost function (38). Similarly, λ_2 and λ_3 are the weights on the $\boldsymbol{x}_0$ and $\boldsymbol{x}$ autoencoding terms, respectively. In Tables 22 and 23 we summarize the results of some of our tests with $x_0, y_0 \in [-2, 2]$. Tables 24 and 25 summarize similar results we obtained with $x_0 \in [-1, 3]$, $y_0 \in [-2, 2]$. These tests show some success of the bidirectional autoencoder we proposed. We expect that modifying the training process so that only the inverse flow map estimator is trained with the autoencoding terms will improve both accuracy and efficiency. This scheme would leave the more accurate forward estimator (with or without the PINN) intact, and train the inverse estimator to be the inverse of the forward estimator. This change will require some rewriting of the code to implement.

PINN with $N_c = 5000$ total collocation points					
trial	N_d	N_v	training time	RMSE	NRMSE
1	2492	1199	483 s	4.65 e–02	1.18 e–02
2	2506	1223	308 s	3.32 e–02	8.44 e–03
3	2523	1218	355 s	1.17 e–01	3.11 e–02
4	2570	1262	361 s	3.15 e–02	8.41 e–03
5	2622	1150	528 s	4.83 e–02	1.25 e–02
mean	2543	1210	407 s	5.53 e–02	1.45 e–02
SD	53	41	94 s	3.53 e–02	9.50 e–03

Table 19: Accuracy and training speed of a PINN for approximating the inverse "jump" flow map of the Van der Pol oscillator (31) with $x_0, y_0 \in [-2, 2] \times [-2, 2]$.

Plain neural net (no PINN)					
trial	N_d	N_v	training time	RMSE	NRMSE
1	2512	1285	41 s	2.78 e–01	7.09 e–02
2	2485	1228	33 s	3.26 e–01	8.30 e–02
3	2579	1213	28 s	4.51 e–01	1.19 e–01
4	2568	1249	24 s	3.75 e–01	1.00 e–01
5	2654	1191	36 s	2.78 e–01	7.19 e–02
mean	2560	1223	32 s	3.42 e–01	8.90 e–02
SD	66	36	7 s	7.32 e–02	2.05 e–02

Table 20: Accuracy and training speed of a plain neural net for approximating the inverse "jump" flow map of the Van der Pol oscillator (31) with $x_0, y_0 \in [-1, 3] \times [-2, 2]$.

Forward "step" flow map estimator results In this section we present numerical results of the feed-forward "step" flow map estimator without PINN (see section 2.2.9). In Table 26 contains results of the tests with $x_0, y_0 \in [-2, 2]$, and Table 27 we summarize the results we obtained for $x_0 \in [-1, 3]$, $y_0 \in [-2, 2]$. The results we obtained with plain feed-forward deep neural networks and "step" forward flow map estimators are quite good: we get more accurate models in less time than when we train the full "jump" forward flow map. We expect that the inverse flow map will also be somewhat more accurate than the "jump" version, once we implement it successfully. For visualization, in Figure 31 we plot the validation data and corresponding neural net predictions from one trial of each region of initial conditions.

2.3 Numerical methods to solve data-driven PDF equations

The physics-informed neural network technique we studied in section 2.2.6 can be viewed as a data-driven discrete least squares approach to solve the Liouville equation. In addition to such technique, we studied direct methods to solve reduced-order PDF equations, i.e., PDF equations for quantities of interest. To illustrate the methodology, consider again the n-dimensional system (11). We have seen that the Liouville equation (15) describes the exact dynamics of the joint PDF of state variables $\boldsymbol{x}(t)$. In most cases, however, we are only interested in the dynamics of a real-valued phase space function

$$u(\boldsymbol{x}) = \mathbb{R}^n \to \mathbb{R} \qquad \text{(observable)}. \tag{44}$$

In models of disease propagation, this phase space function may be represented by the population of susceptible individuals, e.g., by the first component of a nonlinear epidemic model. In this case we set

PINN with $N_c = 5000$ total collocation points					
trial	N_d	N_v	training time	RMSE	NRMSE
1	2512	1285	164 s	3.39 e–01	8.62 e–02
2	2485	1228	123 s	3.20 e–01	8.15 e–02
3	2579	1213	205 s	5.17 e–01	1.37 e–01
4	2568	1249	131 s	3.40 e–01	9.06 e–02
5	2654	1191	165 s	2.66 e–01	6.87 e–02
mean	2560	1223	158 s	3.56 e–01	9.28 e–02
SD	66	36	33 s	9.47 e–02	2.60 e–02

Table 21: Accuracy and training speed of a PINN for approximating the inverse "jump" flow map of the Van der Pol oscillator (31) with $x_0, y_0 \in [-1, 3] \times [-2, 2]$.

Vanilla neural net (no PINN)					
trial	N_d	N_v	training time	NRMSE (forward)	NRMSE (inverse)
1	2492	1199	328 s	8.63 e–04	7.47 e–03
2	2506	1223	334 s	6.33 e–04	8.75 e–03
3	2523	1218	310 s	7.87 e–04	1.96 e–02
4	2570	1262	314 s	8.24 e–04	5.43 e–03
5	2622	1150	321 s	7.81 e–04	8.25 e–03
mean	2543	1210	321 s	7.78 e–04	9.90 e–03
SD	53	41	10 s	8.73 e–05	5.57 e–03

Table 22: Accuracy and training speed of a pair of plain feed-forward neural nets for approximating the forward and inverse "jump" flow maps of the Van der Pol oscillator (31) with $x_0, y_0 \in [-2, 2] \times [-2, 2]$, trained as a bidirectional autoencoder.

$u(\boldsymbol{x}(t)) = x_1(t)$. The probability density function of such observable can be represented as

$$p(z,t) = \int_{-\infty}^{\infty} \cdots \int_{-\infty}^{\infty} \delta\left(z - u(\boldsymbol{x})\right) p(\boldsymbol{x},t) d\boldsymbol{x}, \tag{45}$$

where $\delta(\cdot)$ is the Dirac's delta function (see [14, 25, 21]) and z is a random variable representing the value of $u(\boldsymbol{x}(t))$. Multiplying the Liouville equation (15) by $\delta\left(z - u(\boldsymbol{x})\right)$ and integrating over all phase variables yields

$$\frac{\partial p(z,t)}{\partial t} + \int_{-\infty}^{\infty} \cdots \int_{-\infty}^{\infty} e^{ia(z-u(\boldsymbol{x}))} \nabla \cdot \left(\boldsymbol{G}(\boldsymbol{x}) p(\boldsymbol{x},t)\right) \boldsymbol{x} da = 0. \tag{46}$$

Here we employed the Fourier representation of the Dirac delta function. In general, equation (46) is *unclosed* in the sense that there are terms at the right hand side that cannot be computed based on $p(z,t)$ alone. If we set $\boldsymbol{u}(\boldsymbol{x}(t)) = x_k(t)$, i.e., we are interested in the k-th component of the dynamical system (11) then

PINN with $N_c = 5000$ total forward collocation points					
trial	N_d	N_v	training time	NRMSE (forward)	NRMSE (inverse)
1	2492	1199	912 s	4.40 e–04	7.20 e–03
2	2506	1223	963 s	3.53 e–04	8.47 e–03
3	2523	1218	885 s	4.21 e–04	1.94 e–02
4	2570	1262	949 s	3.59 e–04	4.50 e–03
5	2622	1150	764 s	6.32 e–04	7.89 e–03
mean	2543	1210	895 s	4.41 e–04	9.49 e–03
SD	53	41	79 s	1.13 e–04	5.74 e–03

Table 23: Accuracy and training speed of a pair of neural nets for approximating the simultaneously forward and inverse "jump" flow maps of the Van der Pol oscillator (31) with $x_0, y_0 \in [-2, 2] \times [-2, 2]$, trained as a bidirectional autoencoder. The neural net approximating the forward map is pre-trained as a plain neural net (no PINN); during the joint training we add in the PDE constraint penalty term (37).

Vanilla neural net (no PINN)					
trial	N_d	N_v	training time	NRMSE (forward)	NRMSE (inverse)
1	2512	1285	265 s	9.31 e–04	7.55 e–02
2	2485	1228	232 s	9.05 e–04	8.38 e–02
3	2579	1213	222 s	6.83 e–04	1.36 e–01
4	2568	1249	217 s	1.53 e–03	7.42 e–02
5	2654	1191	260 s	9.30 e–04	6.76 e–02
mean	2560	1223	239 s	9.96 e–04	8.74 e–02
SD	66	36	22 s	3.16 e–04	2.78 e–02

Table 24: Accuracy and training speed of a pair of plain neural nets for approximating the forward and inverse "jump" flow maps of the Van der Pol oscillator (31) with $x_0, y_0 \in [-1, 3] \times [-2, 2]$, trained as a bidirectional autoencoder. Training times include the pre-training of both separate neural nets and the joint training.

