University of California Santa Cruz

Data-Driven Computational Optimal Control for Stochastic Nonlinear Systems

Prof. Daniele Venturi and Prof. Qi Gong

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

DARPA Lagrange Program
Contract Number FA8650-18-1-7842

FINAL REPORT

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

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 (see Figure 1). 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 main research tasks are summarized in Figure 2. 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.

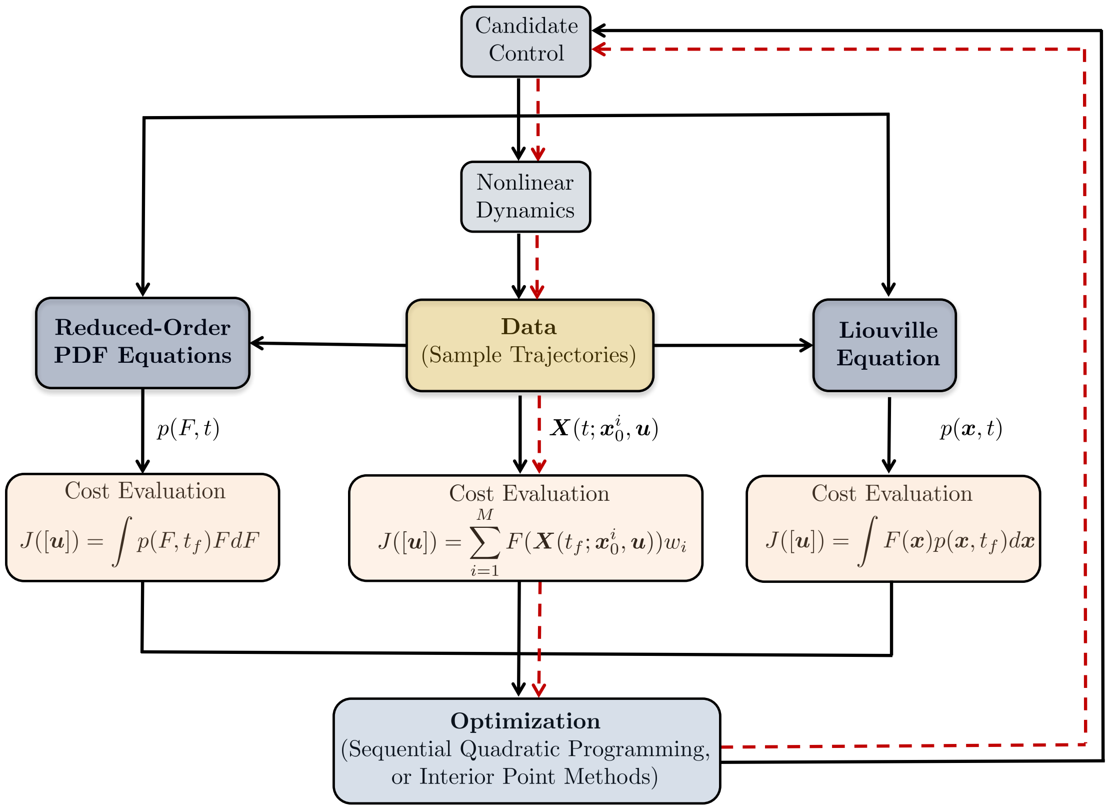

Candidate Control
Nonlinear Dynamics
Reduced-Order PDF Equations
Data (Sample Trajectories)
Liouville Equation
$p(F,t)$
$\boldsymbol{X}(t;\boldsymbol{x}_0^i,\boldsymbol{u})$
$p(\boldsymbol{x},t)$
Cost Evaluation $J([\boldsymbol{u}]) = \int p(F,t_f)F dF$
Cost Evaluation $J([\boldsymbol{u}]) = \sum_{i=1}^{M} F(\boldsymbol{X}(t_f;\boldsymbol{x}_0^i,\boldsymbol{u}))w_i$
Cost Evaluation $J([\boldsymbol{u}]) = \int F(\boldsymbol{x})p(\boldsymbol{x},t_f)d\boldsymbol{x}$
Optimization (Sequential Quadratic Programming, or Interior Point Methods)

Figure 1: Proposed data-driven optimal control architecture, emphasizing the role of probability density function equations instead of nonlinear dynamics in the control loop.

2 Project scope

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: Summary of the research tasks.

2.1 Task I(a): Parallel C++ class to compute hierarchical tensor formats

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 [27] 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 [64] (multivariate SVDs [23]) 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 $f(x_1,\ldots,x_5)$. In a hierarchical Tucker tensor representation f is written as

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

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 [23, 44, 46] 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

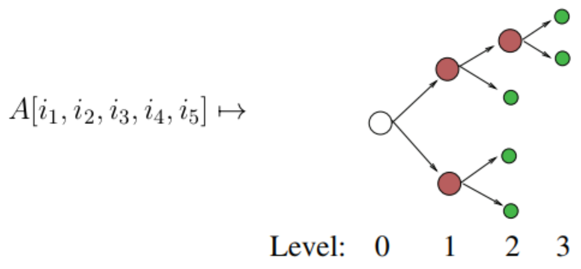


Figure 3: Graph representation of the hierarchical Tucker decomposition of a five-dimensional tensor.

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 [27] 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.,

$$A[i_1, i_2, \ldots, i_d] \in \mathbb{R}, \qquad i_k \in \mathcal{I}_k \qquad \forall k \in 1, \ldots, d. \tag{3}$$

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
$$A[i_1, i_2, \ldots, i_d] \tag{4}$$
2. Group the indexes by permuting them around so that
$$[i_1, i_2, \ldots, i_d] \mapsto \Big[[r_1, r_2, \ldots, r_m], [c_1, c_2, \ldots, c_{d-m}]\Big] \tag{5}$$
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:
$$A[i_1, i_2, \ldots, i_d] \mapsto B[r, c]. \tag{6}$$

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 [23] 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:

$$U_t[\,:\,, i] = \sum_j \sum_l B_t[i, j, l] \cdot (U_{s1}[\,:\,, j] \otimes U_{s2}[\,:\,, l])\,, \tag{7}$$

where the operator : has the Matlab/Octave meaning of "all entries in this index" and $\otimes$ is the Kronecker matrix product. Then (see [23]),

$$B[i, j, k] = \Big\langle U_t[\,:\,, i], \big(U_{s1}[\,:\,, j] \otimes U_{s2}[\,:\,, l]\big) \Big\rangle. \tag{8}$$

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 [23]). 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 $||A - A_{best}|| \leqslant \varepsilon$, then the singular values of $A^{(t)}$, denoted by σ_i for each node t can be estimated by

$$\sum_{i>k} \sigma_i^2 \leqslant \varepsilon^2. \tag{9}$$

On the other hand, if the singular values fulfill the theoretical bound $\sum_{i>k} \sigma_i^2 \leqslant \varepsilon^2/(2d-3)$, then the truncation yields a HT tensor $A_{\mathcal{H}}$ such that $||A - A_{\mathcal{H}}|| \leqslant \varepsilon$. 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.

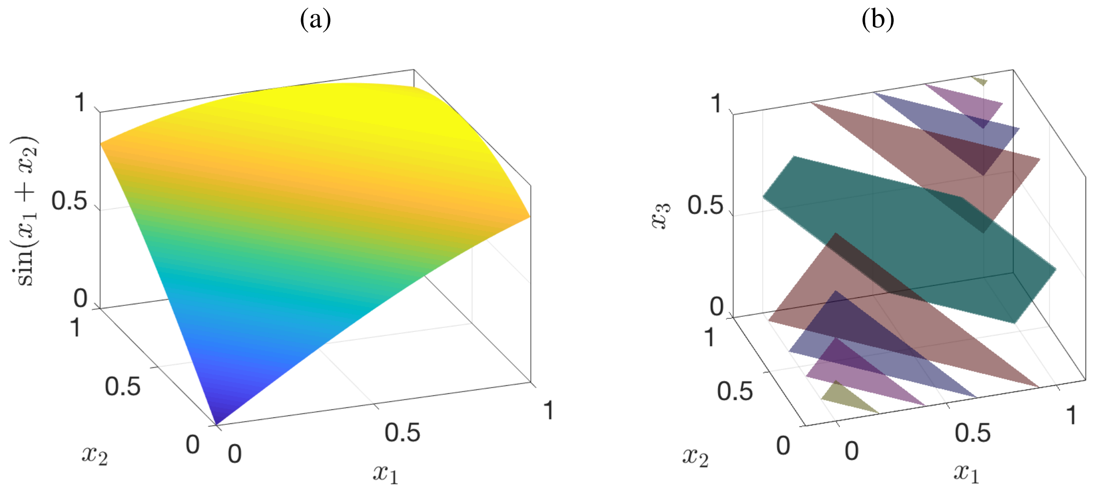


Figure 4: Sine function of equation (10) in (a) 2D, and (b) 3D with level sets (iso-surfaces) corresponding to $g = 0$ (green), $g = 0.2$ (magenta), $g = 0.4$ (blue), $g = 0.6$ (purple) and $g = 0.8$ (light green).

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 $[0,1]^d$.

$$g(x_1, x_2, \dots, x_d) = \sin\left(\sum_{i=1}^{d} x_i\right). \tag{10}$$

It was shown in [48] that $g(x_1, x_2, \dots, x_d)$ can be written as

$$g(x_1, ..., x_d) = \sum_{j=1}^{d} \sin(x_j) \prod_{\substack{i=1 \\ i\neq j}}^{d} \frac{\sin(x_i + \chi_i - \chi_j)}{\sin(\chi_i - \chi_j)}, \tag{11}$$

for any d-tuple of distinct numbers $\{\chi_1, \dots, \chi_d\}$. Therefore, in principle, $g(x_1, x_2, \dots, x_d)$ can be written as a fully diagonal HT decomposition with separation rank equal to $r = d$. In Figure 4 we plot the function (10) 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 (10) 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

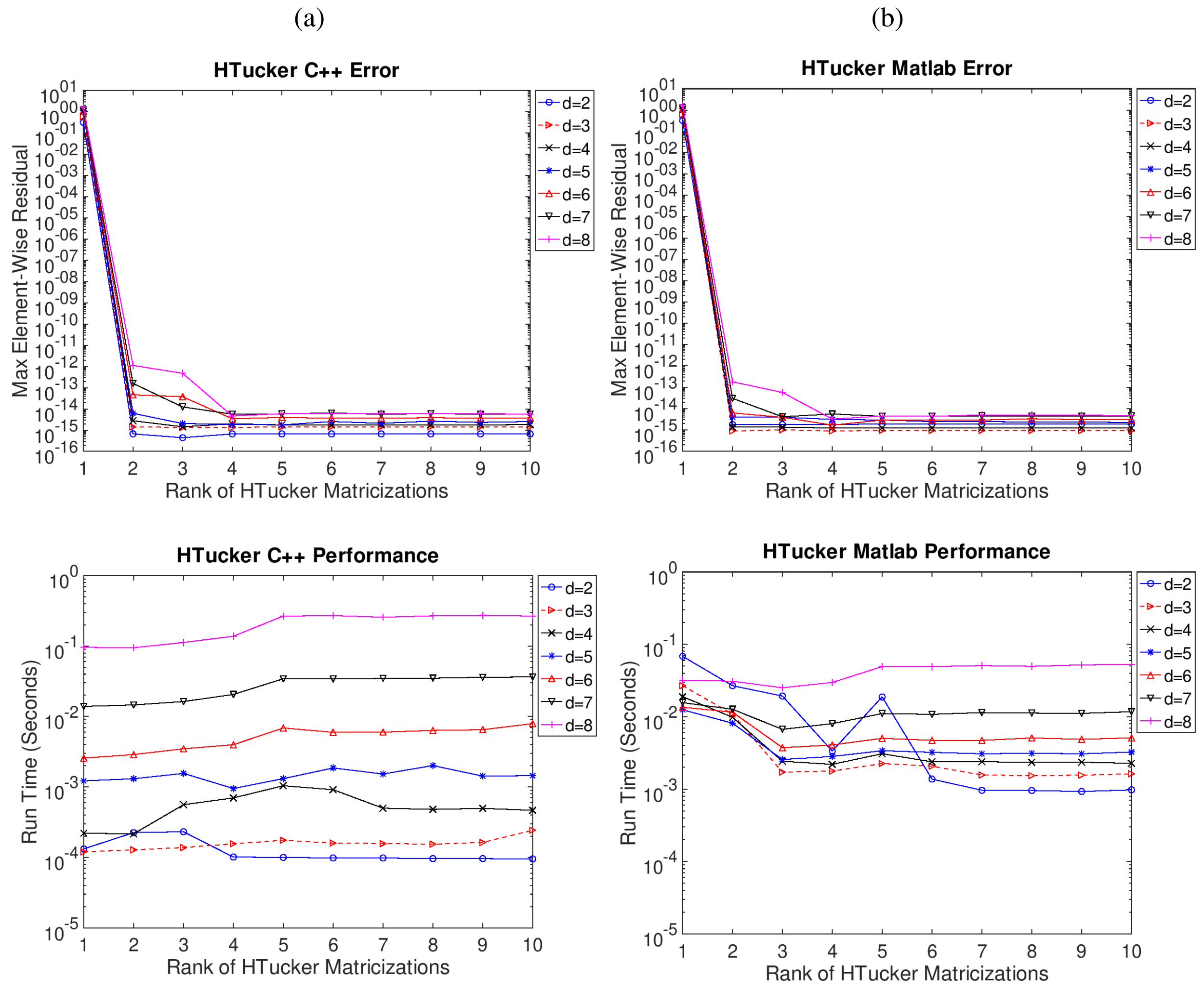
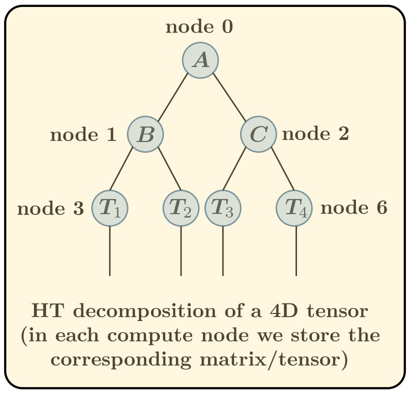


Figure 5: Comparison of Hierarchical HT decomposition of the sine function of equation (10) utilizing existing software tools (b) and the serial algorithm developed in this effort (a) (see section 2.1.4).

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.

2.1.7 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 [24]. 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

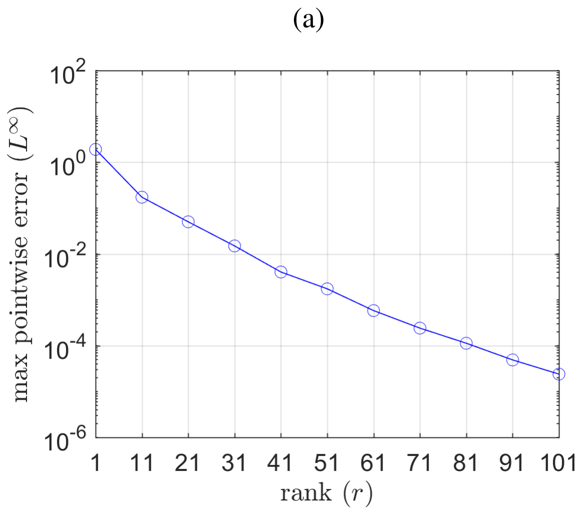

node 0
A
node 1
B
C
node 2
node 3
T_1
T_2
T_3
T_4
node 6
HT decomposition of a 4D tensor
(in each compute node we store the
corresponding matrix/tensor)

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

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

Figure 6: Parallel implementation of the HTucker C++ class for a 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 [44]). 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.



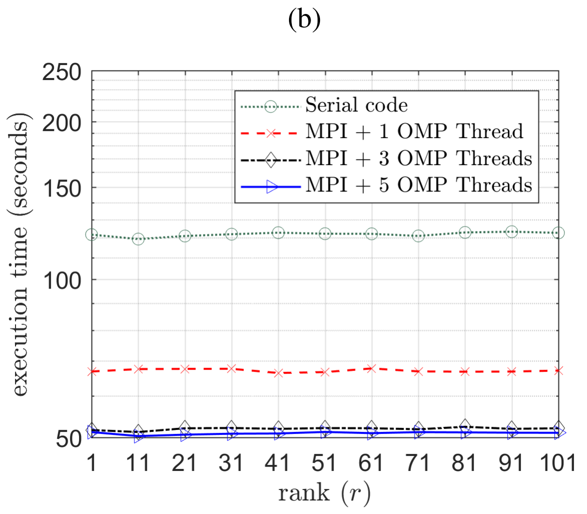


Figure 7: Performance of the HTucker C++ class with "root-to-leaves" [44] truncation in computing the tensor decomposition of the function in equation (12) utilizing 7 MPI nodes with up to 5 OpenMP threads in each node.