(46) reduces to[3]

$$\frac{\partial p(x_k, t)}{\partial t} + \int_{-\infty}^{\infty} \cdots \int_{-\infty}^{\infty} \frac{\partial}{\partial x_k} \left(G_k(\boldsymbol{x}) p(\boldsymbol{x}, t) \right) dx_1 \ldots dx_{k-1} dx_{k+1} \ldots dx_n = 0. \tag{47}$$

The specific form of this equation depends on the underlying dynamical system, i.e., on the nonlinear map $\boldsymbol{G}(\boldsymbol{x})$. Let us provide a simple example.

Example 2.1 (Lorenz-96 system) Consider the Lorenz-96 dynamical system

$$\frac{dx_i}{dt} = (x_{i+1} - x_{i-2}) x_{i-1} - x_i + F \qquad \text{for} \quad i = 1, 2, \ldots, n, \tag{48}$$

where $x_0 = x_d$ and $x_1 = x_{n+1}$. The associated Liouville equation is

[3] By using integration by parts and assuming that the joint PDF $p(\boldsymbol{x}, t)$ decays to zero sufficiently fast at infinity we obtain

$$\int_{-\infty}^{\infty} \cdots \int_{-\infty}^{\infty} \nabla \cdot (\boldsymbol{G}(\boldsymbol{x}) p(\boldsymbol{x}, t)) \, dx_1 \ldots dx_{k-1} dx_{k+1} \ldots dx_n =$$
$$\int_{-\infty}^{\infty} \cdots \int_{-\infty}^{\infty} \frac{\partial}{\partial x_k} \left(G_k(\boldsymbol{x}) p(\boldsymbol{x}, t) \right) dx_1 \ldots dx_{k-1} dx_{k+1} \ldots dx_n.$$

PINN with $N_c = 5000$ total forward collocation points					
trial	N_d	N_v	training time	NRMSE (forward)	NRMSE (inverse)
1	2512	1285	602 s	7.25 e–04	7.16 e–02
2	2485	1228	492 s	5.87 e–04	7.80 e–02
3	2579	1213	582 s	6.12 e–04	1.34 e–01
4	2568	1249	763 s	5.66 e–04	6.64 e–02
5	2654	1191	717 s	5.42 e–04	6.36 e–02
mean	2560	1223	631 s	6.06 e–04	8.27 e–02
SD	66	36	109 s	7.12 e–05	2.92 e–02

Table 25: Accuracy and training speed of a pair of neural nets for approximating the forward and inverse "jump" flow maps of the Van der Pol oscillator (31) with $x_0, y_0 \in [-1, 3] \times [-2, 2]$, trained as a bidirectional autoencoder. The neural net approximating the forward map is pre-trained as a plain neural net (no PINN); during the joint training we add in the PDE constraint penalty term (37). Training times include the pre-training of both separate neural nets and the joint training.

Vanilla neural net (no PINN)					
trial	N_d	N_v	training time	RMSE	NRMSE
1	4214	2058	52 s	1.81 e–03	3.66 e–04
2	4018	1813	38 s	2.01 e–03	3.91 e–04
3	3675	2107	21 s	2.53 e–03	4.57 e–04
4	4214	1911	27 s	3.10 e–03	5.85 e–04
5	3626	1911	27 s	1.66 e–03	2.97 e–04
mean	3959	1960	33 s	2.22 e–03	4.19 e–04
SD	285	120	12 s	5.91 e–04	1.09 e–04

Table 26: Accuracy and training time of a plain feed-forward neural net for approximating the forward "step" flow map of the Van der Pol oscillator (31) with $x_0, y_0 \in [-2, 2] \times [-2, 2]$.

$$\frac{\partial p(\boldsymbol{x}, t)}{\partial t} = -\sum_{i=1}^{d} \frac{\partial}{\partial x_i} \left[\Big((x_{i+1} - x_{i-2}) x_{i-1} - x_i + F \Big) p(\boldsymbol{x}, t) \right]. \tag{49}$$

Suppose we are interested in the PDF of the fifth component of the system, i.e., set $u(\boldsymbol{x}(t)) = x_5(t)$ in equation (44). By integrating (49) with respect to $x_1, \ldots, x_4, x_6, \ldots, x_n$ and assuming that $p(\boldsymbol{x}, t)$ decays fast enough at infinity, we obtain

$$\frac{\partial p(x_5, t)}{\partial t} = \frac{\partial}{\partial x_5} \left[\left(x_5 - F + \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} (x_3 - x_6) x_4 p(x_3, x_4, x_5, x_6, t) dx_3 dx_4 dx_6 \right] . \tag{50}$$

From this equation we see that the evolution of $p(x_5, t)$ depends on an integral involving $p(x_3, x_4, x_5, x_6, t)$. In other words, to solve an equation of this nature, we must find a way to approximate the term involving $p(x_3, x_4, x_5, x_6, t)$. To this end, it is convenient to first transform the integral at the right hand side by using conditional probabilities. Specifically, we can write the joint PDF of $x_3(t)$, $x_4(t)$, $x_5(t)$, and $x_6(t)$ at time t as

$$p(x_3, x_4, x_5, x_6, t) = p(x_5, t) p(x_3, x_4, x_6 | x_5, t), \tag{51}$$

Vanilla neural net (no PINN)					
trial	N_d	N_v	training time	RMSE	NRMSE
1	4214	2058	50 s	1.81 e–03	3.35 e–04
2	4018	1813	38 s	4.71 e–03	8.80 e–04
3	3724	2107	38 s	2.14 e–03	4.05 e–04
4	4018	1862	54 s	1.80 e–03	3.38 e–04
5	3577	1911	32 s	2.45 e–03	4.58 e–04
mean	3910	1950	42 s	2.58 e–03	4.83 e–04
SD	256	127	9 s	1.22 e–03	2.28 e–04

Table 27: Accuracy and training time of a plain feed-forward neural net for approximating the forward "step" flow map of the Van der Pol oscillator (31) with $x_0, y_0 \in [-1, 3] \times [-2, 2]$.

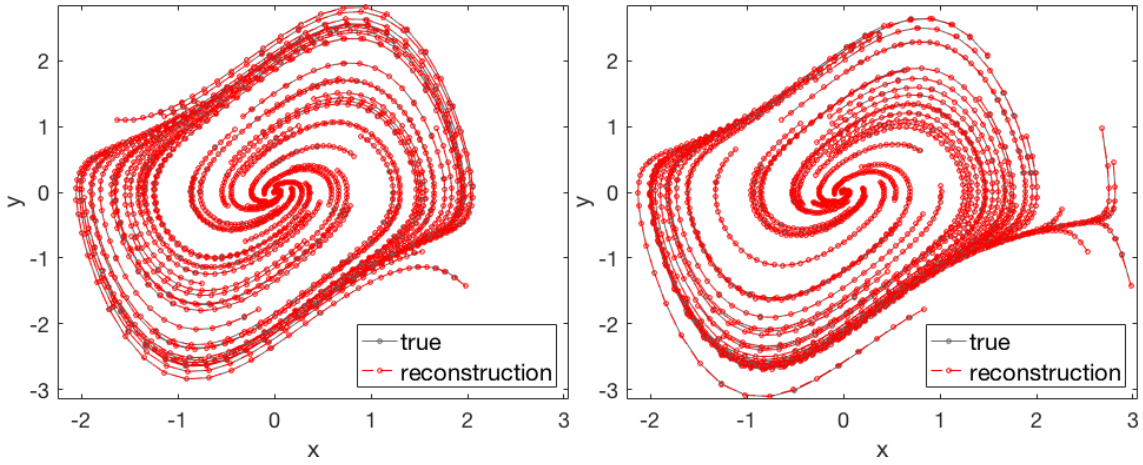


Figure 31: Validation data and corresponding neural net reconstructions by a plain neural net which approximates the forward "step" flow map of the Van der Pol oscillator (31). Left: $x_0, y_0 \in [-2, 2] \times [-2, 2]$ from the model trained in trial 1 of Table 26, for which there were 42/50 successful validation integrations. Right: $x_0, y_0 \in [-1, 3] \times [-2, 2]$ from the model trained in trial 1 of Table 27, for which there were (coincidentally also) 42/50 successful validation integrations.

where $p(x_3, x_4, x_6|x_5, t)$ is the conditional probability density of $x_3(t), x_4(t)$, and $x_6(t)$ given $x_1(t)$ [2, 21]. A substitution of (51) into (50) yields

$$\frac{\partial p(x_5, t)}{\partial t} = \frac{\partial}{\partial x_5}\left[\Big(x_5 - F + \mathbb{E}\big\{(x_3 - x_6)x_4|x_5, t\big\}\Big)p(x_5, t)\right], \tag{52}$$

where

$$\mathbb{E}\left\{(x_3 - x_6)x_4|x_5\right\} = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}(x_3 - x_6)x_4 p(x_3, x_4, x_6|x_5, t)dx_3dx_4dx_6 \tag{53}$$

is the conditional expectation of the random variable $(x_3(t) - x_6(t))x_4(t)$ given a realization of the random variable $x_5(t)$. Note that both random variables are evaluated at the same time t.

Example 2.2 (Divergence-free system) Consider the familiar example we introduced in section 2.2.1, and hereafter rewritten for convenience

$$\begin{cases} \dot{x} = 2xy - 1, \\ \dot{y} = -x^2 - y^2 + \mu. \end{cases} \tag{54}$$

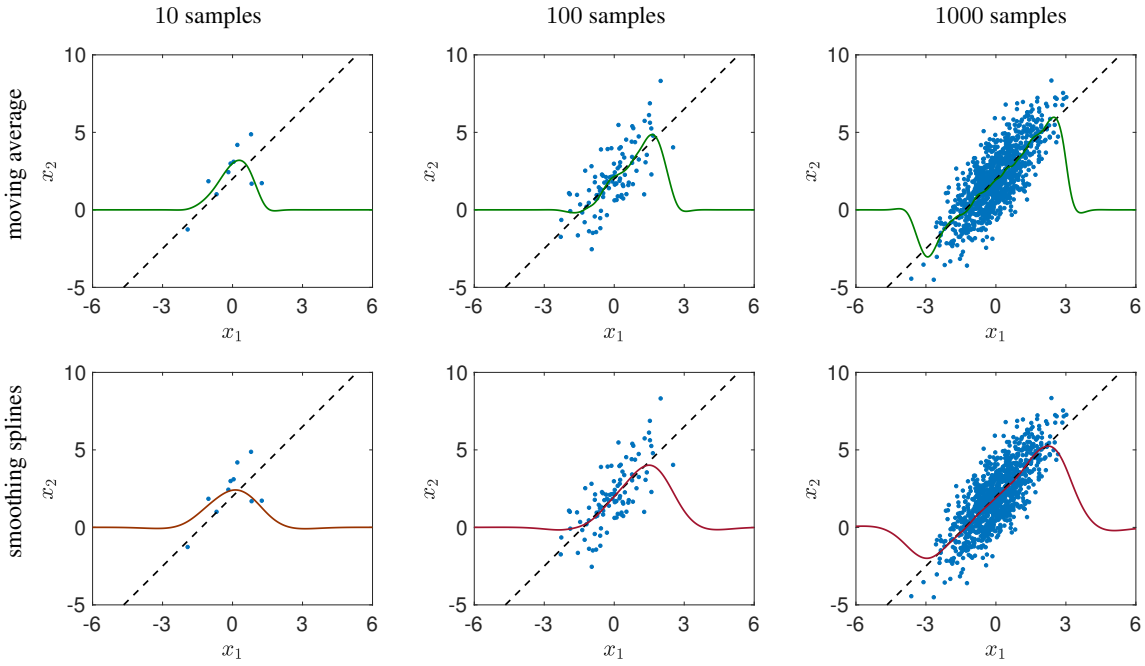


Figure 32: Numerical estimation of the conditional expectation (63) for different number of samples of (61). Shown are results obtained with moving averages (first row) and cubic smoothing splines (second row). It is seen that both methods converge to the correct conditional expectation in the active region as we increase the number of samples.

The Liouville equation associated with (54) is

$$\frac{\partial p(x,y,t)}{\partial t} = -\frac{\partial}{\partial x}\big((2xy-1)p(x,y,t)\big) - \frac{\partial}{\partial y}\big((-x^2-y^2+\mu)p(x,y,t)\big). \tag{55}$$

As before, we set $\mu = 5$. We chose the initial PDF $p_0(x,y)$ to be the product of two independent Gaussians with means $\mu_x = \mu_y = 0.75$ and variances $\sigma_x^2 = \sigma_y^2 = 0.25$. Suppose we are interested in the PDF of the first component of the system, i.e., $x(t)$. To obtain a reduced-order PDF equation for $p(x,t)$, integrate (55) with respect to y. This yields,

$$\frac{\partial p(x,t)}{\partial t} = -\frac{\partial}{\partial x}\left(2x\int_{-\infty}^{\infty} yp(x,y,t)dy - p(x,t)\right). \tag{56}$$

We can rewrite the joint PDF of x and y at time t as $p(x,y,t) = p(x,t)p(y|x,t)$. This allows us to express the unclosed term in (56) as a conditional expectation

$$\int_{-\infty}^{\infty} yp(x,y,t)dy = p(x,t)\int_{-\infty}^{\infty} yp(y|x,t)dy = p(x,t)\mathbb{E}\{y(t)|x(t)\}. \tag{57}$$

Equation (56) can now be rewritten in terms of $\mathbb{E}\{y|x,t\}$

$$\frac{\partial p(x,t)}{\partial t} = -\frac{\partial}{\partial x}\left(2xp(x,t)\mathbb{E}\{y(t)|x(t)\} - p(x,t)\right). \tag{58}$$

The evolution equation for $p(x,t)$ is unclosed because it contains terms that depend on both x and y. In order to remedy this problem, approximation of the unclosed term is necessary. This can be performed using regression methods (as in section 2.3) or neural networks (as in Section 2.3.2).

Remark 2.6 More generally, if we are interested in the PDF of k-th component of the system (11), then we need to express the right hand side of (47) in terms of conditional expectations, and estimate such expectations from data. If $G_k(\boldsymbol{x})$ is in the form of a sum of separable functions, i.e.,

$$G_k(\boldsymbol{x}) = \sum_{l=1}^{r} \prod_{j=1}^{n} f_{kl}^{j}(x_j), \tag{59}$$

then we can explicitly write (47) as

$$\frac{\partial p(x_k,t)}{\partial t} + \frac{\partial}{\partial x_k}\left(p(x_k,t) \sum_{l=1}^{r} f_{kl}^{k}(x_k) \mathbb{E}\left\{ f_{kl}^{1}(x_1)...f_{kl}^{k-1}(x_{k-1}) f_{kl}^{k+1}(x_{k+1})...f_{kl}^{n}(x_n) \Big| x_k \right\} \right) = 0. \tag{60}$$

Computing conditional expectations from data or sample trajectories is a key step in determining accurate closure approximations of reduced-order PDF equations. A major challenge to fitting a conditional expectation is ensuring accuracy and stability. More importantly, the estimator must be flexible and effective for a wide range of numerical applications.

2.3.1 Estimating conditional expectations from data: smoothing splines and moving averages

In this section we present two different approaches to estimate conditional expectations from data based on moving averages and smoothing splines. The moving average estimate is obtained by first sorting the data into bins and then computing the average within each bin. With such averages available, we can construct a smooth interpolant using the average value within each bin. Some factors that affect the bin average approximation are the bin size (the number of samples in each bin) and the interpolation method used in the final step. Another approach to estimate conditional expectations uses smoothing splines. This approach seeks to minimize a penalized sum of squares. A smoothing parameter determines the balance between smoothness and goodness-of-fit in the least-squares sense [6]. The choice of smoothing parameter is critical to the accuracy of the results. Specifying the smoothing parameter a priori is generally yields poor estimates [22]. Instead, cross-validation and maximum likelihood estimators can guide the choice the optimal smoothing value for the data set [26]. Such methods can be computationally intensive, especially when the spline estimate is performed at each time step. Other techniques to compute conditional expectations can leverage on recent developments on deep learning [10]. In Figure 32 we compare the performance of the moving average and smoothing splines approaches in approximating the conditional expectation of two jointly Gaussian random variables. Specifically, we consider the joint distribution

$$p(x_1,x_2) = \frac{1}{2\pi\sigma_1\sigma_2\sqrt{1-\rho^2}} \exp\left(-\frac{1}{2(1-\rho^2)} \left[\frac{(x_1-\mu_1)}{\sigma_1^2}\frac{(x_2-\mu_2)}{\sigma_2^2} - \frac{2\rho(x_1-\mu_1)(x_2-\mu_2)}{\sigma_1\sigma_2)} \right] \right) \tag{61}$$

with parameters $\rho = 3/4$, $\mu_1 = 0$, $\mu_2 = 2$, $\sigma_1 = 1$ $\sigma_2 = 2$. As is well known [21], the conditional expectation of x_2 given x_1 can be expressed as[4]

$$\mathbb{E}\{x_2|x_1\} = \mu_2 + \rho\frac{\sigma_2}{\sigma_1}(x_1-\mu_1) = 2 + \frac{3}{2}x_1. \tag{63}$$

[4]Given two random variables with joint PDF $p(x_1,x_2)$, the conditional expectation of x_2 given x_1 is defined as

$$\mathbb{E}\{x_2|x_1\} = \int_{-\infty}^{\infty} x_2 p(x_2|x_1)dx_2 = \frac{1}{p(x_1)}\int_{-\infty}^{\infty} x_2 p(x_1,x_2)dx_2, \tag{62}$$

where $p(x_1)$ is the marginal of $p(x_1,x_2)$ with respect to x_2.