2.1.8 Numerical results: parallel C++ code

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

$$g(x_1, x_2, x_3, x_4) = \exp\left[\frac{\sin(5x_1x_2)\cos(5x_3)}{1+\cos(10x_1x_4)^2}\right]. \tag{12}$$

We sample g on a $60 \times 60 \times 60 \times 60$ grid in $[0,1]^4$. This yields a 4D numerical tensor with 12.96 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.

2.1.9 Application to a 4D Liouville equation

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

$$\begin{cases} \dfrac{\partial p(t, \boldsymbol{x})}{\partial t} + \boldsymbol{G} \cdot \nabla p(t, \boldsymbol{x}) = 0 \qquad t \geqslant 0, \quad \boldsymbol{x} \in \mathcal{D}, \\ p(0, \boldsymbol{x}) = p_0(\boldsymbol{x}) = \dfrac{1}{(4\pi^2\sigma^4)} \exp\left[-\dfrac{x_1^2 + x_2^2 + x_3^2 + x_4^2}{2\sigma^2} \right], \qquad \sigma = 2. \end{cases} \tag{13}$$

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

$$p(t, \boldsymbol{x}) = p_0(\boldsymbol{x} - \boldsymbol{G}t). \tag{14}$$

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:

$$\left.\frac{\partial p}{\partial x_1}\right|_{(t,[x_1^i, x_2^j, x_3^k, x_4^w])} \approx (\boldsymbol{D} \circ_1 \boldsymbol{P}(t))[i, j, k, w], \tag{15}$$

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 13 can be compactly written in an HTucker form as

$$\frac{d\boldsymbol{P}(t)}{dt} = -\sum_{k=1}^{4} G_k \boldsymbol{D} \circ_k \boldsymbol{P}(t). \tag{16}$$

To integrate the ODE system (16) in time, we use the the second-order explicit Adams-Bashforth scheme

$$\boldsymbol{P}(t_{n+1}) = \boldsymbol{P}(t_n) - \frac{\Delta t}{2} \sum_{k=1}^{4} G_k \boldsymbol{D} \circ_k (3\boldsymbol{P}(t_n) - \boldsymbol{P}(t_{n-1})). \tag{17}$$

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 (13) 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

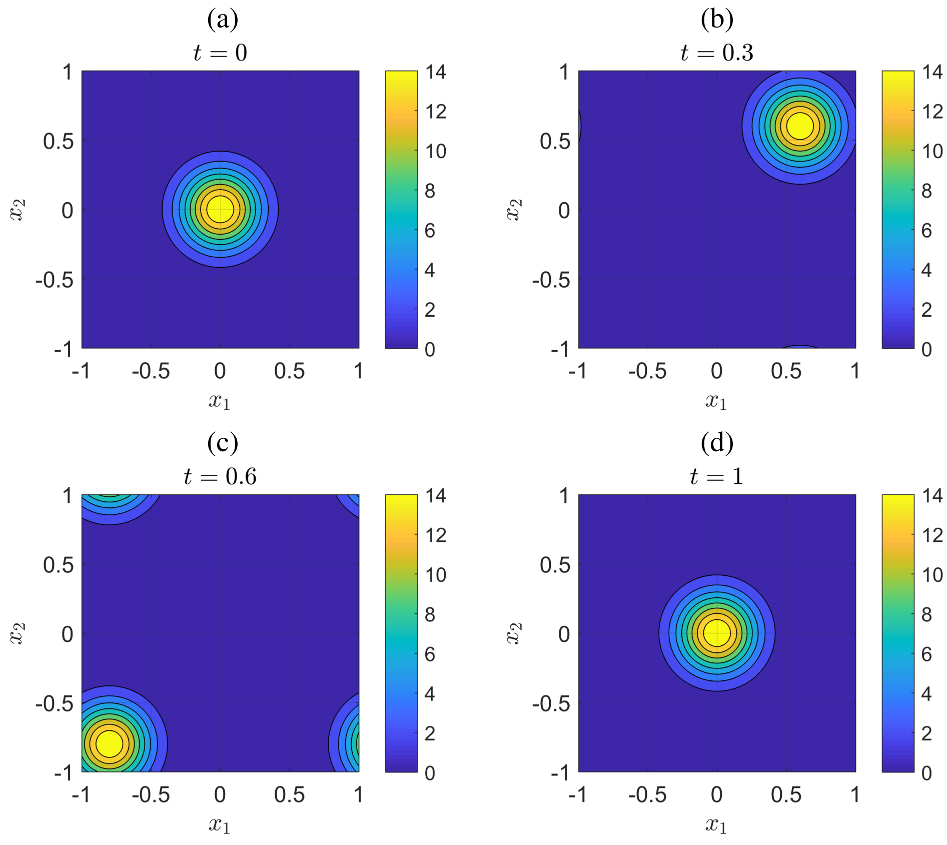


Figure 8: x_1x_2-plane solutions of the initial/boundary value problem in equation (13) with x_3 and x_4 both set to $2t$, at times (a) $t = 0.0$, (b) $t = 0.3$, (c) $t = 0.5$, (d) $t = 1.0$.

need to 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.

2.1.10 Parallel linear solvers for high-dimensional systems in the HT format

In the previous Section we studied solutions to high dimensional PDEs through the use of explicit multi-step schemes of the form

$$\boldsymbol{P}_{n+1} = \mathcal{R}(\boldsymbol{P}_n, \ldots, \boldsymbol{P}_{n-m}), \tag{18}$$

where $\mathcal{R}$ is linear in each argument, and $\boldsymbol{P}_n = \boldsymbol{P}(t_n)$ However, schemes of this form are not in general stable for all contractive $\mathcal{R}$. Moreover, they have increasingly restrictive time step limitations for situations in which the iterative schemes are stable. Thus, for increased numerical stability it is often necessary to use *implicit time stepping schemes*, i.e. schemes which implicitly represent the next iterate as the solution to a linear system of equations

$$\mathcal{L}(\boldsymbol{P}_{n+1}) = \mathcal{R}(\boldsymbol{P}_n, \ldots, \boldsymbol{P}_{n-m}), \tag{19}$$

where the (big) matrix $\mathcal{L}$ is assumed to be invertible. In the context of hierarchical Tucker tensor formats, solving such an equation for the next iterate with a restricted set of ranks – i.e., on a *manifold of tensors*

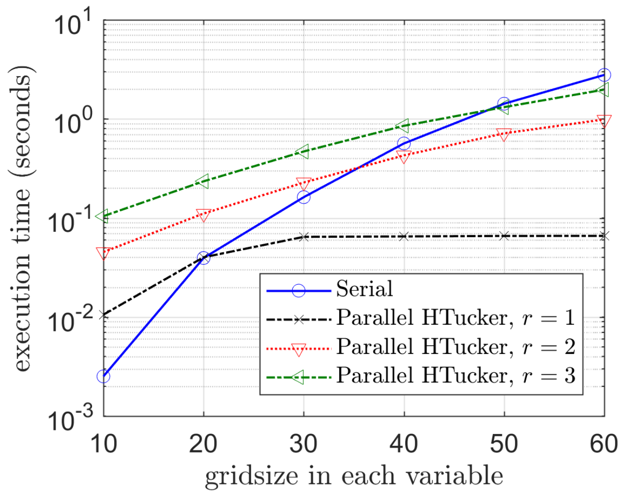


Figure 9: Computational time required to advect the solution back to initial position ($t = 1$) for the number of collocation points in each variable, x_i, utilizing 10^6 time steps from $t = 0$ to $t = 1$.

with constant rank – is computationally difficult, as it requires *Riemannian optimization*.[1] Perhaps, the simplest prototype problem one can think of is an equation of the form (19), where $\mathcal{L} = \bigotimes_{k=1}^{d} A_k$ is a tensor (Kronecker) product of invertible matrices. For this case there is an analytic solution of the form $\mathcal{L}^{-1} = \bigotimes_{k=1}^{d} A_k^{-1}$. An example of a more difficult non-separable case is the elliptic problem

$$\nabla^2 \phi(\boldsymbol{x}) = f(\boldsymbol{x}) \qquad \boldsymbol{x} \in \mathbb{R}^d, \tag{20}$$

where the Laplace operator in d dimensions is defined as

$$\nabla^2 = \sum_{k=1}^{d} I_1 \otimes \cdots \otimes I_{k-1} \otimes D_k^2 \otimes I_{k+1} \otimes \cdots \otimes I_d. \tag{21}$$

Here D_k^2 denotes the second-order derivative operator on a variable x_k. Written in this form we can see that attempting to directly invert this operator is not a simple task, though such inversions do exist (see, e.g., [26]). A possible approach to solve high-dimensional linear systems of the form (19) by using HTucker tensor formats is to reformulate the problem as an optimization problem. With this in mind, we aim at computing the solution to (19) by solving the following problem

$$\boldsymbol{P}_{n+1} \leftarrow \underset{\boldsymbol{P}}{\operatorname{argmin}} \, ||\mathcal{L}(\boldsymbol{P}) - \mathcal{R}(\boldsymbol{\Phi}_n, \ldots, \boldsymbol{\Phi}_{n-m})||_2^2 \tag{22}$$
$$\text{subject to: } \boldsymbol{P} \text{ is a HT tensor with constant rank}$$

It was shown in [16] that the problem above can be expressed as an optimization on a *Riemannian manifold* defined by the constrained ranks of $\boldsymbol{P}$. Roughly speaking, a Riemannian manifold is a topological space which is locally flat and has an inner product which smoothly varies from one point to another. We can find a local minimum to the optimization problem above by using the Riemannian line search algorithm described in [16], which is locally convergent [2, 30]. The algorithm involves a *retraction step* which can be accomplished via high-order singular value decomposition [30].

[1]Solving the linear system (19) on a tensor manifold with constant rank allows us to avoid the computationally intensive "rank reduction" step, which cannot be avoided if explicit schemes are used to solve high-dimensional Liouville equations.

Table 1: Preliminary results of the Poisson boundary value problem, equation (20) using the HTucker linear solver outlined in Section 2.1.10, with Riemann line search algorithm, 4th-order finite differences and Dirichlet boundary conditions.

Numerical solution of $\nabla^2\phi = f$ with HT-tensors and Riemannian optimization				
Dimension (d)	Iterate	Riemannian gradient	Residual	Max rank
2	320	3.16413e-1	4.84126e-05	4
3	124	2.61959e-2	1.51751e-05	5
4	32	4.41596e-3	5.14874e-05	3

2.1.10.1 Preliminary numerical results: To test the performance of the parallel linear solver we developed based on Riemannian line search, we have implemented a discrete form of Poisson's equation (20) with Dirichlet boundary conditions on the hyper-cube $[0, 1]^d$. To construct a benchmark solution to such problem, we take the function

$$\phi(\boldsymbol{x}) = \frac{\prod_{k=1}^{d} \sin(\pi x_k)}{2 + \sin\left(s\pi \prod_{k=1}^{d} x_k)\right)} \tag{23}$$

and compute its Laplacian $\nabla^2\phi$. This gives us the forcing f, which we approximate with 4th-order finite differences. The Riemanian optimization problem to compute the solution to the Poisson equation (20) can be formulated as follows

$$\underset{\boldsymbol{P}}{\operatorname{argmin}} \ \|\mathcal{L}\boldsymbol{P} - \boldsymbol{f}\|_2^2 \qquad \text{subject to: } \boldsymbol{P} \text{ is a HT tensor with constant rank.} \tag{24}$$

Here $\mathcal{L}$ is the discrete form of the Laplace operator (21), while $\boldsymbol{f}$ is an HT tensor representing the right hand side of (20).

In Table 1 we summarize the preliminary numerical results we obtained for $d = 2, 3, 4$. The frequency parameter s in (23) was chosen to be 1, specifically with the idea that a smoother unknown function will yield smaller hierarchical ranks. The number of collocation points along each axis of the box $[0, 1]^d$ is set to 31. The stopping condition used is to halt the iterations when there did not exist a step size small enough (but nonzero within floating point definition) to impact the value of the cost function within machine accuracy. The ranks were chosen based on several numerical tests, with the results yielding lowest found residuals given in Table 1. Based on numerical findings of these preliminary tests, it would appear that it is easier to find a decrease step size in low dimensions. This can be seen by the fact that larger dimensions failed to find a next viable iterate sooner, when [2] showed that such an iterate should always exist. With problems stated, it should be emphasized that the residual was brought down to near 10^{-5}. We are currently working on implementing a more advanced optimization framework – such as the Gauss-Newton method derived in [16] – to compute the solution to (22).

2.2 Task I(b): Data-driven methods to compute PDFs and flow maps

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

$$\dot{\boldsymbol{x}} = \boldsymbol{G}(\boldsymbol{x}(t)), \qquad \boldsymbol{x}(0) = \boldsymbol{x}_0 \sim p_0(\boldsymbol{x}), \tag{25}$$

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*

$$\boldsymbol{x} = \boldsymbol{\Phi}(\boldsymbol{x}_0, t), \tag{26}$$

which is a function of both the initial condition $\boldsymbol{x}_0$ and time t. It can be shown [9] that the *forward flow map* satisfies the flow map equation

$$\frac{\partial \boldsymbol{\Phi}(\boldsymbol{x}_0, t)}{\partial t} - (\boldsymbol{G}(\boldsymbol{x}_0) \cdot \nabla)\, \boldsymbol{\Phi}(\boldsymbol{x}_0, t) = 0 \qquad \boldsymbol{\Phi}(\boldsymbol{x}_0, 0) = \boldsymbol{x}_0. \tag{27}$$

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

$$\frac{\partial \boldsymbol{\Phi}_0(\boldsymbol{x}, t)}{\partial t} + (\boldsymbol{G}(\boldsymbol{x}) \cdot \nabla)\boldsymbol{\Phi}_0(\boldsymbol{x}, t) = 0, \qquad \boldsymbol{\Phi}_0(\boldsymbol{x}, 0) = \boldsymbol{x}. \tag{28}$$

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

$$\frac{\partial p(\boldsymbol{x}, t)}{\partial t} + \nabla \cdot [p(\boldsymbol{x}, t)\boldsymbol{G}(\boldsymbol{x})] = 0. \tag{29}$$

The analytical solution to (29) can be expressed with the method of characteristics as

$$p(\boldsymbol{x}, t) = p_0(\boldsymbol{\Phi}_0(\boldsymbol{x}, t)) \exp\left[-\int_0^t \nabla \cdot \boldsymbol{G}(\boldsymbol{\Phi}(\boldsymbol{x}_0, \tau))d\tau\right], \tag{30}$$

where $\boldsymbol{\Phi}_0(\boldsymbol{x}, t)$ is the *inverse flow map* [19] satisfying (28). From (30), we see that if the system is volume-preserving, i.e., if $\nabla \cdot \boldsymbol{G} = 0$ then we have

$$p(\boldsymbol{x}, t) = p_0(\boldsymbol{\Phi}_0(\boldsymbol{x}, t)). \tag{31}$$

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 (25) 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 [57]. 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).

In the first case, the PDF of the system is estimated by training the neural net sketched in Figure 10 entirely with sample paths[2] of (25). In practice, we minimize a cost function of the form

$$MSE_{\text{data}}(\boldsymbol{\theta}_1, ..., \boldsymbol{\theta}_M, t) = \frac{1}{N_d}\sum_{k=1}^{N_d}\left[\log\left(p(\boldsymbol{x}^{(k)}, t)\right) - \log\left(\hat{p}(\boldsymbol{x}^{(k)}, t)\right)\right]^2, \tag{32}$$

[2]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 $t = t_f$.