Such conditional expectation is plotted in Figure 32 (dashed line), together with the plots of the conditional average estimates we obtain with the moving average and the smoothing spline approaches for different numbers of samples. It is seen that both methods converge to the correct conditional expectation as we increase the number of samples. Note, however, that convergence is achieved in regions where the PDF (61) is not small (see the subsequent Remark 2.8). Both estimators require setting suitable parameters to compute expectations, e.g., the width of the moving average window in the moving average approach, or the smoothing parameter in the cubic spline approximant.

Remark 2.7 If the joint PDF of x_1 and x_2 is not compactly supported, then the conditional expectation is defined on the whole real line. It is computationally challenging to estimate the expectation (63) in regions where the joint PDF is very small [4]. At the same time, if we are not interested in rare events (i.e., tails of probability densities), then resolving the dynamics in such regions of such small probability is not needed. This means that if we have available a sufficient number of sample trajectories, we can identify the active regions where the dynamics are happening with high probability and approximate the conditional expectation only within such regions [5]. Outside the active regions, we set the expectation equal to zero.

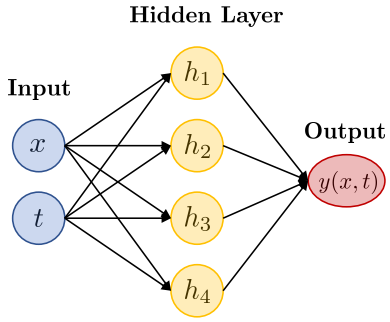
Remark 2.8 If the joint PDF of x_1 and x_2 is compactly supported, e.g. uniform in the square $[0,1]^2$, then the conditional expectation is undefined outside the support of the joint PDF. This means, in principle, that we are not allowed to set any value for the conditional expectation outside the domain where it exists. However, a quick look at the structure of the reduced-order PDF equations we are considering, e.g., equation (60), suggests that the conditional expectation plays the role of a velocity field advecting the reduced-order PDF. Therefore, setting such velocity vector equal to zero in the regions where the reduced order PDF is very small or even undefined, does not affect the PDF propagation process. On the other hand, setting the conditional expectation equal to zero in low- or zero-probability regions greatly simplifies the mathematical discretization of PDEs in the form (60).

2.3.2 Neural network estimation of conditional expectations

In this section we describe a specific class of feed-forward neural networks, i.e., *radial basis networks*, to effectively estimate conditional expectations. Radial basis neural networks have a fixed three-layer architecture consisting of an input layer, a hidden layer, and an output layer. The relationship between each layer is depicted in Figure 33.

1. The **input layer** has as many neurons as there are inputs. For example, if the neural net is used to approximate the conditional expectation $\mathbb{E}\{y(t)|x(t)\}$ then it takes two inputs: t and $x(t)$. Therefore, the input layer for this problem consists of two neurons. This layer is also responsible for scaling the input vector $\boldsymbol{x}(t)$ using input weights.

2. The **hidden layer** has a variable number of neurons, say M, each with a center $\boldsymbol{c}_m$ for $m = 1, \ldots, M$. Generally, we choose the number of neurons, M, to be significantly smaller than the number of training samples used. Each neuron in the hidden layer represents a radial basis function, e.g., a Gaussian
$$h_m(\boldsymbol{x}, t) = \exp\left(-\frac{\|\boldsymbol{x}(t) - \boldsymbol{c}_m(t)\|^2}{2b^2}\right), \tag{64}$$
where b is the *spread* (or variance) of the basis function.



$$h_j(x,t) = \exp\left[-\frac{(x-c_j(t))^2}{2b^2}\right]$$

$$\mathbb{E}\{x(t)|y(t)\} = \sum_{j=1}^{M} w_j h_j(x,t)$$

Figure 33: Architecture of a radial basis neural network that receives two inputs (i.e., x and t) and returns one output $y(x,t)$, i.e., the conditional expectation $\mathbb{E}[x(t)|y(t)]$.

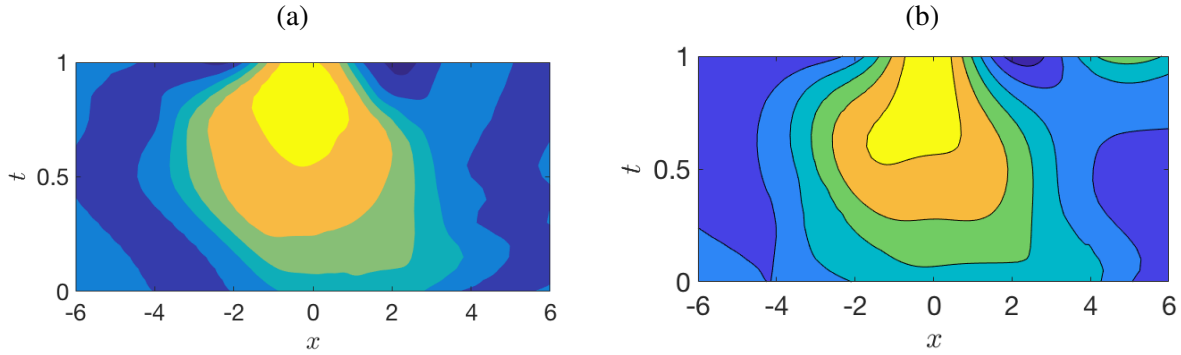


Figure 34: Dynamical system (54). (a) An approximation of $\mathbb{E}\{y(t)|x(t)\}$ obtained via methods described in Section 2.3.1 using 5000 sample trajectories. (b) An approximation of $\mathbb{E}\{y(t)|x(t)\}$ obtained using a radial basis neural network with 25 neurons in the hidden layer and 500 training samples. The radial basis network is able to recover the conditional expectation with only 500 samples, whereas the regression method (a) requires upwards of 5000 samples for the same level of accuracy.

3. The **output layer** performs a linear mapping from the hidden space to the output space. For example, if we are interested in representing the conditional expectation of $y(t)$ given $x(t)$ (see equation (58)) then the output layer takes the form

$$\mathbb{E}\{y(t)|x(t)\} = \sum_{m=1}^{M} w_m h_m(x,t).$$

The spread of the radial basis neural network, i.e., the parameter b in (64), acts as a smoothing parameter. A large spread leads to a smooth function approximation, while a small spread allows for more variation between neurons. For illustration purposes, let we consider the system (54) with random initial condition distributed as described in Example 2.2. Let us set the spread of our radial basis neural net as $b = 1.8$. In Figure 34, we compare the conditional expectation approximation generated via the methods presented in sections 2.3.1 and 2.3.2 respectively. In most cases, radial basis neural networks require more many neurons than a comparable feed-forward network or another regression method. However, the trade off is that radial basis networks often take less time and fewer samples to train. This is demonstrated in Figure 34, were show that our radial basis neural network with 25 neurons requires only 500 sample trajectories to approximate $\mathbb{E}\{y(t)|x(t)\}$ in (56), while smoothing splines and moving average regression methods require

		Neurons per layer			
		10	25	50	100
No. Samples	10	1 s	1 s	1 s	1 s
	50	1 s	2 s	3 s	6 s
	100	2 s	4 s	7 s	14 s
	500	26 s	55 s	108 s	255 s
	750	70 s	143 s	264 s	710 s

Table 28: Training time of a neural net for approximating the unclosed term in (56), depending on the availability of data.

thousands more samples to achieve the same level of accuracy. The radial basis neural network requires at least one neuron for each pattern that appears in the training data. For large or unsmooth data sets (i.e. data that is spiky with steep gradients), radial basis neural networks can become costly to train. This is clearly demonstrated in Table 28. For this reason, the radial basis networks are appropriate for computing closure approximations if and only if the conditional expectation being approximated is relatively smooth, which is often the case.

Training the neural net To initialize training of the neural net, we need sample trajectories, e.g., of the system (54). Start by drawing P samples of the initial condition from the initial pdf, $p_0(x, y)$. Next, we evolve each initial condition samples forward in time using a one-step method such as the Runge-Kutta scheme with time step Δt. This yields $N = P/\Delta t$ data points to feed into the neural network for training. We feed N input vectors to the network along with N target values for y. For the i^{th} sample in the training set, $\boldsymbol{x}_i = (x_i, t_i)$ are the input values and y_i is the target output. The algorithm iteratively creates a radial basis network one neuron at a time. Neurons are added to the network until an error tolerance is met or a user-defined number of neurons has been reached. We start with 0 neurons. The network is simulated, error targets are checked. If the error target is met, the algorithm stops, otherwise new neurons are added iteratively. Once each neuron in the network has been assigned input weights we need to adjust output weights to minimize error. Given M centers, we define the transfer matrix $\boldsymbol{\Phi}^C$ as

$$\boldsymbol{\Phi}^C_{ij} = \exp\left(-\frac{||\boldsymbol{x}_i - \boldsymbol{c}_j||_2^2}{2b^2}\right).$$

The transfer matrix satisfies

$$\boldsymbol{\Phi}^C \boldsymbol{w} + \boldsymbol{\beta} = \boldsymbol{y}, \tag{65}$$

where $\boldsymbol{y}$ is the target vector, $\boldsymbol{\beta}$ is the bias vector (to be found), and $\boldsymbol{w}$ is the weight vector (to be found). Solving the least squares problem 65 yields the desired output weights and biases. We seek a global optimal approximation to the training data in the minimum mean square error (MSE) sense.

$$MSE = \frac{1}{N}\sum_{i=1}^{N}\left\| y_i - \sum_{m=1}^{M} w_m h_m(x_i, t_i)\right\|^2.$$

The mean squared error of the network approximation is checked, and the algorithm will exit if the error target is met. Otherwise the next neuron is added. This process is repeated until the error target is met or the maximum number of neurons is reached.

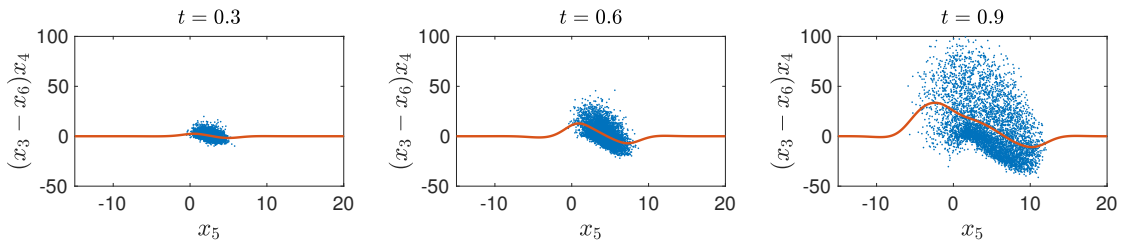


Figure 35: Data-driven smoothing spline estimation of the conditional expectations arising in the study of the Lorenz-96 dynamical system (48).

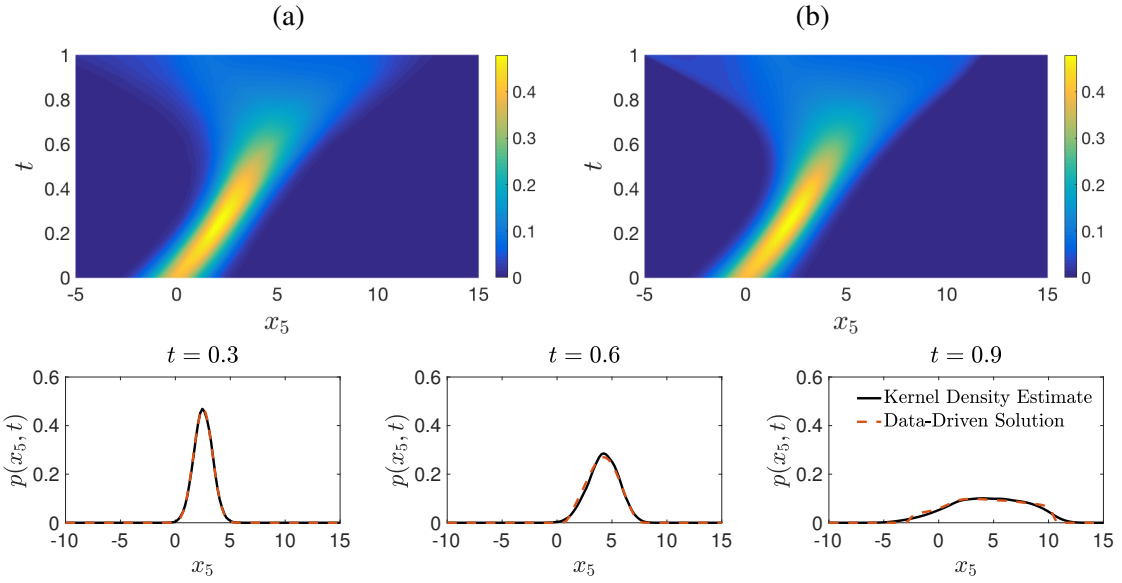


Figure 36: Lorenz-96 dynamical system. (a) Accurate kernel density estimate of $p_5(x_5, t)$ based on 25000 sample trajectories. (b) Numerical solution of (50) obtained by estimating $\mathbb{E}\left[\big(x_3(t) - x_6(t)\big)x_4(t)|x_5(t)\right]$ with 5000 sample trajectories.

2.3.3 Numerical results

Lorenz-96 system In Figure 35, we summarize the results we obtained by applying the smoothing spline conditional expectation estimator to the Lorenz-96 model introduced in (48). This system has polynomial-type nonlinearities. The quantity of interest, x_5, is indicated in the x-axis of the plots. When using this approach, we must be careful to provide enough samples for the estimator to adequately capture the support of the underlying PDF. If we do not have enough samples, the estimator will not be consistent with the true conditional expectation. In Figure 36 we plot the PDF dynamics we obtain by solving (50) with an accurate Fourier spectral method. The conditional expectation is estimated based on 5000 sample trajectories.

Divergence-free system (54) In Figure 37, we plot the PDF dynamics we obtained by sampling (54), and then solving (58) with a Fourier spectral method. The conditional expectation here is estimated based on the moving average regression method detailed in section 2.3.1 and 5000 sample trajectories. Similarly, in Figure 38, we plot the PDF dynamics we obtained by solving (58) with the conditional expectation estimated using a radial basis neural network and $M = 25$ neurons in the hidden layer. The network was trained with

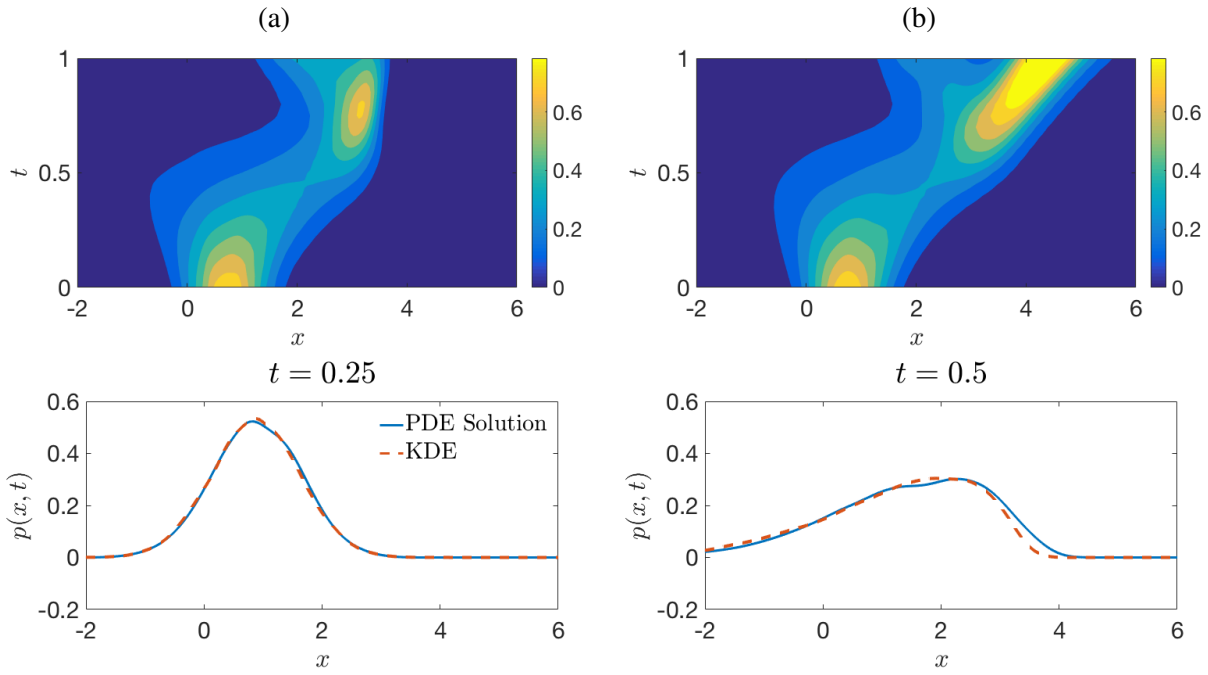


Figure 37: Divergence-free dynamical system (54). (a) Accurate kernel density estimate of $p(x,t)$ based on 30000 sample trajectories. (b) Numerical solution of (58) obtained by estimating $\mathbb{E}\{y(t)|x(t)\}$ using the moving average regression method, as outlined in section 2.3.1.

500 sample trajectories, i.e., *an order of magnitude less than the moving average case*.
In Table 29 we summarize L^2 errors between the benchmark reduced order PDF $p(x,t)$ at $t = 0.5$ and the PDF we obtained from the reduced order equation (58), with the conditional expectation computed by using the radial basis neural network. Specifically, we show results as a function of the number of neurons of the hidden layer and the number of samples used to train the neural network approximating $\mathbb{E}\{y(t)|x(t)\}$.