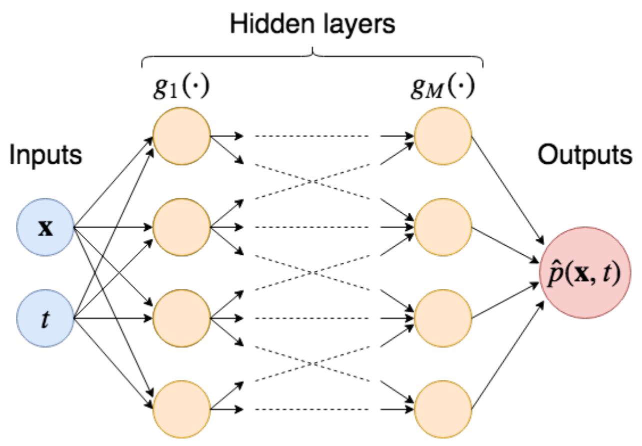


Figure 10: Architecture of a feed-forward neural net for approximating $p(\boldsymbol{x},t)$ by a composition of functions: $\hat{p}(\boldsymbol{x},t) = g_M \circ g_{M-1} \circ \dots g_1(\boldsymbol{x},t)$.

where $p(\boldsymbol{x}^{(k)},t)$ is obtained by solving (29) with the method of characteristics, $\hat{p}(\boldsymbol{x}^{(k)},t)$ is the neural net representation

$$\hat{p}(\boldsymbol{x},t) = g_M \circ g_{M-1} \circ \dots g_1(\boldsymbol{x},t) \tag{33}$$

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. 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 (32), $\boldsymbol{x}^{(k)} = \boldsymbol{\Phi}(\boldsymbol{x}_0^{(k)}, t)$ denotes the the position of the particle $\boldsymbol{x}_0^{(k)}$ at time t, which can be easily determined by integrating system (25) 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 (33) into the Liouville equation (29), i.e.,

$$R(\boldsymbol{x},t) = \frac{\partial \hat{p}(\boldsymbol{x},t)}{\partial t} + \nabla \cdot \left(\boldsymbol{G}(\boldsymbol{x})\hat{p}(\boldsymbol{x},t)\right). \tag{34}$$

In this case, the cost functional we consider is

$$MSE_{\text{PINN}}(\boldsymbol{\theta}_1, ..., \boldsymbol{\theta}_M, t) = MSE_{\text{data}}(\boldsymbol{\theta}_1, ..., \boldsymbol{\theta}_M, t) + \mu MSE_{\mathcal{L}}(\boldsymbol{\theta}_1, ..., \boldsymbol{\theta}_M, t), \tag{35}$$

where μ is penalty parameter and

$$MSE_{\mathcal{L}}(\boldsymbol{\theta}_1, ..., \boldsymbol{\theta}_M, t) = \frac{1}{N_c} \sum_{k=1}^{N_c} \left[R(\boldsymbol{x}^{(k)}, t)\right]^2 \tag{36}$$

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 (25)). The residual (34) can be easily evaluated by using automatic differentiation techniques applied to (33).

2.2.1.1 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

$$\begin{cases} \dot{x} = 2xy - 1 \\ \dot{y} = -x^2 - y^2 + \mu \end{cases} \tag{37}$$

The phase portraits of the system (37) are plotted in Figure 11 for different values of the parameter μ. We observe that the system undergoes two saddle node bifurcations at $\mu = 1$.

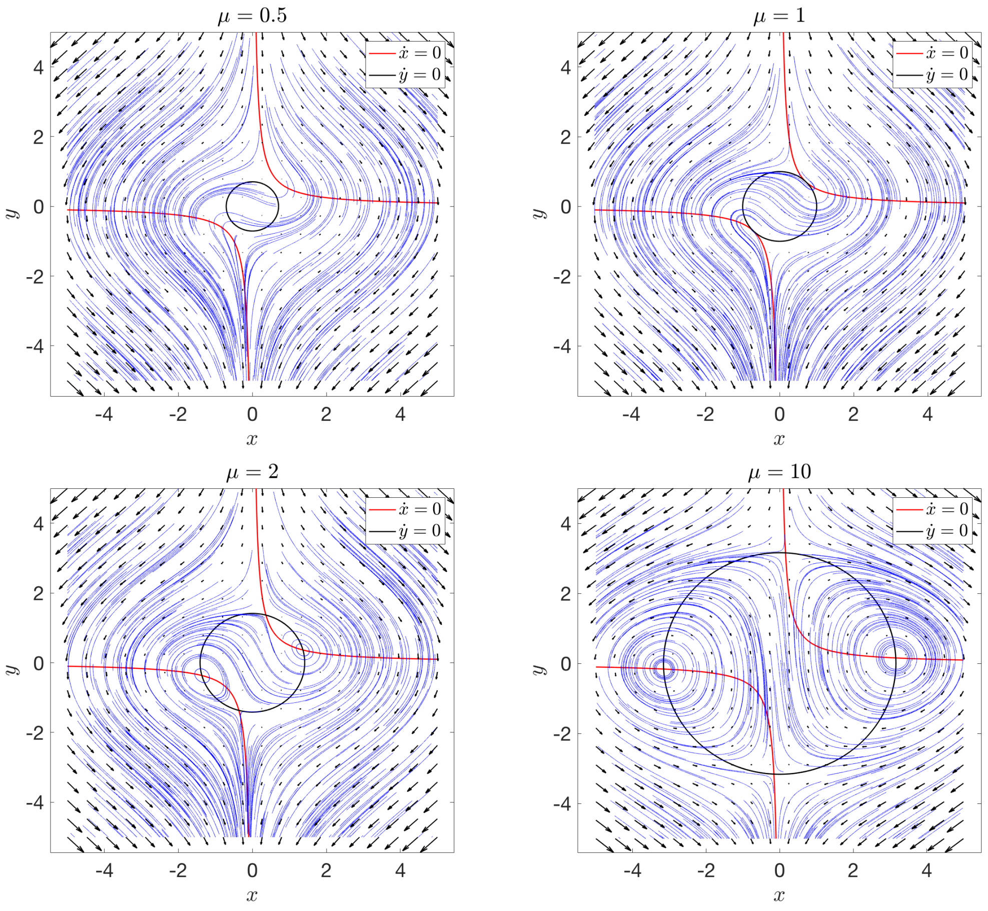


Figure 11: Phase portraits of the system (37) for different values of μ.

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 (25).

Data-driven machine learning This algorithm is purely based on data, i.e., sample trajectories of (25), *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 (32).

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 (25) 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

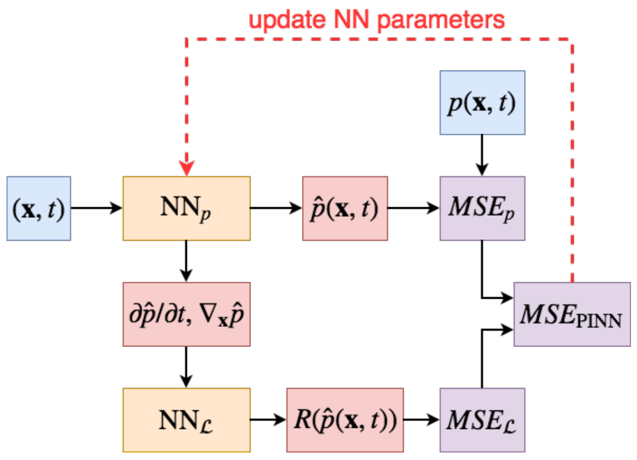


Figure 12: Training physics-informed neural nets (PINN).

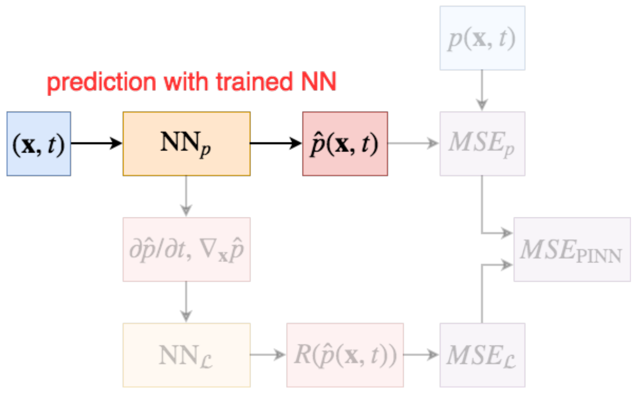

prediction with trained NN
$p(\mathbf{x}, t)$
$(\mathbf{x}, t)$
NN_p
$\hat{p}(\mathbf{x}, t)$
MSE_p
$\partial\hat{p}/\partial t, \nabla_{\mathbf{x}}\hat{p}$
MSE_{PINN}
$\mathrm{NN}_{\mathcal{L}}$
$R(\hat{p}(\mathbf{x}, t))$
$MSE_{\mathcal{L}}$

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

(30). 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 (29).

2. The two nets are trained simultaneously with the cost function (35) (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 (25) 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 (37) 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;
- 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.

2.2.4.1 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 (37) 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 [57]. We used an L-BFGS [10] 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. 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 (30). 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 (37) rapidly advect the smooth Gaussian into a thin curve. In Table 2 and 3 we present speed and accuracy results for estimating $p(x, y, t = 1.0)$ depending on the architecture of the neural net. For Table 4 and 5 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.

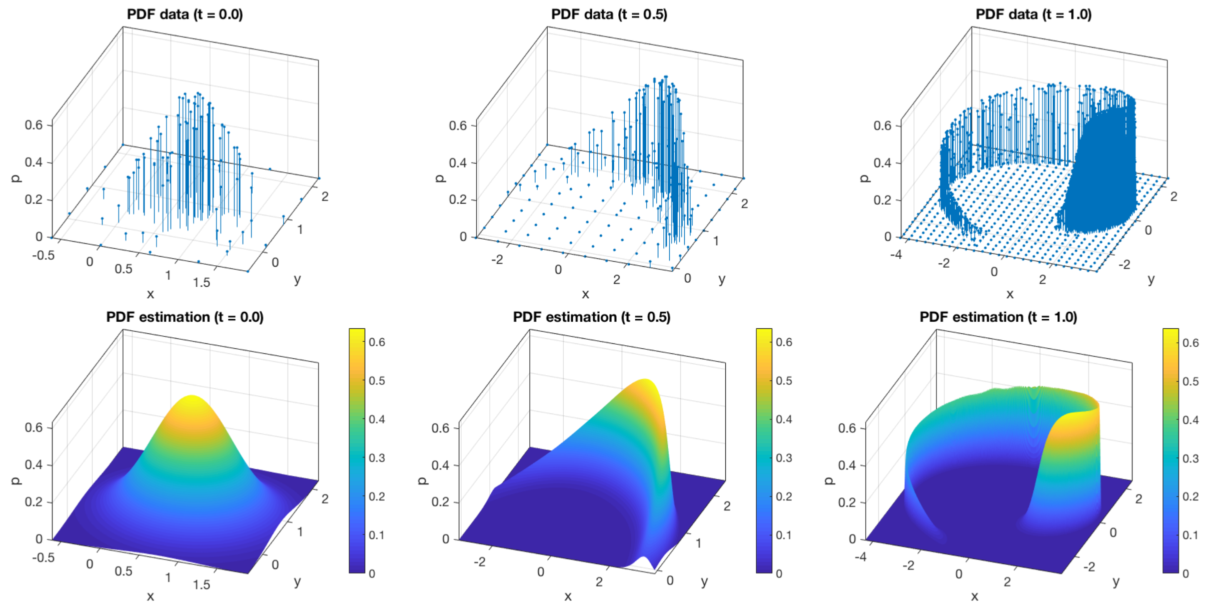


Figure 14: Training data (top, 100, 200, and 2000 points, left to right) and 1000 × 1000 reconstructions (bottom) of initial PDF $p_0(x, y)$, $p(x, y, t = 0.5)$, and $p(x, y, t = 1.0)$, left to right, using 8 hidden layers with 20 neurons each.

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 2 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 3 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 2 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.

2.2.4.2 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 $t \in [0, 1]$.

Table 2: Validation RMSE results for estimating the final time PDF $p(x, y, t = 1.0)$ for training data, N_d, fixed at 2000, $N_s = 1100$ points for forward propagation, $N_m \times N_m = 30 \times 30$ uniform grid for backward propagation, measured for $N_v = 2500$ points on a 50×50 uniform grid.

		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 3: Training time for estimating the final time PDF $p(x, y, t = 1.0)$ for training data ($N_d = 2000$) and validation data ($N_v = 2500$) using TensorFlow 1.8 [1] on a 2012 MacBook Pro with 2.5 GHz Intel Core i5 processor and 4 GB RAM.

		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

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 (37), 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, $\left(x^{(i,k)}, y^{(i,k)}, t_k\right)$, $i = 1, \dots, N_s$, $k = 0, \dots, N_t - 1$. 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 (30) 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 numerical integration and the solution to the Liouville equation again supplied probability values.

Table 4: Validation RMSE measures for $N_v = 2500$ data points on a 50×50 uniform grid for estimating the PDF $p(x, y, t = 1.0)$ using 8 hidden layers and 20 neurons per-layer.

		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 5: Training time of a neural net with 8 layers and 20 neurons per layer, for estimating the PDF $p(x, y, t = 1.0)$.

		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

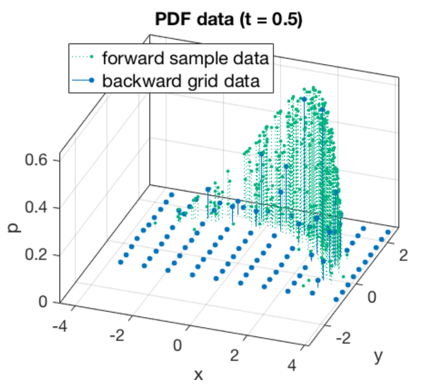
This first grid encouraged the net to learn the boundaries of approximate support of the PDF at each time step.

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 (37). 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 6 and 7 include results on the training speed and accuracy of the neural net depending on the net architecture. Tables 8 and 9 include results on the training speed and accuracy of the neural net depending on the availability of training data.

(a) (b)



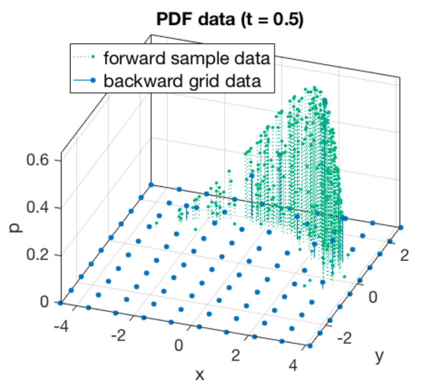


Figure 15: Forward/backward sample strategy for generating PDF data at $t = 0.5$ with $N_s = 100$ forward samples, and two $N_m \times N_m = 10 \times 10$ grids of backward samples shown in (a) and (b).

Discussion Tables 6 and 8 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 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 6 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 7. Finally, Table 8 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.

2.2.4.3 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 [57] 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 (35)). 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 10 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 11 with Table 7). With enough collocation points, we can also compensate for having only a few training data points, as seen in Table 12. 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 13), so overall there are no time savings with this method. The improved accuracy gained by using a PINN may be worth

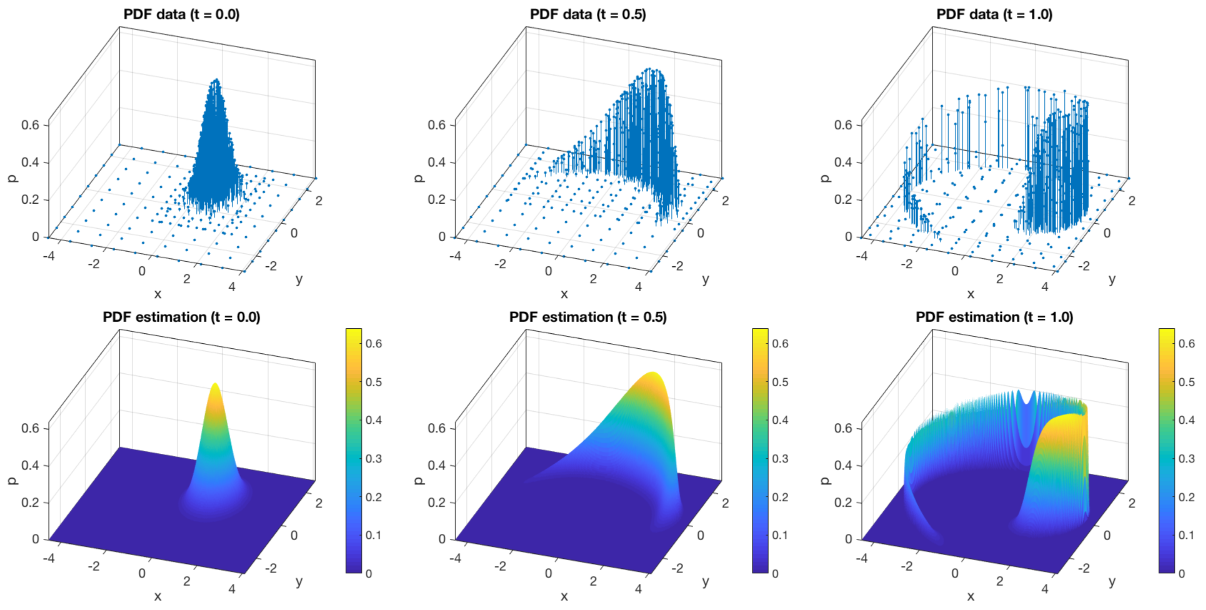


Figure 16: Training data (top) and PDF reconstruction (bottom) using 8 hidden layers and 20 neurons per layer, at times $t = 0.0$, 0.5, and 1.0.

the increased training time, depending on the problem. This will be important moving forward, as some systems may be more poorly behaved than (37), 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.

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 [70].

2. Aside from the PINN, the neural net architecture we used here was very bare-bones. Many sophisti-

Table 6: Validation RMSE results measures over $N_v = 2500$ data points for estimating the time dependent PDF $p(x, y, t)$ over the interval $t \in [0.0, 1.0]$ using $N_d = 5000$ training data distributed in 10 snapshots.

		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 7: Neural net training time for estimating the time dependent PDF $p(x, y, t)$ described in Table 6.

		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

cated 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 (35). 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 [57], which imposes the governing PDE on the model with a single penalty term (36). 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 (29) 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 net multiple times, each time increasing the weight of the second term (36) in the training cost function (35). 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.

2.2.5 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 (29) as a

Table 8: Validation RMSE results for estimating the time dependent PDF $p(x, y, t)$ over the interval $t \in [0.0, 1.0]$ using a neural net with 8 hidden layers and 20 neurons per layer.

		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 9: Training time of a neural net with 8 hidden layers, 20 neurons per layer, for estimating the time dependent PDF $p(x, y, t)$ over the interval $t \in [0.0, 1.0]$.

		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

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 (35), 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.

2.2.5.1 Increasing the penalty weight Increasing the penalty weight μ in (35) 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 (29). However, the sequences of weights we tried typically made the solution *worse* over time [52]. We ran 10 simulations, each with two different sequences of penalty weights, i.e.,

$$\{\mu_E\} = \left\{\sqrt{i_{E-1} + E}\right\} \qquad \text{and} \qquad \{\mu_E\} = \{i_{E-1} + E\}, \tag{38}$$

where $E = 1, 2, \dots, 10$ 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.

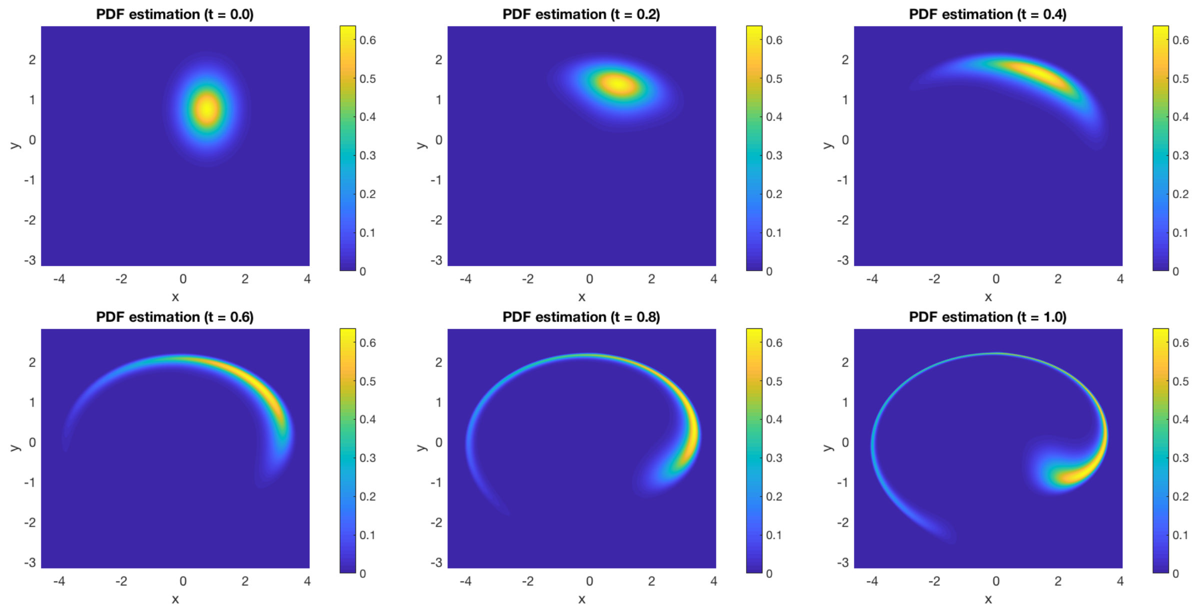


Figure 17: Prediction of $p(x, y, t)$ for the system (37) using a physics-informed neural net with 8 hidden layers, each with 20 neurons, trained on $N_d = 2000$ data points and $N_c = 8000$ total collocation points.

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 $\{\mu_E\} = \{\sqrt{i_{E-1} + E}\}$ 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 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.

2.2.5.2 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

Table 10: 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 trained on $N_d = 2000$ data points and $N_c = 8000$ total 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 11: Training time of a physics-informed neural net for estimating the time dependent PDF $p(x, y, t)$ over the interval $t \in [0.0, 1.0]$ with a physics-informed neural net trained on $N_d = 2000$ data points and $N_c = 8000$ total collocation points.

		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

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

$$\{N_{c,E}\}_{\text{root}} = \left\{ N_c \sqrt{\frac{E/E_{\text{max}} + 1}{2}} \right\}, \tag{39}$$

$$\{N_{c,E}\}_{\text{linear}} = \left\{ N_c \frac{E/E_{\text{max}} + 1}{2} \right\}. \tag{40}$$

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 (39)-(40), and compared the results to the unrefined PINN results used before. We notice that for the root sequence (39), 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 (40), 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 (35) significantly more robust, and with the sequence (40) 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.

Table 12: Validation RMSE ($N_v = 2500$) 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 with 8 hidden layers and 20 neurons per layer.

		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 13: 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 with 8 hidden layers and 20 neurons per layer.

		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

2.2.5.3 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 $\{\mu_E\} = \left\{\sqrt{i_{E-1} + E}\right\}$. 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 (39) with $\{\mu_E\} = \left\{\sqrt{i_{E-1} + E}\right\}$, 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 (40) with $\{\mu_E\} = \left\{\sqrt{i_{E-1} + E}\right\}$, 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.

2.2.5.4 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 (32), encourages the model to accurately learn minute differences where p is small, but ignore relatively large differences between

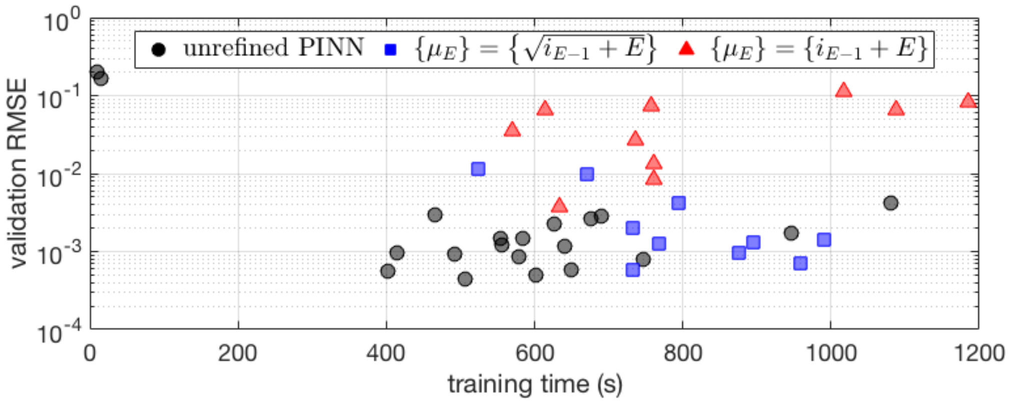


Figure 18: Performance of physics-informed neural nets(8 hidden layers with 20 neurons each) with cost function (35) and fixed $\mu = 1$ (unrefined PINN) versus neural nets where we increase the penalty parameter $\{\mu_E\}$ ($E = 1, \ldots, 10$) on-the-fly as optimization proceeds.

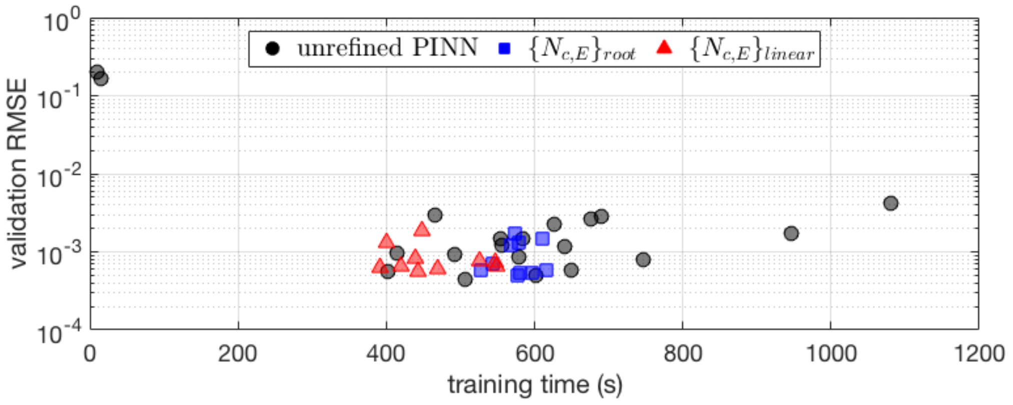


Figure 19: Performance of physics-informed neural nets with cost function (35) and fixed $\mu = 1$ (unrefined PINN) versus neural nets where we randomly select collocation points $\{N_{c,E}\}$ according to the sequences in (39)-(40). Here we use $N_d = 2500$ training data, $N_c = 10000$ total collocation points, $E_{\max} = 10$ epochs, and 8 hidden layers with 20 neurons each.

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 (32), we calculate

$$MSE_{\text{data, true}}(\boldsymbol{\theta}) = \frac{1}{N_d}\sum_{i=1}^{N_d}\left[\exp\log\hat{p}\left(\boldsymbol{x}^{(i)},t\right) - p\left(\boldsymbol{x}^{(i)},t\right)\right]^2. \tag{41}$$

  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.

$$MSE_{\text{data, weighted}}(\boldsymbol{\theta}) = \frac{1}{N_d}\sum_{i=1}^{N_d} p\left(\boldsymbol{x}^{(i)},t\right)\left[\log\hat{p}\left(\boldsymbol{x}^{(i)},t\right) - \log p\left(\boldsymbol{x}^{(i)},t\right)\right]^2. \tag{42}$$

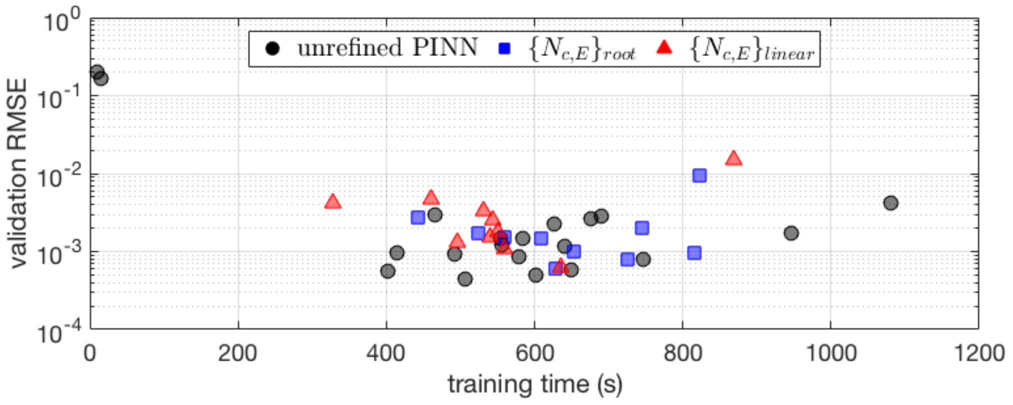


Figure 20: Performance of physics-informed neural nets (8 hidden layers with 20 neurons each) with cost function (35) and fixed $\mu = 1$ (unrefined PINN) versus neural nets where we randomly select collocation points $\{N_{c,E}\}$ in each iteration/epoch, along with increasing scalar penalty weights $\{\mu_E\} = \{\sqrt{i_{E-1} + E}\}$.

- **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 [45]:

$$\begin{aligned} \text{sD}_{\text{KL}}(p\|\hat{p}(\boldsymbol{\theta})) &= \text{D}_{\text{KL}}(p\|\hat{p}(\boldsymbol{\theta})) + \text{D}_{\text{KL}}(\hat{p}(\boldsymbol{\theta})\|p), \\ &= \int p(\boldsymbol{x}) \left[\log p(\boldsymbol{x}) - \log \hat{p}(\boldsymbol{x})\right] d\boldsymbol{x} + \int \hat{p}(\boldsymbol{x}) \left[\log \hat{p}(\boldsymbol{x}) - \log p(\boldsymbol{x})\right] d\boldsymbol{x}. \end{aligned} \tag{43}$$

  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

$$\begin{aligned} \text{sD}_{\text{KL}}(p\|\hat{p}(\boldsymbol{\theta})) \approx V \frac{1}{N_d} \sum_{i=1}^{N_d} p\left(\boldsymbol{x}^{(i)}, t\right) \left[\log p\left(\boldsymbol{x}^{(i)}, t\right) - \log \hat{p}\left(\boldsymbol{x}^{(i)}, t\right)\right] \\ + V \frac{1}{N_d} \sum_{i=1}^{N_d} \exp\left(\log \hat{p}\left(\boldsymbol{x}^{(i)}, t\right)\right) \left[\log \hat{p}\left(\boldsymbol{x}^{(i)}, t\right) - \log p\left(\boldsymbol{x}^{(i)}, t\right)\right] \\ = \frac{V}{N_d} \sum_{i=1}^{N_d} \left[p\left(\boldsymbol{x}^{(i)}, t\right) - \exp\left(\log \hat{p}\left(\boldsymbol{x}^{(i)}, t\right)\right)\right] \left[\log p\left(\boldsymbol{x}^{(i)}, t\right) - \log \hat{p}\left(\boldsymbol{x}^{(i)}, t\right)\right]. \end{aligned} \tag{44}$$

  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 (41).

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 (35) for the so-called "unrefined PINN". For validation, we still use the root mean square error (RMSE) with 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

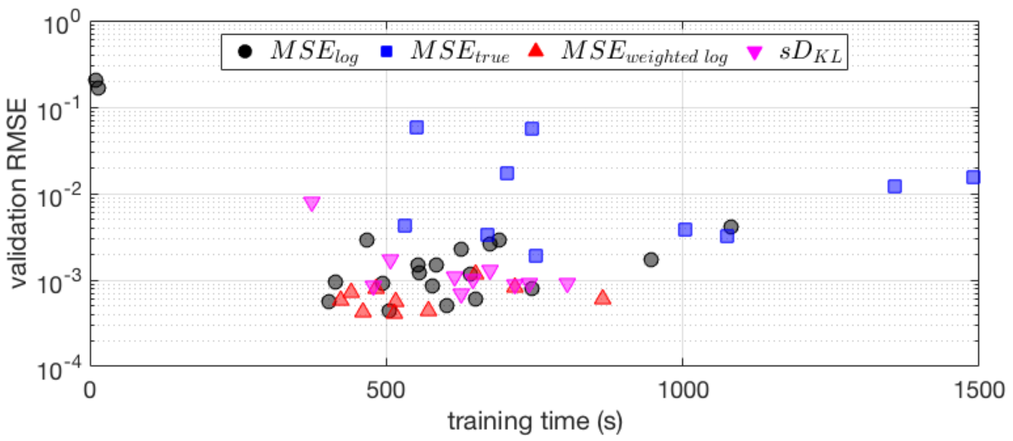


Figure 21: Performance of physics-informed neural nets (8 hidden layers with 20 neurons each) with cost function (35), fixed $\mu = 1$, $N_d = 2500$ training data, and $N_c = 10000$ collocation points.

summarizes all results we obtained. We see that if we use the cost (41), 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 (42) 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 (44) 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 (41) takes longer and is significantly less accurate than training on the log probability data (32). 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 (42) and the symmetric Kullback-Leibler divergence (44) compare well to MSE_{data} (32). 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.

2.2.6 Application to the Van der Pol oscillator

So far we tested our algorithms to the divergence-free system (37). 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

$$\begin{cases} \dot{x} = y, \\ \dot{y} = \mu\left(1 - x^2\right) y - x. \end{cases} \tag{45}$$

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 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

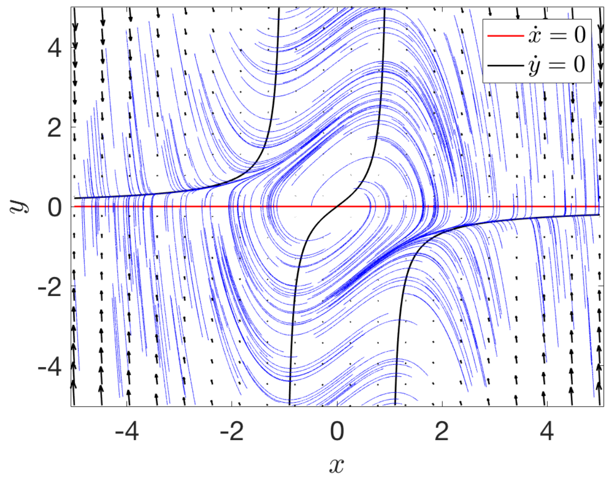


Figure 22: Phase portrait of the Van der Pol oscillator (45) for $\mu = 1$.