2.4 Applications of data-driven optimal control strategies

As we anticipated in the quarter-by-quarter breakdown of the proposed research activities, in Q1 we studied several nonlinear models involving swarms of attacking/defending drones, as well as disease propagation models on random networks of interacting individuals. The purpose of such study was to identify two interesting systems which we could build upon to test the proposed data-driven optimal control architecture. Hereafter, we describe such systems in detail.

2.4.1 Swarm of attacking/defending agents

In this section we describe a dynamical model for planning the motion of a swarm of autonomous vehicles (defenders) to protect a High Value Unit (HVU) which is under attack by a swarm of autonomous agents (attackers). The core of the model was developed in a recent paper [27] on modeling combat situations with multiple heterogeneous agents and parameter uncertainties. The goal is to optimally protect a HVU from multiple incoming attackers. The HVU can be a stationary unit, such as a base or population center, or a moving unit such as an aircraft carrier. The attackers are unmanned vehicles following automatic target tracking towards the HVU while performing basic obstacle avoidance around defenders. It is assumed that attacker fire is directed entirely towards the HVU with no fire to spare for self-defense. A swarm of

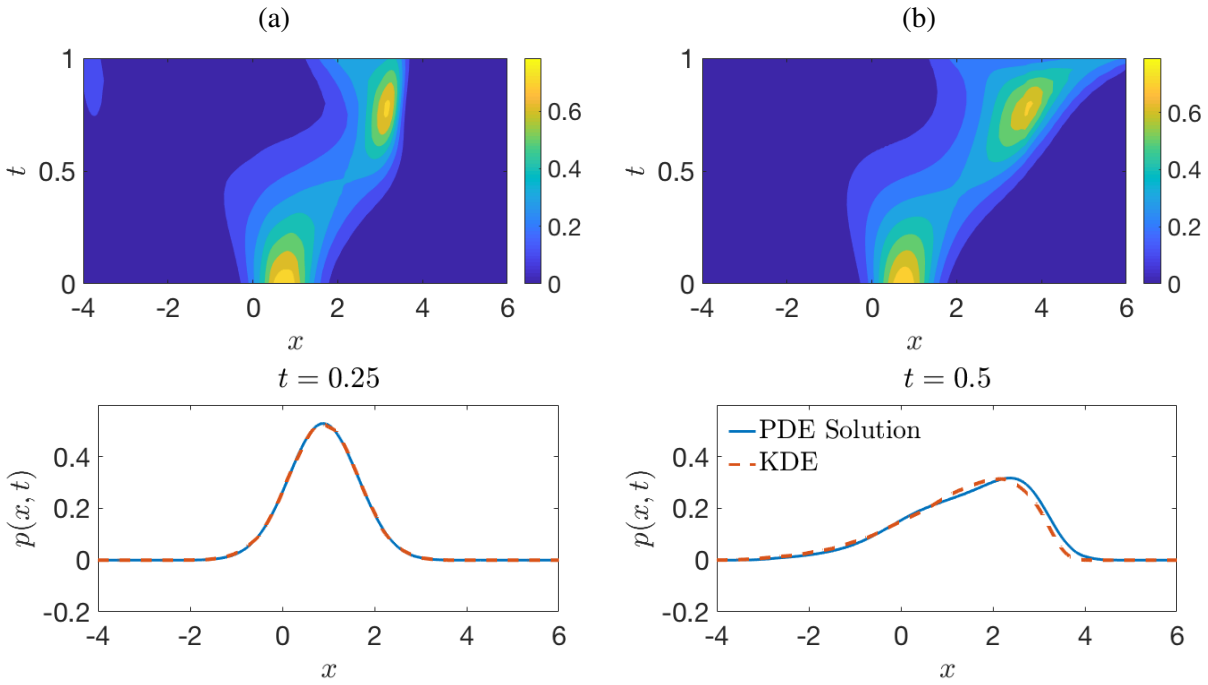


Figure 38: Divergence-free dynamical system (54). (a) Accurate kernel density estimate of $p(x,t)$ based on 30000 sample trajectories. (b) Numerical solution of (56) obtained by estimating $\mathbb{E}[y|x,t]$ using a radial basis neural network trained with 500 sample trajectories.

defenders is dispatched to protect the HVU at time $t = 0$. The single-minded focus of the attacker means that the survival of the defenders is not in danger, however the goal is to minimize the probability that the HVU is destroyed. Figure 39 diagrams the attrition interactions among HVU, attackers and defenders.

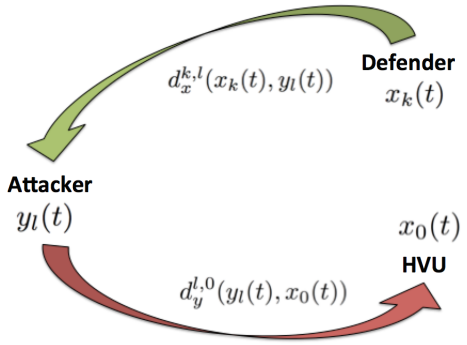


Figure 39: Interception of HVU-focused attack. $d_x^{k,l}(\boldsymbol{x}_k(t), \boldsymbol{y}_l(t))$ is the damage rate of the k-th defender against the l-th attacker; $d_y^{l,0}(\boldsymbol{y}_l(t), \boldsymbol{x}_0(t))$ is the damage rate of the l-th attacker against the HVU.

Attacker and defender dynamics Each of the K defenders are modeled as Dubin's vehicle with dynamics

$$\dot{\boldsymbol{x}}_k(t) = \begin{pmatrix} \dot{x}_{k,1}(t) \\ \dot{x}_{k,2}(t) \\ \dot{x}_{k,3}(t) \end{pmatrix} = \begin{pmatrix} v_D \sin(x_{k,3}(t)) \\ v_D \cos(x_{k,3}(t)) \\ u_k(t) \end{pmatrix} = \boldsymbol{f}_k(\boldsymbol{x}_k(t), \boldsymbol{u}_k(t)), \qquad k = 1, \ldots, K \tag{66}$$

		Neurons per layer			
		10	25	50	100
No. Samples	10	7.81 e–02	3.84 e–02	4.00 e–02	3.33 e–01
	50	1.66 e–02	1.54 e–02	2.28 e–02	5.36 e–02
	100	1.66 e–02	2.07 e–03	3.04 e–03	4.13 e–02
	500	1.35 e–01	4.01 e–02	6.4 e–02	4.43 e–02
	1000	4.19 e–02	4.94 e–02	4.77 e–02	4.35 e–02

Table 29: Divergence-free dynamical system (54). L^2 errors between the benchmark reduced order PDF $p(x,t)$ at $t = 0.5$ and the one obtained from the reduced order equation (58), with $\mathbb{E}\{y(t)|x(t)\}$ estimated using radial basis neural networks. Specifically, we plot the error versus the number of neurons of the hidden layer and the number of samples used to train the neural network.

where v_D is the modulus of the defender velocity. The control $u_k(t) \in \mathbb{R}$ is subject to the constraint $u_{min} \leq u_k(t) \leq u_{max}$. Each of the L attackers have deterministic dynamics but with uncertainty stemming from unknown initial locations and unknown velocities. We group all uncertain parameters in the attacking swarm into a vector $\boldsymbol{\omega}$, and assume that it has a known joint distribution $\phi(\boldsymbol{\omega})$. Attackers move with uncertain but constant velocity and a heading determined by a weighted combination of their desire to evade each defender but also to track the HVU. These dynamics are expressed as:

$$\dot{\boldsymbol{y}}_l = \begin{pmatrix} \dot{y}_{l,1}(t,\boldsymbol{\omega}) \\ \dot{y}_{l,2}(t,\boldsymbol{\omega}) \end{pmatrix} = v_A \frac{\boldsymbol{H}^l(\boldsymbol{R}^l(t,\boldsymbol{\omega}))}{\|\boldsymbol{H}^l(\boldsymbol{R}^l(t,\boldsymbol{\omega}))\|}, \tag{67}$$

where $\boldsymbol{H}^l$ is a two-dimensional heading vector

$$\boldsymbol{H}^l(\boldsymbol{R}^l(t,\boldsymbol{\omega})) = \begin{pmatrix} H_1^l(\boldsymbol{R}^l(t,\boldsymbol{\omega})) \\ H_2^l(\boldsymbol{R}^l(t,\boldsymbol{\omega})) \end{pmatrix}, \tag{68}$$

and $\boldsymbol{R}^l$ is the vector of relative positions between attacker $\boldsymbol{y}_l(t,\boldsymbol{\omega})$ and the defenders at $\boldsymbol{x}_k$, $k = 1,\ldots,K$ defined as

$$\boldsymbol{R}^l(t,\boldsymbol{\omega}) = \left(\boldsymbol{R}^{l,0}(t,\boldsymbol{\omega}), \boldsymbol{R}^{l,1}(t,\boldsymbol{\omega}), \ldots, \boldsymbol{R}^{l,K}(t,\boldsymbol{\omega})\right), \quad \boldsymbol{R}^{l,k}(t,\boldsymbol{\omega}) = \begin{pmatrix} y_{l,1}(t,\boldsymbol{\omega}) - x_{k,1}(t) \\ y_{l,2}(t,\boldsymbol{\omega}) - x_{k,2}(t) \end{pmatrix}. \tag{69}$$