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 (30), will alleviate this problem.

2.2.6.1 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`dop853` ODE integrator [28, 33]. We then used simple trapezoidal integration over the resulting time series to approximate (30), 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 $p(\boldsymbol{x}, t = t_k)$.

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

$$NRMSE_{\text{val}} = \frac{\sqrt{\frac{1}{N_v}\sum_{i=1}^{N_v}\left[p\left(\boldsymbol{x}^{(i)}\right) - \hat{p}\left(\boldsymbol{x}^{(i)}\right)\right]^2}}{\max_i p\left(\boldsymbol{x}^{(i)}\right) - \min_i p\left(\boldsymbol{x}^{(i)}\right)}. \tag{46}$$

For example, one trial on the two-dimensional divergence-free test system (37) 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.

2.2.6.3 Numerical results In this Section we study the random initial value problem for the Van der Pol equation (45). We consider three different cases

Table 14: Training time, RMSE, and NRMSE of Van der Pol oscillator, equation (45), with initial PDF centered at origin, utilizing randomized samples and increasing penalty weights.

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

- 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.

2.2.6.4 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 (42) for the training cost and randomized collocation points according to the formula (40) 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 14 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.

2.2.6.5 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 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.

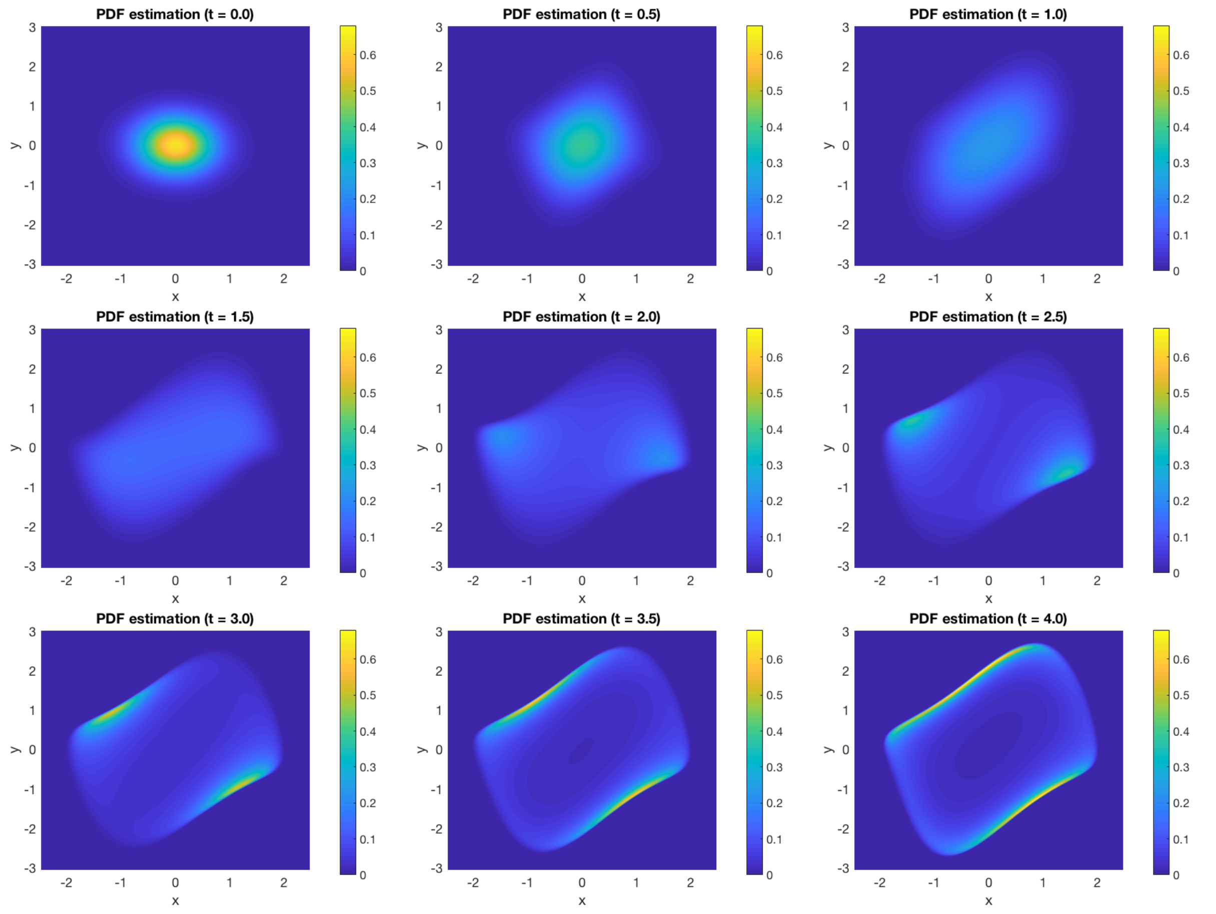


Figure 23: Time evolution of $p(x, y, t)$ for the Van der Pol oscillator, equation (45), where the initial PDF is the product of two indepent Gaussians with means $\mu_x = \mu_y = 0$ and variances $\sigma_x^2 = \sigma_y^2 = 0.25$. Here the net is trained with data in the interval $t_0 = 0.0$ to $t_f = 3.0$, so the final two frames are extrapolated.

2.2.6.6 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.

2.2.7 Improved data generation with convex hull approximation and PDF support tracking

Our previous method of data generation relied heavily on backward sampling. We observed that samples

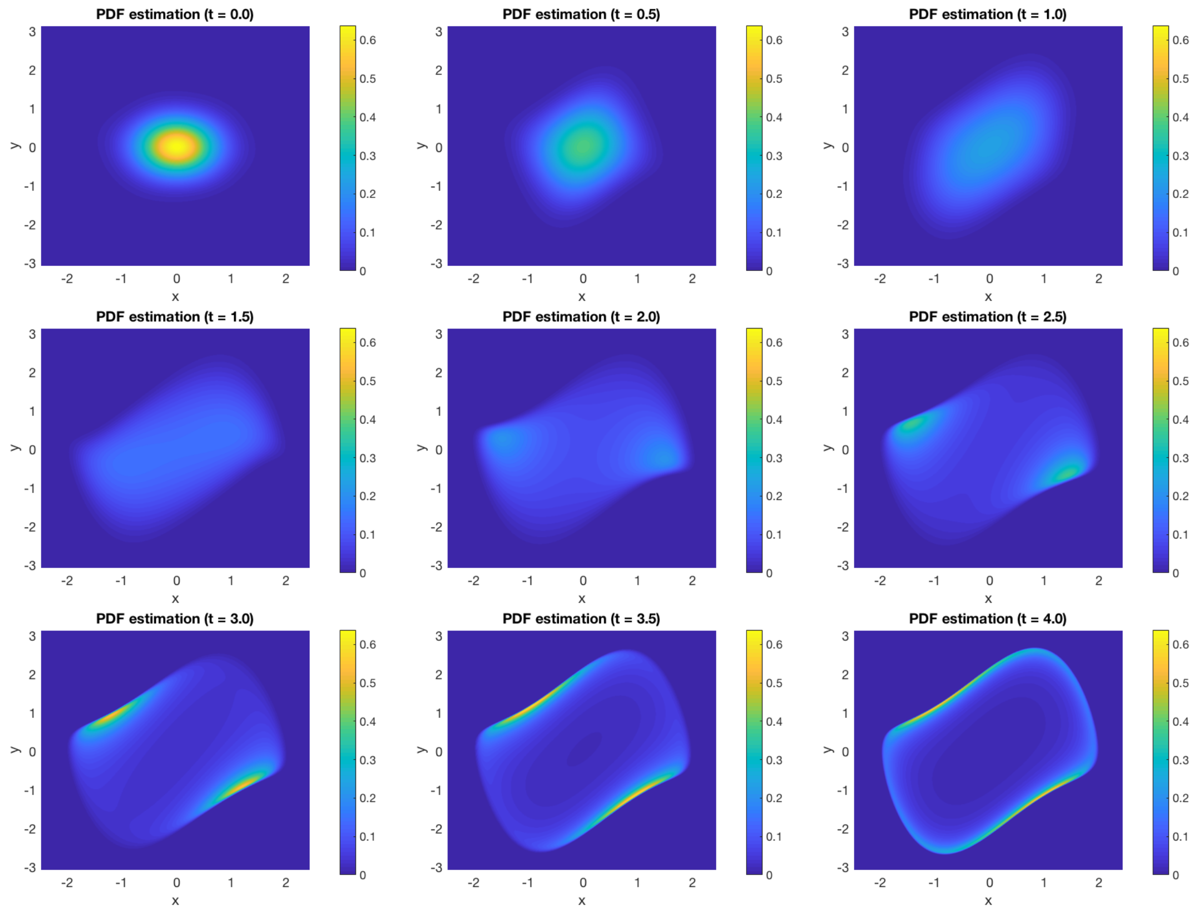


Figure 24: Time evolution of $p(x, y, t)$ for the Van der Pol oscillator (45) where the initial PDF is the product of two indepent Gaussians with means $\mu_x = \mu_y = 0$ and variances $\sigma_x^2 = \sigma_y^2 = 0.25$. Here the net is trained with data in the interval $t_0 = 0.0$ to $t_f = 4.0$.

generated in this way often shot back far outside the region of interest enclosing the support of the PDF, and, even worse, numerical integration would simply fail. Thus, in higher-dimensions this strategy would be very ineffective. Hence, we studied new ways to obtain better data from the support of the PDF, while at the same time reducing the number of trajectories which need to be integrated.

2.2.7.1 Convex hull stratification algorithm We have observed that the support of $p(\boldsymbol{x}, t_f)$ (t_f final time) often lies on a thin filament or a strange attractor, which is of relatively low dimension compared to the overall phase space. Given just a few data points $\{\boldsymbol{x}^{(i)}(t_f)\}$ generated by forward integration, we wish to construct an estimate of the support of the PDF $p(\boldsymbol{x}, t_f)$. Such an estimate would allow for smarter backward integration, i.e. picking samples in such a way as to fill in the gaps in the data. To this end, we implemented a convex hull stratification algorithm (CHSA) similar to the one proposed in [70]. CHSA builds an *approximately* convex hull of high-dimensional data by representing each data point as a linear combination of its K nearest neighbors. For our purposes, an approximately convex hull is far better than a truly convex hull, since in general, the support of $p(\boldsymbol{x}, t_f)$ is highly non-convex. In the CHSA, the L_2 norm of the weight vector roughly stratifies each data point by its distance to the boundary of the data cloud (in this case, the trajectory tube at t_f). In addition, each point with any negative weights is picked out as

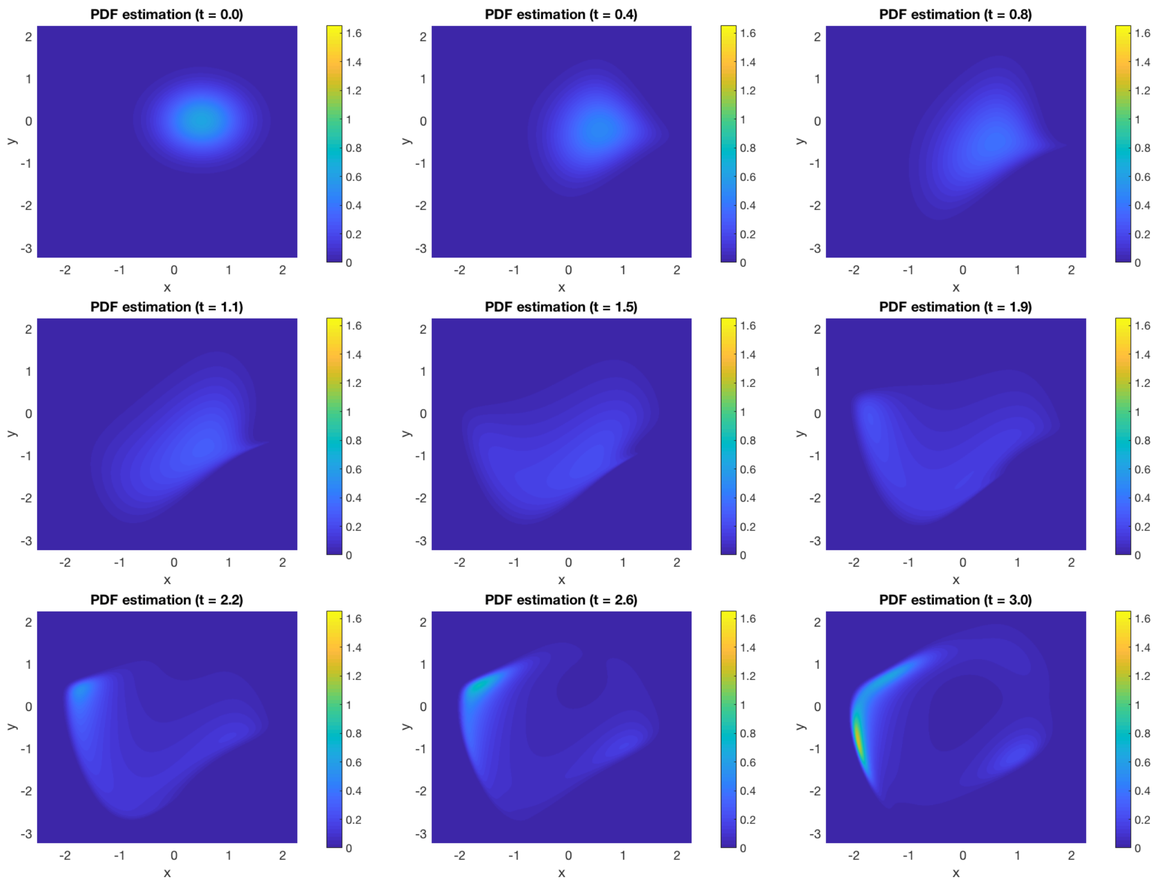


Figure 25: Time evolution of $p(x, y, t)$ for the Van der Pol oscillator (45) where the initial PDF is the product of two independent Gaussians with means $\mu_x = 0.5$, $\mu_y = 0$ and variances $\sigma_x^2 = \sigma_y^2 = 0.25$. Here the net was trained with data in the interval $t_0 = 0.0$ to $t_f = 3.0$.

a vertex of the approximate convex hull, since the only way to represent vertices as linear combinations of other points is through negative weights [70]. There are several parameters that needs to be tuned in the algorithm which can provide quite different representations of such approximately convex hull. With this representation at hand, however, how exactly to generate additional data is another question. Here we use a heuristic process wherein we sampled a few points in a Gaussian distribution centered at each vertex, as well as a few points on the lines connecting each vertex and its n nearest neighbors. An example of the results of the CHSA are shown in Figure 28. Each of these new samples is then integrated backwards, with results saved at all time steps (assuming integration is successful). Most of the time integration is successful, since the new samples tend to be near the boundary of the support of $p(\boldsymbol{x}, t_f)$.