The heading vector for each attacker is determined by averaging the attacker's conflicting impulses. On the one hand, the attacker is endeavoring to track the HVU, and it is being herded away through avoidance of the defenders. This averaging is given by the sum

$$\boldsymbol{H}^l(\boldsymbol{R}^l) = \sum_{k=0}^{K} \alpha^{l,k}\left(\|\boldsymbol{R}^{l,k}\|\right) \frac{\boldsymbol{R}^{l,k}}{\|\boldsymbol{R}^{l,k}\|} \tag{70}$$

where $\alpha^{l,k}\left(\|\boldsymbol{R}^{l,k}\|\right)$ weights the influence of each agent based on radial distance. When $k = 0$, $\alpha^{l,k}\left(\|\boldsymbol{R}^{l,k}\|\right)$ returns a negative value attracting the attacker to the HVU, and when $k = 1,\ldots,K$, $\alpha^{l,k}\left(\|\boldsymbol{R}^{l,k}\|\right)$ returns a positive value repelling the attacker from the defenders. These values are set, for $k = 1,\ldots,K$, as

$$\begin{cases} \alpha^{l,0}\left(\|\boldsymbol{R}^{l,0}\|\right) = -\frac{1}{\sigma_0} e^{\frac{-\|\boldsymbol{R}^{l,0}\|}{\sigma_0}} \\ \alpha^{l,k}\left(\|\boldsymbol{R}^{l,k}\|\right) = \frac{1}{\sigma_k} e^{\frac{-\|R^{l,k}\|}{\sigma_k}}. \end{cases} \tag{71}$$

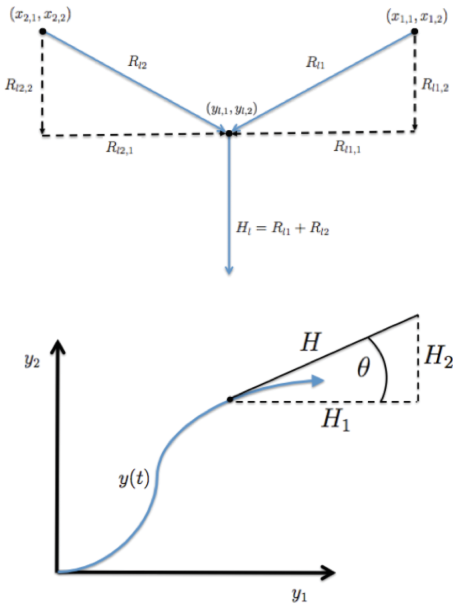


Figure 40: Averaged herding heading example for one attacker and two defenders

System attrition The damage rate of the k-th defender against the l-th attacker is defined by

$$d_x^{k,l}(\boldsymbol{x}_k(t), \boldsymbol{y}_l(t,\boldsymbol{\omega})) = \lambda_D \left[1 + \frac{\left[\theta^{k,l}\right]^2}{3}\right]^{-3/2} \cdot \Phi\left(\frac{-r^{k,l}(t)}{\sigma_D}\right) \tag{72}$$

where $r^{k,l}(t)$ is the distance between the k-th defender and the l-th attacker and Φ is the cumulative Normal distribution function. The damage rate of the l-th attacker against the HVU is defined by

$$d_y^{l,0}(\boldsymbol{y}_l(t,\boldsymbol{\omega}), \boldsymbol{x}_0) = \lambda_A \Phi\left(\frac{-r^{l,0}(t)}{\sigma_A}\right), \tag{73}$$

where, as with the defenders $r^{l,0}(t)$, is the distance between the l-th attacker and the HVU while λ_A and σ_A are constants that calibrate intensity and range. Based on damage rates, it can be shown [27] that the probability of survival of the l-th attacker satisfies differential equation

$$\frac{dQ_l(t,\boldsymbol{\omega})}{dt} = -Q_l(t,\boldsymbol{\omega}) \sum_{k=1}^{K} d_x^{k,l}(\boldsymbol{x}_k(t), \boldsymbol{y}_l(t,\boldsymbol{\omega})) \tag{74}$$

with initial condition $Q_l(0,\boldsymbol{\omega}) = 1$, while the probability of survival of the HVU is obtained by solving

$$\frac{dP_0(t,\boldsymbol{\omega})}{dt} = -P_0(t,\boldsymbol{\omega}) \sum_{l=1}^{L} Q_l(t,\boldsymbol{\omega}) d_y^{l,0}(\boldsymbol{y}_l(t,\boldsymbol{\omega}), \boldsymbol{x}_0), \tag{75}$$

with $P_0(0,\boldsymbol{\omega}) = 1$. The following control problem yields *optimal defenders' paths, maximizing the probability of survival of the HVU*.

HVU attack/interception problem For K defenders and L attackers with uncertainty prescribed by the joint probability density function ϕ, determine the control $\boldsymbol{u} : [0, t_f] \to U \in \mathbb{R}^K$ that maximizes the

Variable	Description
S	Proportion of population that is healthy and susceptible to HIV
X	Proportion of population infected with HIV
R	Proportion of population that is protected from HIV
s_i	Probability that vertex i is susceptible
x_i	Probability that vertex i is infected
r_i	Probability that vertex i is protected
Parameter	Description
β	Probability that transmission will occur between two connected individuals
γ	Probability that an infected individual will recover
p	Occupation probability
p_k	Degree distribution
K	Mean degree

Table 30: Definition of all phase variables and parameters appearing in the nonlinear system (78).

probability of survival, i.e., minimizes

$$J = \int_{\Omega} \left[1 - P_0(t_f, \boldsymbol{\omega})\right] \phi(\boldsymbol{\omega}) d\boldsymbol{\omega}$$

subject to

$$\begin{cases} \dot{\boldsymbol{x}}_k(t) = \boldsymbol{f}_k(\boldsymbol{x}_k(t), u_k(t)), & \boldsymbol{x}_k(0) = \boldsymbol{x}_{k0}, \\ \dot{\boldsymbol{y}}_l(t, \boldsymbol{\omega}) = \boldsymbol{g}_l(\boldsymbol{x}_1(t), \ldots, \boldsymbol{x}_K(t), \boldsymbol{y}_l(t, \boldsymbol{\omega})), & \boldsymbol{y}_l(0, \boldsymbol{\omega}) = \boldsymbol{\gamma}_l(\boldsymbol{\omega}), \\ \dot{Q}_l(t, \boldsymbol{\omega}) = -Q_l(t, \boldsymbol{\omega}) \displaystyle\sum_{k=1}^{K} d_x^{k,l}(\boldsymbol{x}_k(t), \boldsymbol{y}_l(t, \boldsymbol{\omega})), & Q_l(0, \boldsymbol{\omega}) = 1, \\ \dot{P}_0(t, \boldsymbol{\omega}) = -P_0(t, \boldsymbol{\omega}) \displaystyle\sum_{l=1}^{L} Q_l(t, \boldsymbol{\omega}) d_y^{l,0}(\boldsymbol{y}_l(t, \boldsymbol{\omega}), \boldsymbol{x}_0), & P_0(0, \boldsymbol{\omega}) = 1, \end{cases}$$

and additional geometric constraints. Here, $l = 1, \ldots, L$ (attackers) and $k = 1 \ldots, K$ (defenders).

The HUV attach interception problem is ideal for testing the proposed data-driven optimal control architecture. In particular, in [27] we were able to generate numerical solutions (in low dimensions) by using the collocation method. However, our experience shows that as we include more realistic sources of uncertainties, existing state-of-the-art computational control methods fail to produce optimal solutions. The proposed new schemes can open a new path for solving this type of optimal control problems.

2.4.2 Disease propagation on random networks

We will consider a model to describe and predict HIV/AIDS transmission through a random network of individuals. The objective is to *minimize the probability*, x_i, that a given individual in the network will be infected with HIV. The network has n nodes (vertices), representing n indivinuals. Edges between vertices represent interactions between individuals. The degree of a vertex is the number of edges connected to it. The degree distribution p_k of the graph is an ordered list of vertex degrees. All these quantities are defined in Table 30. Each individual in the network falls into one of three categories: *susceptible*, *infected*, *removed*.

The probability that the vertex i belongs to one of these three categories is governed by the equations

$$\frac{ds_i}{dt} = -s_i \sum_{j=1}^{n} A_{ij} B_{ij} x_j, \tag{76}$$

$$\frac{dx_i}{dt} = s_i \sum_{j=1}^{n} A_{ij} B_{ij} x_j - \gamma x_i, \tag{77}$$

$$\frac{dr_i}{dt} = \gamma x_i, \tag{78}$$

where $s_i + x_i + r_i = 1$, and $i = 1, \ldots, n$. The matrices A and B are known as the adjacency matrix and the transmission matrix respectively. They are both set to be *random*. Notice that the evolution of each component s_i depends on all components of $\boldsymbol{x}$, i.e., the system is fully coupled. Similarly, the evolution of x_i and r_i depend on all components of $\boldsymbol{x}$ or $\boldsymbol{s}$. This makes the system (78) too complex to solve exactly in its full form. Instead, we focus on some vertex (or subset of vertices) of interest and approximate the unclosed terms using the methods we outlined in section 2.3.

The components of the adjacency matrix are determined by the underlying network model. The structure of the network has a large impact on the dynamics of disease propogation. For the purposes of this research, we will employ a configuration network model with a given degree distribution, p_k. Hereafter we will describe several common choices for the degree distribution for a social network.

- **Poisson degree distribution** The Erdös-Rényi model, otherwise known as a Poisson random graph, is a network model in which we fix the number of vertices and the occupation probability. Edges between vertices are present with probability p and absent with probability $1-p$. The degree distribution for this graph is

  $$p_k = \binom{n-1}{k} p^k (1-p)^{n-1-k}$$

  where n is the number of vertices on the graph and p_k is the probability that a given vertex is connected to exactly k other vertices.

- **Watts-Strogatz graph** The Watts-Strogatz model, otherwise known as the small world model, is an undirected graph with n vertices and $nK/2$ edges, where K is the mean degree. Edges between vertices are present with probability p and absent with probability $1-p$. The degree distribution for this graph is

  $$p_k = \sum_{n=0}^{\min(k-K/2, K/2)} \binom{K/2}{n} (1-\beta)^n \beta^{K/2-n} \frac{(\beta K/2)^{k-K/2-n}}{(k-K/2-n)!} e^{-\beta K/2},$$

  where p_k is the probability that a given vertex on the graph is connected to exactly k other vertices.

- **Exponential degree distribution** Exponential random graphs are a popular choice due to their ability to represent complex structural tendencies that commonly arise in social networks. The degree distribution for this graph is

  $$p_k = (1 - e^{-\lambda}) e^{-\lambda k},$$

  where $\lambda > 0$ is the exponential parameter and p_k is the probability that a given vertex is connected to exactly k other vertices.

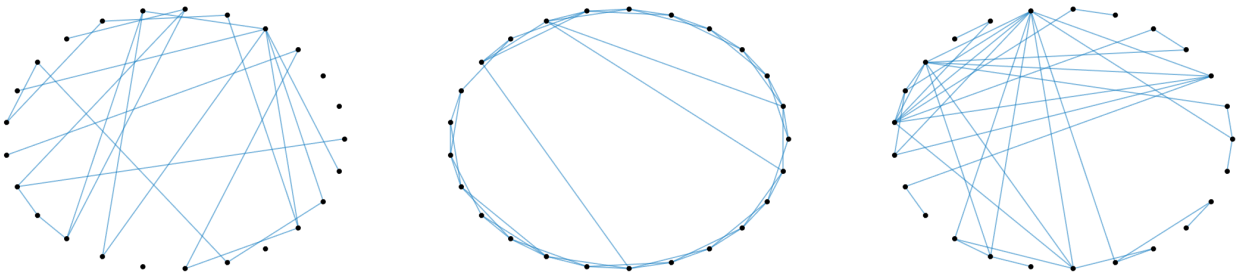


Figure 41: Sample configurations of a Poisson random graph (left) a Watts-Strogatz random graph (center) and an exponential random graph (right) with $N = 25$ vertices. The Poisson and Watts-Strogatz random graphs were generated with occupation probability $p = 0.15$. The exponential random graph was generated with exponential parameter $\lambda = 4$.

The components of the transmission matrix, B, are determined as follows

$$B_{ij} = \begin{cases} 0, & (i \vee j) \text{ practiced safe sex} \vee (i \vee j) \text{ is protected} \\ & \vee\ (i \wedge j) \text{ are susceptible} \vee (i \wedge j) \text{ are infected} \\ 1 - (1 - \beta)^{\eta}, & \text{otherwise} \end{cases} . \tag{79}$$

Each element, B_{ij}, represents the probability of transmission between vertices i and j. Partner selection is a complex process with many social, cultural, and behavioural influences. Social nuances can be encoded in either the adjacency matrix, A, or the transmission matrix B. Characteristics such as partner preference (heterosexual and homosexual), relationship stability, age difference, and social status can contribute to the formation of a partnership. Long-term, stable relationships are characterized by high η while short-term, casual partnerships are characterized by low η. The disease propagation model is initialized with a small number, c, of infected individuals placed randomly amongst the population. The initial conditions for (78) are

$$s_i(0) = 1 - c/n, \qquad x_i(0) = c/n, \qquad r_i(0) = 0. \tag{80}$$

References

[1] M. Abadi, A. Agarwal, and et al. P. Barham.

[2] C. Brennan and D. Venturi. Data-driven closures for stochastic dynamical systems. *J. Comput. Phys.*, 372:281–298, 2018.

[3] R. H. Byrd, P. Lu, J. Nocedal, and C. Zhu. A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific Computing*, 16:1190–1208, September 1995.

[4] G. Casella and R. L. Berger. *Statistical Inference*. Duxbury Press, 2001.

[5] A. J. Chorin and X. Tu. Implicit sampling for particle filters. *PNAS*, 41:17249–17254, 2009.

[6] P. Craven and G. Wahba. Smoothing noisy data with spline functions. *Numerische Mathematik*, 31(4):377–403, 1979.

[7] M. Ehrendorfer. The Liouville equation in atmospheric predictability. In *Seminar on predictability of weather and climate*, pages 47–81, Shinfield Park, Reading, 2003. ECMWF.

[8] L. Grasedyck. Hierarchical singular value decomposition of tensors. *SIAM J. Matrix Anal. & Appl.*, 31(4):2029–2054, January 2010.

[9] L. Grasedyck and C. Löbbert. Distributed hierarchical svd in the hierarchical tucker format. *Numerical Linear Algebra with Applications*, page e2174.

[10] D. Graupe. *Deep learning neural networks: design and case studies*. World Scientific, 2016.

[11] W. Hackbusch. *Tensor Spaces and Numerical Tensor Calculus*. Springer Berlin Heidelberg, 2012.

[12] Ernst Hairer, Syvert P. Nørsett, and Gerhard Wanner. *Solving Ordinary Differential Equations*, volume I - Nonstiff Problems. Springer-Verlag, 2nd edition, 1993.

[13] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python. http://www.scipy.org/, 2001–.

[14] A. I. Khuri. Applications of Dirac's delta function in statistics. *Int. J. Math. Educ. Sci. Technol.*, 35(2):185–195, 2004.

[15] D. Kressner and C. Tobler. Algorithm 941: htucker – a Matlab toolbox for tensors in hierarchical Tucker format. *ACM Transactions on Mathematical Software*, 40(3):1–22, 2014.

[16] Solomon Kullback and Richard Leibler. On information and sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86, 1951.

[17] L. De Lathauwer, B. De Moor, and J. Vandewalle. A multilinear singular value decomposition. *SIAM J. Matrix Anal. & Appl.*, 21(4):1253–1278, 2000.

[18] M. J. Mohlencamp and L. Monzón. Trigonometric identities and sums of separable functions. *Math. Intelligencer*, 27:65–69, 2005.

[19] Ralph Neuneier and Hans Georg Zimmermann. *How to Train Neural Networks*, pages 373–423. Springer Berlin Heidelberg, 1998.

[20] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer-Verlag, 2006.

[21] A. Papoulis. *Probability, random variables and stochastic processes*. McGraw-Hill, third edition, 1991.

[22] S. B. Pope and R. Gadh. Fitting noisy data using cross-validated cubic smoothing splines. *Communications in Statistics - Simulation and Computation*, pages 349–376, 1988.

[23] M. Raissi, P. Perdikaris, and G. E. Karniadakis. Physics informed deep learning (part I): data-driven solutions of nonlinear partial differential equations. *arXiv preprint arXiv:1711.10561 [cs.AI]*, 2017.

[24] D. Venturi. A fully symmetric nonlinear biorthogonal decomposition theory for random fields. *Physica D*, 240(4-5):415–425, 2011.

[25] D. Venturi and G. E. Karniadakis. Convolutionless Nakajima-Zwanzig equations for stochastic analysis in nonlinear dynamical systems. *Proc. R. Soc. A*, 470(2166):1–20, 2014.

[26] G. Wahba. A comparison of gcv and gml for choosing the smoothing parameter in the generalized spline smoothing problem. *Annals of Statistics*, 13(4):1378–1402, 1985.

[27] C. Walton, P. Lambrianides, I. Kaminer, J. Royset, and Q. Gong. Optimal motion planning in rapid-fire combat situations with attacker uncertainty. *Naval Research Logistics (NRL)*, 65:101–119, 2018.

[28] L. Ziegelmeier, M. Kirby, and C. Peterson. Stratifying high-dimensional data based on proximity to the convex hull boundary. *SIAM Review*, 59(2):346–365, 2017.