2.2.7.2 Reducing the computational cost of numerical integration by tracking the PDF support We previously filled in the phase space by backward integration from a grid of points at each time step. This was expensive even in just a few dimensions, and many of these integrations were wasted either because they provided only one data point, or failed and were then set to minimum density. To overcome this problem, at each time snapshot we picked points $\left\{\boldsymbol{x}^{(i)}(t_k)\right\}$ by Latin hypercube sampling. Of these points, any which landed close enough to the center of the trajectory tube (generated by forward sampling and backward sampling from final time as described above) were numerically integrated, while any points lying outside

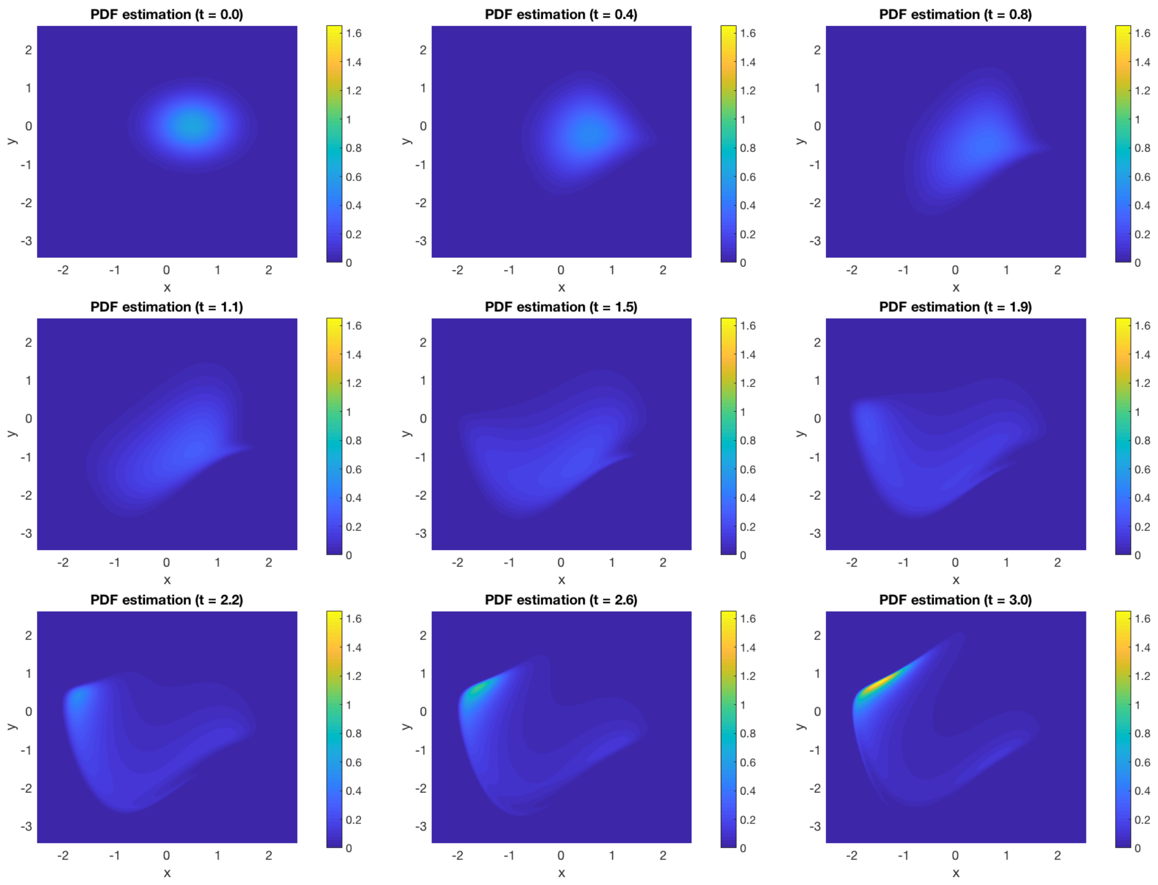


Figure 26: Time evolution of $p(x, y, t)$ for the Van der Pol oscillator (45) where the initial PDF is the product of two independent Gaussians with means $\mu_x = 0.5$, $\mu_y = 0$ and variances $\sigma_x^2 = \sigma_y^2 = 0.25$. Here the net was trained with data in the interval $t_0 = 0.0$ to $t_f = 3.0$.

of a few standard deviations were set to the minimum density. This simple change drastically reduces the number of integrations: *we can typically reduce the number of samples needed by an order of magnitude or more*.

2.2.8 Application to high-dimensional problems

2.2.8.1 Forced duffing oscillator With the ability to generate high-dimensional data efficiently, we can now test our method on high-dimensional problems. The first problem we consider here is a periodically

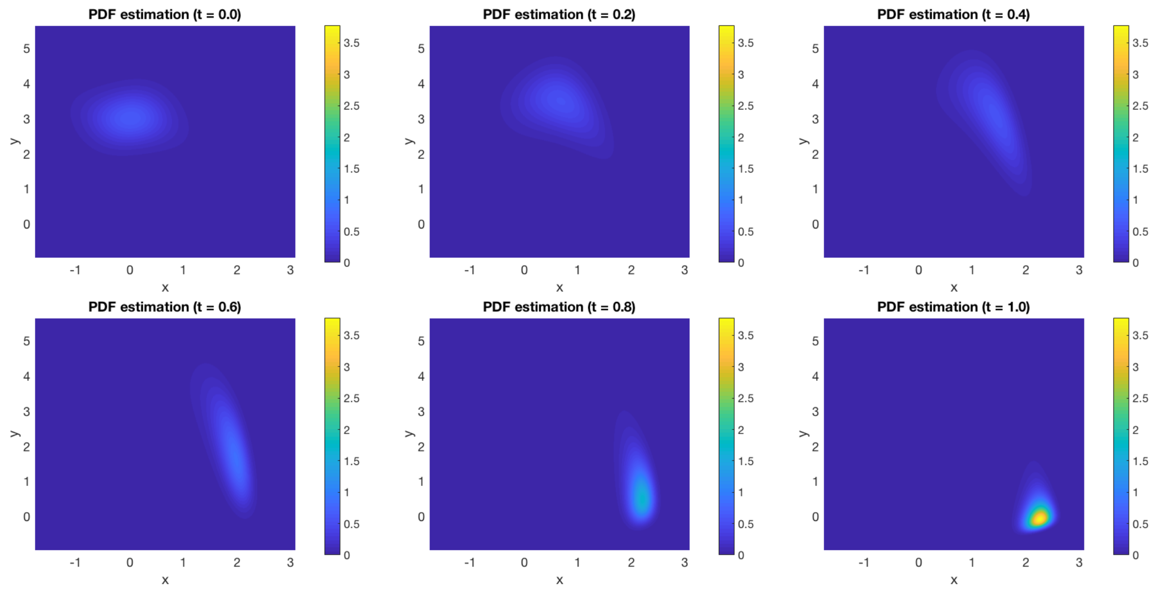


Figure 27: Time evolution of $p(x, y, t)$ for the Van der Pol oscillator (45) where the initial PDF is the product of two independent Gaussians with means $\mu_x = 0$, $\mu_y = 3$ and variances $\sigma_x^2 = \sigma_y^2 = 0.25$. Here the net was trained with data in the interval $t_0 = 0.0$ to $t_f = 1.0$.

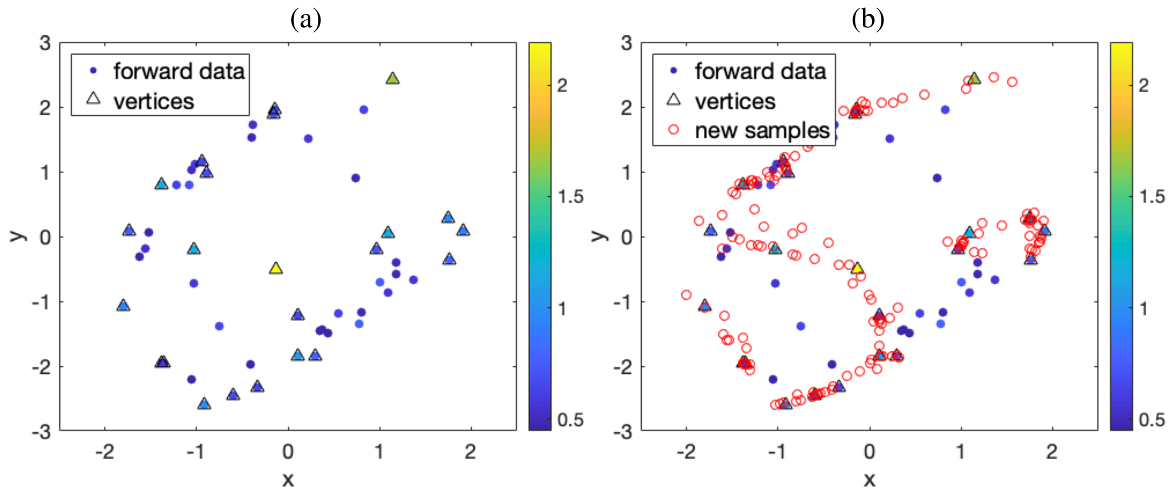


Figure 28: Convex hull stratification algorithm performed on $N_s = 50$ forward samples of the Van der Pol oscillator with $t_f = 4$: (a) forward samples with vertices picked out, color coded by L_2 weight norms; (b) as left, with additional backward samples.

forced Duffing oscillator with random parameters and random initial condition:

$$\begin{cases} \dot{x}_1 = x_2 \\ \dot{x}_2 = -\delta x_2 - x_1\left(\alpha + \beta x_1^2\right) + \gamma\cos(\omega t) \\ \dot{\delta} = 0 \\ \dot{\alpha} = 0 \\ \dot{\beta} = 0 \\ \dot{\gamma} = 0 \\ \dot{\omega} = 0 \end{cases} \qquad \begin{cases} x_1(t=0) \sim \mathcal{N}(0,1) \\ x_2(t=0) \sim \mathcal{N}(0,1) \\ \delta \sim \mathcal{N}(0.5, 0.25^2) \\ \alpha \sim \mathcal{N}(-1, 0.25^2) \\ \beta \sim \mathcal{N}(1, 0.25^2) \\ \gamma \sim \mathcal{N}(1, 0.25^2) \\ \omega \sim \mathcal{N}(0.5, 0.25^2) \end{cases} \tag{47}$$

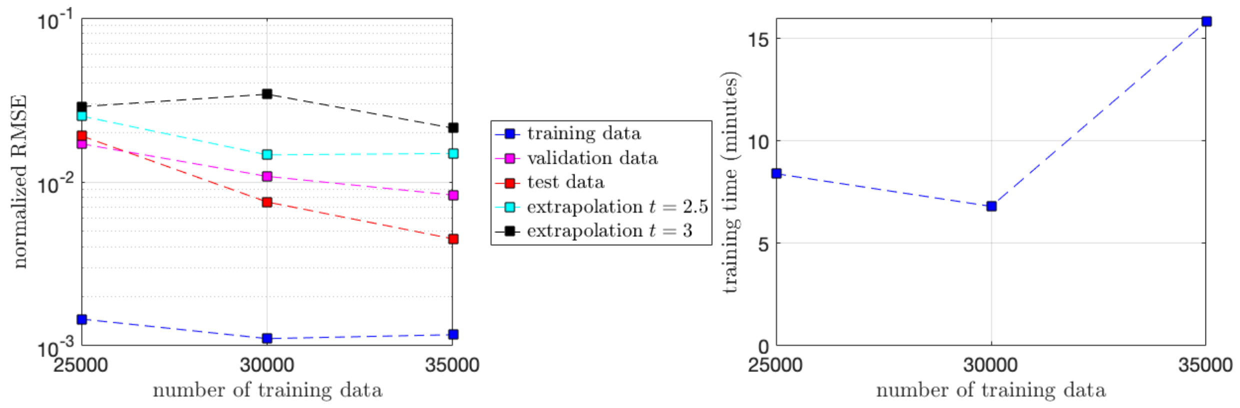


Figure 29: Error and training time for different data sets.

Equation (47) is, in effect, a seven-dimensional dynamical system. We use a physics-informed neural net to approximate the time-dependent PDF for $t \in [0, 2]$, and test its ability to extrapolate to $t_f = 2.5$ and $t_f = 3$. The neural net has eight hidden layers with 64 neurons each, and is trained on an NVIDIA RTX 2080Ti GPU (this hardware upgrade was necessary to facilitate larger neural nets and data sets). In addition to the training data set, we generate a validation set and three test data sets with $t_f = 2$, 2.5, and 3. The test data set with $t_f = 2$ takes the place of our former validation data set, i.e. we use this to measure the accuracy of the net within the training domain. The test sets with $t_f = 2.5$ and $t_f = 3$ include time snapshots both in the training region $t \in [0, 2]$ and outside of it, thus allowing us to quantify how well the neural net extrapolates forward in time. Finally, the new validation data set is used during training to measure the error in generalization: if we change the Liouville penalty weight μ_E, the difference in validation and training error can inform the choice of μ_{E+1}. Moreover, we use change in validation error as a stopping condition: once the difference between training rounds reaches some small threshold, we stop training as there is typically no improvement to be gained by continuing. We use the same neural net architecture and vary the number of sample trajectories (and total size of the training set). Other data sets (validation, testing, extrapolation ($t_f = 2.5$) testing, and extrapolation ($t_f = 3.0$) testing) are fixed among all trials:

- $N_{\text{val}} = 10000$ validation data points from 449 total sample trajectories,
- $N_{\text{test}} = 10000$ test data points from 460 total sample trajectories,
- $N_{\text{ext1}} = 10000$ extrapolation ($t_f = 2.5$) test data points from 448 total sample trajectories,
- $N_{\text{ext2}} = 10000$ extrapolation ($t_f = 3$) test data points from 487 total sample trajectories.

Note that the time snapshots are for these additional data sets are different than those for the training data set. Our numerical results are shown in Figure 29. In Figure 30 we plot the temporal evolution of the two-dimensional conditional PDF that solves the stochastic Duffing equation (47).

2.2.8.2 Fixed-wing unmanned aerial vehicle (UAV) model Next we consider a more challenging problem, i.e., computing the flow map of the fixed-wing UAV model proposed in [59, 60]. Such model represents

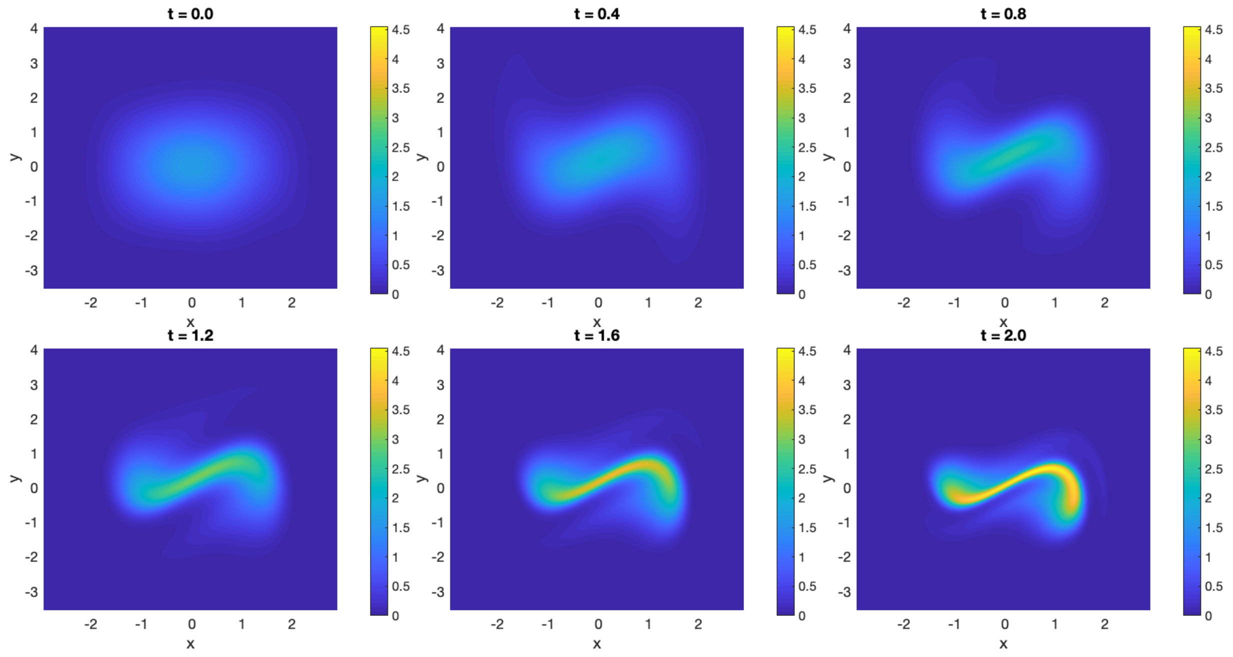


Figure 30: Neural network approximation of the conditional PDF $\hat{p}(x, y, t|\delta = 0.5, \alpha = -1, \beta = 1, \omega = 1, \gamma = 0.5)$ based on samples of the stochastic Duffing equation (47).

the UAV as a point-mass, but it also accounts for side-slip and thrust. The nonlinear system of equations is

$$\text{(UAV model)} \quad \begin{cases} \dot{x} = v\cos\gamma\cos\sigma \\ \dot{y} = v\cos\gamma\sin\sigma \\ \dot{z} = v\sin\gamma \\ \dot{v} = \dfrac{1}{m}(-D + T\cos\alpha) - g\sin\gamma \\ \dot{\gamma} = \dfrac{1}{mv}(L\cos\mu + T\cos\mu\sin\alpha) - \dfrac{g}{v}\cos\gamma \\ \dot{\sigma} = \dfrac{1}{mv\cos\gamma}(L\sin\mu + T\sin\mu\sin\alpha) \\ \dot{T} = u_T \\ \dot{\alpha} = u_\alpha \\ \dot{\mu} = u_\mu \end{cases} \tag{48}$$

where (x, y, z) is the position in a flat earth reference frame, v is the velocity, (γ, σ) are elevation and heading angles, T is the thrust, α is the angle of attack, μ is the bank angle (see Figure 31). The three controls, u_T, u_α, and u_μ here are pre-computed from a deterministic minimum-time path planning problem between the points $(x_0, y_0, z_0) = (0, 0, 600)$ to $(x(t_f), y(t_f), z(t_f)) = (1000, 1000, 600)$ subject to constraints. The other parameters are $m = 2$ (the UAV mass), $g = 9.8$ gravitational acceleration, while L and D are lift and drag forces given by

$$L = \frac{1}{2}\rho v^2 S C_L, \qquad D = \frac{1}{2}\rho v^2 S C_D, \tag{49}$$

where $\rho = 1.21e^{-z/8000}$ is the mass density of air, $S = 0.982$ is the wing area, and C_L and C_D are the lift

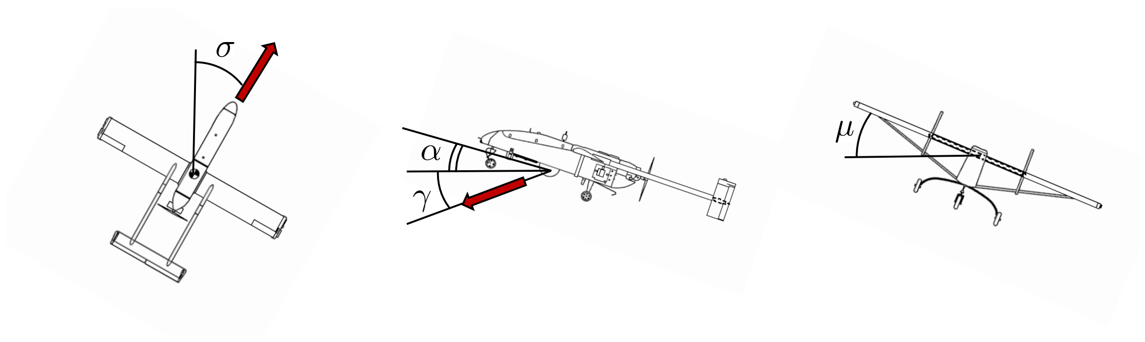


Figure 31: Model of an Unmanned Aerial Vehicle (UAV).

and drag coefficients. These are calculated by

$$C_L = (C_{x0} + C_{xa}\alpha)\sin\alpha - (C_{z0} + C_{za}\alpha)\cos\alpha, \tag{50}$$

$$C_D = -(C_{x0} + C_{xa}\alpha)\cos\alpha - (C_{z0} + C_{za}\alpha)\sin\alpha. \tag{51}$$

Note that C_L and C_D depend on the parameters C_{x0}, C_{xa}, C_{z0}, and C_{za} which are usually unknown and must be estimated from data. Thus, we assume these have given probability distributions, and augment the system (48) with

$$\dot{C}_{x0} = \dot{C}_{xa} = \dot{C}_{z0} = \dot{C}_{za} = 0. \tag{52}$$

This system we obtain in this way effectively has 13 dimensions plus time, and it is significantly more complicated than the Duffing equation (47). We introduce uncertainty into all of the initial conditions. For simplicity we let each variable be independently Gaussian, with

$$\begin{cases} x_0 \sim \mathcal{N}(0, 5^2) \\ y_0 \sim \mathcal{N}(0, 5^2) \\ z_0 \sim \mathcal{N}(600, 1^2) \\ v_0 \sim \mathcal{N}(27.5, 0.1^2) \\ \gamma_0 \sim \mathcal{N}(0, 0.001^2) \\ \sigma_0 \sim \mathcal{N}(0, 0.001^2) \\ T_0 \sim \mathcal{N}(16.1, 0.01^2) \\ \alpha_0 \sim \mathcal{N}(-0.0088, 0.0001^2) \\ \mu_0 \sim \mathcal{N}(0, 0.001^2) \\ C_{x0} \sim \mathcal{N}(-0.03554, 0.001^2) \\ C_{xa} \sim \mathcal{N}(0.00292, 0.0001^2) \\ C_{z0} \sim \mathcal{N}(-0.055, 0.001^2) \\ C_{za} \sim \mathcal{N}(-5.578, 0.01^2). \end{cases} \tag{53}$$

The preliminary numerical result we obtained so far with neural net approximations of the 13-dimensional joint PDF using both standard deep neural nets and physics informed neural nets, are not satisfactory. This appears to be caused by the extremely peaked density in some regions of the phase space. In principle, we could artificially reduce the density by increasing the initial uncertainty (to potentially physically improbable values). However, we found that this can yield integration failure: in fact it can happen that $\cos\gamma \to 0$ in (48) (i.e., the UAV flies straight up or down), and thus $\dot{\sigma}$ becomes singular. Hence to solve this problem we turned to our flow map approximators (see Section 2.2.10).

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

$$\mathbb{E}\left\{x_1(t)x_3(s)\right\} = \int_{-\infty}^{\infty} \Phi_1(\boldsymbol{x}_0, t)\Phi_3(\boldsymbol{x}_0, s)p_0(\boldsymbol{x}_0)d\boldsymbol{x}_0. \tag{54}$$

On the other hand, the solution to the Liouville equation (30) 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.

2.2.9.1 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.

- **"Jump" flow map estimator.** The first possibility is to learn it exactly in the form defined in equation (26), 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 the inverse flow map. Recall that evaluating eq. (30) 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 $\boldsymbol{x}_0$.

2.2.9.2 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 32. If we use the physics-informed

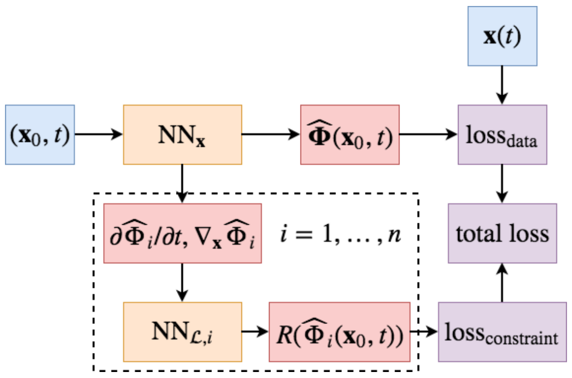

$\mathbf{x}(t)$
$(\mathbf{x}_0, t)$
$\mathrm{NN}_{\mathbf{x}}$
$\widehat{\boldsymbol{\Phi}}(\mathbf{x}_0, t)$
$\mathrm{loss}_{\mathrm{data}}$
$\partial\widehat{\Phi}_i/\partial t, \nabla_{\mathbf{x}}\widehat{\Phi}_i$
$i = 1, \ldots, n$
total loss
$\mathrm{NN}_{\mathcal{L},i}$
$R(\widehat{\Phi}_i(\mathbf{x}_0, t))$
$\mathrm{loss}_{\mathrm{constraint}}$

Figure 32: 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 (27).

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:

$$\mathrm{loss}_{\mathrm{fwd}}(\boldsymbol{\theta}; E) = \mathrm{loss}_{\mathrm{data}}(\boldsymbol{\theta}) + \mu_E \mathrm{loss}_{\mathrm{constraint}}(\boldsymbol{\theta}), \tag{55}$$

where $\boldsymbol{\theta}$ are the shared parameters of both nets and μ_E is a weight which depends on the training epoch E. Here the training data loss is a simple mean square error loss given by

$$\mathrm{loss}_{\mathrm{data}} = MSE_{\mathrm{data}}(\boldsymbol{\theta}) = \frac{1}{N_d} \sum_{j=1}^{N_d} \left[\boldsymbol{x}^{(j)} \left(t^{(j)} \right) - \hat{\boldsymbol{\Phi}} \left(\boldsymbol{x}_0^{(j)}, t^{(j)} \right) \right]^2, \tag{56}$$

or a "logcosh" loss given by

$$\mathrm{loss}_{\mathrm{data}} = \mathrm{logcosh}_{\mathrm{data}}(\boldsymbol{\theta}) = \frac{1}{N_d} \sum_{j=1}^{N_d} \log \left[\cosh \left(\boldsymbol{x}^{(j)} \left(t^{(j)} \right) - \hat{\boldsymbol{\Phi}} \left(\boldsymbol{x}_0^{(j)}, t^{(j)} \right) \right) \right]. \tag{57}$$

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:

$$\mathrm{loss}_{\mathrm{constraint}}(\boldsymbol{\theta}; E) = \sum_{i=1}^{n} \frac{1}{N_{c,E}} \sum_{j=1}^{N_{c,E}} \left[R \left(\hat{\Phi}_i \left(\boldsymbol{x}_0^{(j)}, t^{(j)} \right) \right) \right]^2, \tag{58}$$

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 (27), i.e. $R(\widehat{\Phi}_i) = \partial\widehat{\Phi}_i/\partial t - \boldsymbol{G} \cdot \nabla\widehat{\Phi}_i$, 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.

2.2.9.3 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 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

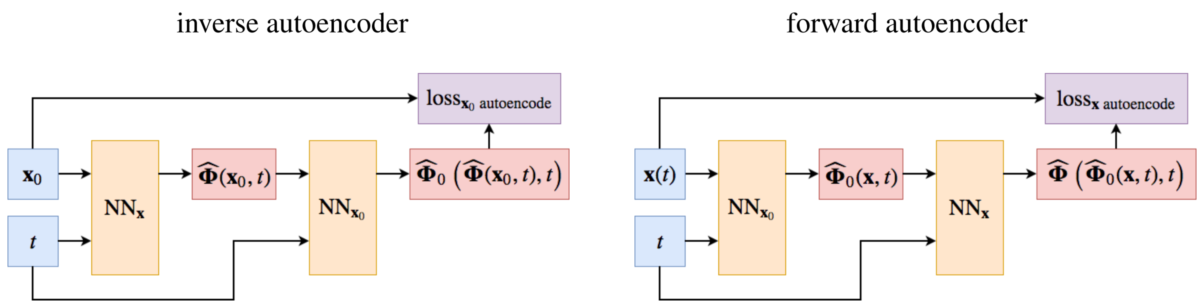

inverse autoencoder
forward autoencoder
$\text{loss}_{\mathbf{x}_0 \text{ autoencode}}$
$\text{loss}_{\mathbf{x} \text{ autoencode}}$
$\mathbf{x}_0$
t
$\text{NN}_{\mathbf{x}}$
$\widehat{\boldsymbol{\Phi}}(\mathbf{x}_0, t)$
$\text{NN}_{\mathbf{x}_0}$
$\widehat{\boldsymbol{\Phi}}_0\left(\widehat{\boldsymbol{\Phi}}(\mathbf{x}_0, t), t\right)$
$\mathbf{x}(t)$
$\widehat{\boldsymbol{\Phi}}_0(\mathbf{x}, t)$
$\widehat{\boldsymbol{\Phi}}\left(\widehat{\boldsymbol{\Phi}}_0(\mathbf{x}, t), t\right)$

Figure 33: Diagrams representing the inverse (eq. (60)) and forward (eq. (61)) autoencoders for flow map approximation.

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 (55) 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

$$\begin{aligned}\text{loss}_{\text{total}}(\boldsymbol{\theta}_{\text{fwd}}, \boldsymbol{\theta}_{\text{inv}}; E) &= \text{loss}_{\text{fwd}}(\boldsymbol{\theta}_{\text{fwd}}; E) + \lambda_1 \cdot \text{loss}_{\text{inv}}(\boldsymbol{\theta}_{\text{inv}}; E) \\ &\quad + \lambda_2 \cdot \text{loss}_{\boldsymbol{x}_0 \text{ autoencode}}(\boldsymbol{\theta}_{\text{fwd}}, \boldsymbol{\theta}_{\text{inv}}) + \lambda_3 \cdot \text{loss}_{\boldsymbol{x} \text{ autoencode}}(\boldsymbol{\theta}_{\text{fwd}}, \boldsymbol{\theta}_{\text{inv}}).\end{aligned} \tag{59}$$

Here we denoted by $\boldsymbol{\theta}_{\text{fwd}}$, $\boldsymbol{\theta}_{\text{inv}}$ and loss_{fwd}, loss_{inv} as the parameters and loss functions (see eq. 55) of the forward and inverse flow map approximators, respectively. Further, we let λ_1, λ_2, and λ_3 be constant scalar weights, and

$$\text{loss}_{\boldsymbol{x}_0 \text{ autoencode}}(\boldsymbol{\theta}_{\text{fwd}}, \boldsymbol{\theta}_{\text{inv}}) = \frac{1}{N_d} \sum_{j=1}^{N_d} \left[\boldsymbol{x}_0^{(j)} - \hat{\boldsymbol{\Phi}}_0\left(\hat{\boldsymbol{\Phi}}\left(\boldsymbol{x}_0^{(j)}, t^{(j)}\right), t^{(j)}\right)\right]^2. \tag{60}$$

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

$$\text{loss}_{\boldsymbol{x} \text{ autoencode}}(\boldsymbol{\theta}_{\text{fwd}}, \boldsymbol{\theta}_{\text{inv}}) = \frac{1}{N_d} \sum_{j=1}^{N_d} \left[\boldsymbol{x}^{(j)} - \hat{\boldsymbol{\Phi}}\left(\hat{\boldsymbol{\Phi}}_0\left(\boldsymbol{x}^{(j)}, t^{(j)}\right), t^{(j)}\right)\right]^2. \tag{61}$$

Figure 33 helps make these loss terms more intuitive. Note that eqs. (60) and (61) are for the "jump" flow map estimator – analogous terms for the "step" estimator can be easily defined.

2.2.9.4 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 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` [28, 33] 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 $\{\boldsymbol{x}^{(j)}(t_f)\}$, $j = 1, \ldots, N_s^b$. 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 (27) 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 (28) 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.

2.2.9.5 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[3]

$$\frac{\partial \Phi_i(\boldsymbol{x}(t), t + \Delta t)}{\partial t} - \boldsymbol{G}(\boldsymbol{x}(t), t + \Delta t) \cdot \nabla \Phi_i(\boldsymbol{x}(t), t + \Delta t) = 0. \tag{62}$$

Similarly, the components of the inverse flow map satisfy

$$\frac{\partial \Phi_i^{-1}(\boldsymbol{x}(t + \Delta t), t)}{\partial t} + \boldsymbol{G}(\boldsymbol{x}(t + \Delta t), t) \cdot \nabla \Phi_i^{-1}(\boldsymbol{x}(t + \Delta t), t) = 0. \tag{63}$$

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 $t_0, t_1, \ldots, t_{N_t - 1}$.

[3]Note that equations (62) and (63) hold for non-autonomous systems of the form $\dot{\boldsymbol{x}} = \boldsymbol{G}(\boldsymbol{x}, t)$, $\boldsymbol{x}(0) = \boldsymbol{x}_0$.

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

$$NRMSE_{\text{val}} = \frac{\sqrt{\frac{1}{N_v}\sum_{j=1}^{N_v}\sum_{i=1}^{n}\left[x_i^{(j)}\left(t^{(j)}+\Delta t\right)-\widehat{\Phi}_i^{(j)}\left(x_i^{(j)}\left(t^{(j)}\right),t^{(j)}\right)\right]^2}}{\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left[\max_j x_i^{(j)}\left(t^{(j)}+\Delta t\right)-\min_j x_i^{(j)}\left(t^{(j)}+\Delta t\right)\right]^2}}. \tag{64}$$

We tested our algorithms by approximating the flow maps generated by the Van der Pol oscillator (45) 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.

2.2.9.6 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 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 (40), with $E_{\max} = 10$ iterations/epochs. All loss terms were modeled as mean square errors. In all trials, data generation took approximately 1 second. In Tables 15 and 16 we summarize results of the tests with $x_0, y_0 \in [-2, 2]$. In particular, results of Table 16 are obtained with physics-informed "jump" forward flow map estimators. Similarly, Tables 17 and 18 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 34 we plot the validation data and corresponding neural net predictions from one trial of each region of initial conditions.

2.2.9.7 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 (40)

Table 15: Accuracy and training time (single CPU implementation using TensorFlow 1.8 [1]) of a plain neural net for approximating the forward "jump" flow map of the Van der Pol oscillator (45) with $x_0, y_0 \in [-2, 2] \times [-2, 2]$.

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 16: Accuracy and training time of a PINN for approximating the forward "jump" flow map of the Van der Pol oscillator (45) with $x_0, y_0 \in [-2, 2] \times [-2, 2]$.

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

with $E_{\max} = 10$ iteration/epochs. Latin hypercube sampling created many outliers in the collocation data. We sought to blunt 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 [51]. For the data loss we still used MSE. In Tables 19 and 20 we summarize results of the tests we performed with $x_0, y_0 \in [-2, 2]$. In particular, Table 20 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 21 and 22 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 (58). 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.

2.2.9.8 "Jump" flow map estimator with bidirectional autoencoder results We conducted one set

Table 17: Accuracy and training time of a plain neural net for approximating the forward "jump" flow map of the Van der Pol oscillator (45) with $x_0, y_0 \in [-1, 3] \times [-2, 2]$.

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 18: Accuracy and training time of a PINN for approximating the forward "jump" flow map of the Van der Pol oscillator (45) 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

of tests of the bidirectional autoencoder sketched in Figure 33 using no PINN, and another using a PINN enforcing the *forward* flow map equations (27) *only*, with $N_c = 5000$ collocation points. We didn't go through the trouble 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 (40) with $E_{\max} = 10$. 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 (59). Similarly, λ_2 and λ_3 are the weights on the $\boldsymbol{x}_0$ and $\boldsymbol{x}$ autoencoding terms, respectively. In Tables 23 and 24 we summarize the results of some of our tests with $x_0, y_0 \in [-2, 2]$. Tables 25 and 26 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.

2.2.9.9 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 27 contains results of the tests with $x_0, y_0 \in [-2, 2]$, and Table 28 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

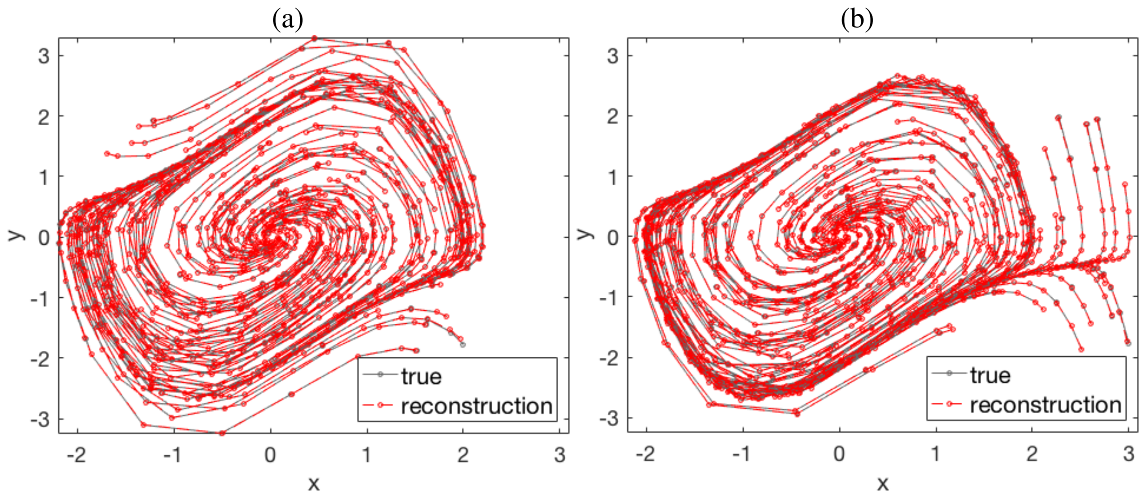


Figure 34: Validation data and corresponding neural net reconstructions by a PINN which approximates the forward "jump" flow map of the Van der Pol oscillator (45): (a) $x_0, y_0 \in [-2,2] \times [-2,2]$ from the model trained in trial 1 of Table 16; (b) $x_0, y_0 \in [-1,3] \times [-2,2]$ from the model trained in trial 1 of Table 18.

Table 19: Accuracy and training speed of a plain neural net for approximating the inverse "jump" flow map of the Van der Pol oscillator (45) with $(x_0, y_0) \in [-2,2] \times [-2,2]$.

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

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 35 we plot the validation data and corresponding neural net predictions from one trial of each region of initial conditions.

2.2.10 "Jump" flow map estimator for high-dimensional problems

To facilitate application of Jump flow map estimators to high dimensional problems with varying scales, we developed two improvements to our algorithms. First, for problems with parameter uncertainty we structured the neural net so that it would take parameter inputs, but not predict the value of the parameter at time t (passing the value of the parameter through directly). Secondly, we scaled the training data output $\boldsymbol{x}(t)$ to the domain $[-1,1]^n$ based on the hyper-cube bounds of the data. In this way, we were able to overcome difficulties related to the fact that training data for different variables can have different orders of magnitude.

Table 20: Accuracy and training speed of a PINN for approximating the inverse “jump” flow map of the Van der Pol oscillator (45) with $(x_0, y_0) \in [-2, 2] \times [-2, 2]$.

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 21: Accuracy and training speed of a plain neural net for approximating the inverse “jump” flow map of the Van der Pol oscillator (45) with $(x_0, y_0) \in [-1, 3] \times [-2, 2]$.

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

2.2.10.1 Numerical results: estimation of the fixed-wing UAV flow map We integrated 200 trajectories with initial conditions sampled in the hypercube defined by $\boldsymbol{\mu} \pm 2\boldsymbol{\sigma}$, where $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ are the means and standard deviations given in (53). We did not use backward sampling for this problem, since these integrations often failed to converge. We took outputs at all times t_k chosen by the adaptive time-stepping of SciPy’s `dopri5` [28, 33], giving us a total of 37665 data points. Validation and test data were generated from 100 forward samples each, yielding 18673 and 18882 total data points, respectively. The time interval was $t \in [0, 41.4353]$, where $t_f = 41.4353$ was the minimum time obtained for the deterministic optimal control problem [60]. We did not test for the ability to extrapolate, since the open-loop control was only defined in that time interval, however extrapolation testing could be done if we trained on data only up to some $t'_f < 41.4353$. For both the forward and inverse flow map, we used a plain deep neural net with four hidden layers of 64 neurons each, and varied the number of training data used in each training epoch. We set $N_{d,E} = E/E_{\text{max}}$ with $E_{\text{max}} = 10$ epochs, and stopped training early once the difference in validation NRMSE between rounds dropped below 10^{-4}.

- The forward jump approximator was trained in 313 seconds on an NVIDIA RTX 2080Ti GPU, and reached a final training NRMSE of 9.61 e–04, validation NRMSE of 9.72 e–04, and test NRMSE or 9.78 e–04. We visualize the output in Figure 36
- The inverse jump approximator was trained in 290 seconds on an NVIDIA RTX 2080Ti GPU, and reached a final training NRMSE of 2.51 e–03, validation NRMSE of 4.20 e–03, and test NRMSE or 3.90 e–03.

Table 22: Accuracy and training speed of a PINN for approximating the inverse "jump" flow map of the Van der Pol oscillator (45) 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	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 23: 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 (45) with $(x_0, y_0) \in [-2, 2] \times [-2, 2]$, trained as a bidirectional autoencoder.

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

2.3 Task II: Numerical methods to solve data-driven PDF equations

The physics-informed neural network technique we studied in Section 2.2.4.3 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 (25). We have seen that the Liouville equation (29) 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

$$u(\boldsymbol{x}) = \mathbb{R}^n \to \mathbb{R} \qquad \text{(observable).} \tag{65}$$

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

$$p(z,t) = \int_{-\infty}^{\infty} \cdots \int_{-\infty}^{\infty} \delta\left(z - u(\boldsymbol{x})\right) p(\boldsymbol{x},t) d\boldsymbol{x}, \tag{66}$$

where $\delta(\cdot)$ is the Dirac's delta function (see [40, 65, 53]) and z is a random variable representing the value of $u(\boldsymbol{x}(t))$. Multiplying the Liouville equation (29) by $\delta\left(z - u(\boldsymbol{x})\right)$ and integrating over all phase variables yields

$$\frac{\partial p(z,t)}{\partial t} + \int_{-\infty}^{\infty} \cdots \int_{-\infty}^{\infty} e^{ia(z-u(\boldsymbol{x}))} \nabla \cdot \left(\boldsymbol{G}(\boldsymbol{x}) p(\boldsymbol{x},t)\right) \boldsymbol{x} da = 0. \tag{67}$$

Table 24: 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 (45) with $(x_0, y_0) \in [-2, 2] \times [-2, 2]$, trained as a bidirectional autoencoder.

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 25: 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 (45) with $(x_0, y_0) \in [-1, 3] \times [-2, 2]$, trained as a bidirectional autoencoder.

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

Here we employed the Fourier representation of the Dirac delta function. In general, equation (67) 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 (25) then (67) reduces to[4]

$$\frac{\partial p(x_k, t)}{\partial t} + \int_{-\infty}^{\infty} \cdots \int_{-\infty}^{\infty} \frac{\partial}{\partial x_k} \left(G_k(\boldsymbol{x}) p(\boldsymbol{x}, t) \right) dx_1 \dots dx_{k-1} dx_{k+1} \dots dx_n = 0. \tag{68}$$

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

$$\frac{dx_i}{dt} = (x_{i+1} - x_{i-2}) x_{i-1} - x_i + F \qquad \text{for} \quad i = 1, 2, \dots, n, \tag{69}$$

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

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

Table 26: 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 (45) with $(x_0, y_0) \in [-1, 3] \times [-2, 2]$, trained as a bidirectional autoencoder.

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 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 (45) with $(x_0, y_0) \in [-2, 2] \times [-2, 2]$.

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

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

$$\frac{\partial p(\boldsymbol{x}, t)}{\partial t} = -\sum_{i=1}^{d} \frac{\partial}{\partial x_i} \left[\Big((x_{i+1} - x_{i-2})x_{i-1} - x_i + F \Big) p(\boldsymbol{x}, t) \right]. \tag{70}$$

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 (65). By integrating (70) 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

$$\frac{\partial p(x_5, t)}{\partial t} = \frac{\partial}{\partial x_5} \left[\left(x_5 - F + \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} (x_3 - x_6) x_4 p(x_3, x_4, x_5, x_6, t) dx_3 dx_4 dx_6 \right. \right]. \tag{71}$$

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

$$p(x_3, x_4, x_5, x_6, t) = p(x_5, t) p(x_3, x_4, x_6 | x_5, t), \tag{72}$$

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 $x_1(t)$ [9, 53]. A substitution of (72) into (71) yields

$$\frac{\partial p(x_5, t)}{\partial t} = \frac{\partial}{\partial x_5} \left[\Big(x_5 - F + \mathbb{E}\{(x_3 - x_6) x_4 | x_5, t\} \Big) p(x_5, t) \right], \tag{73}$$

Table 28: Accuracy and training time of a plain feed-forward neural net for approximating the forward "step" flow map of the Van der Pol oscillator (45) with $(x_0, y_0) \in [-1, 3] \times [-2, 2]$.

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

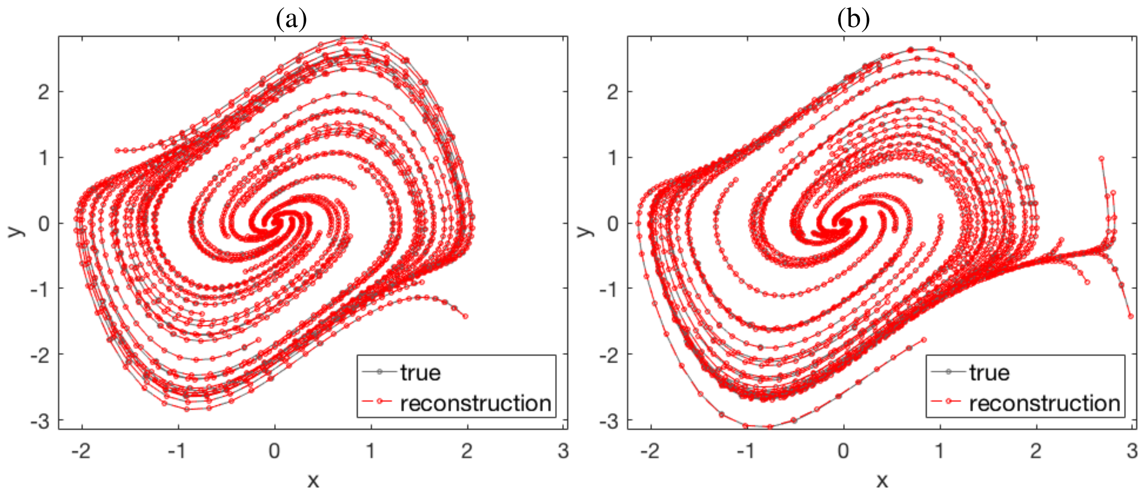


Figure 35: Validation data and corresponding neural net reconstructions by a plain neural net which approximates the forward "step" flow map of the Van der Pol oscillator (45): (a) $(x_0, y_0) \in [-2, 2] \times [-2, 2]$ from the model trained in trial 1 of Table 27; (b) $(x_0, y_0) \in [-1, 3] \times [-2, 2]$ from the model trained in trial 1 of Table 28.

where

$$\mathbb{E}\{(x_3 - x_6)x_4|x_5\} = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}(x_3 - x_6)x_4 p(x_3, x_4, x_6|x_5, t)dx_3dx_4dx_6 \tag{74}$$

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

$$\begin{cases} \dot{x} = 2xy - 1, \\ \dot{y} = -x^2 - y^2 + \mu. \end{cases} \tag{75}$$

The Liouville equation associated with (75) is

$$\frac{\partial p(x, y, t)}{\partial t} = -\frac{\partial}{\partial x}\big((2xy - 1)p(x, y, t)\big) - \frac{\partial}{\partial y}\big((-x^2 - y^2 + \mu)p(x, y, t)\big). \tag{76}$$

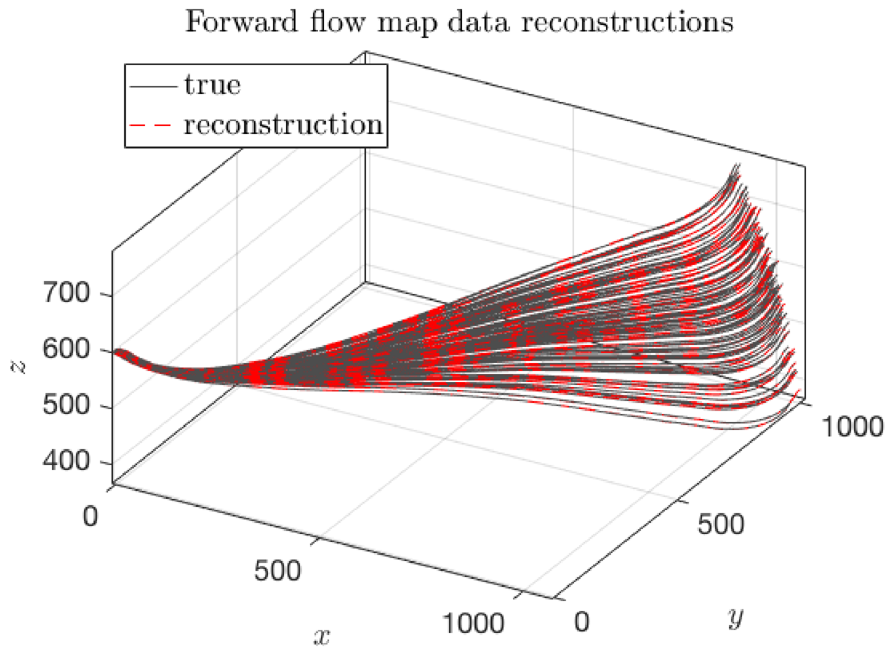


Figure 36: Reconstruction of 100 sample trajectories of the 13-dimensional fixed-wing UAV model (48) with initial condition and parameter uncertainty specified in (53).

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 (76) with respect to y. This yields,

$$\frac{\partial p(x,t)}{\partial t} = -\frac{\partial}{\partial x}\left(2x\int_{-\infty}^{\infty} yp(x,y,t)dy - p(x,t)\right). \tag{77}$$

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 (77) as a conditional expectation

$$\int_{-\infty}^{\infty} yp(x,y,t)dy = p(x,t)\int_{-\infty}^{\infty} yp(y|x,t)dy = p(x,t)\mathbb{E}\{y(t)|x(t)\}. \tag{78}$$

Equation (77) can now be rewritten in terms of $\mathbb{E}\{y|x, t\}$

$$\frac{\partial p(x,t)}{\partial t} = -\frac{\partial}{\partial x}\left(2xp(x,t)\mathbb{E}\{y(t)|x(t)\} - p(x,t)\right). \tag{79}$$

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 (25), then we need to express the right hand side of (68) 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.,

$$G_k(\boldsymbol{x}) = \sum_{l=1}^{r}\prod_{j=1}^{n} f_{kl}^{j}(x_j), \tag{80}$$

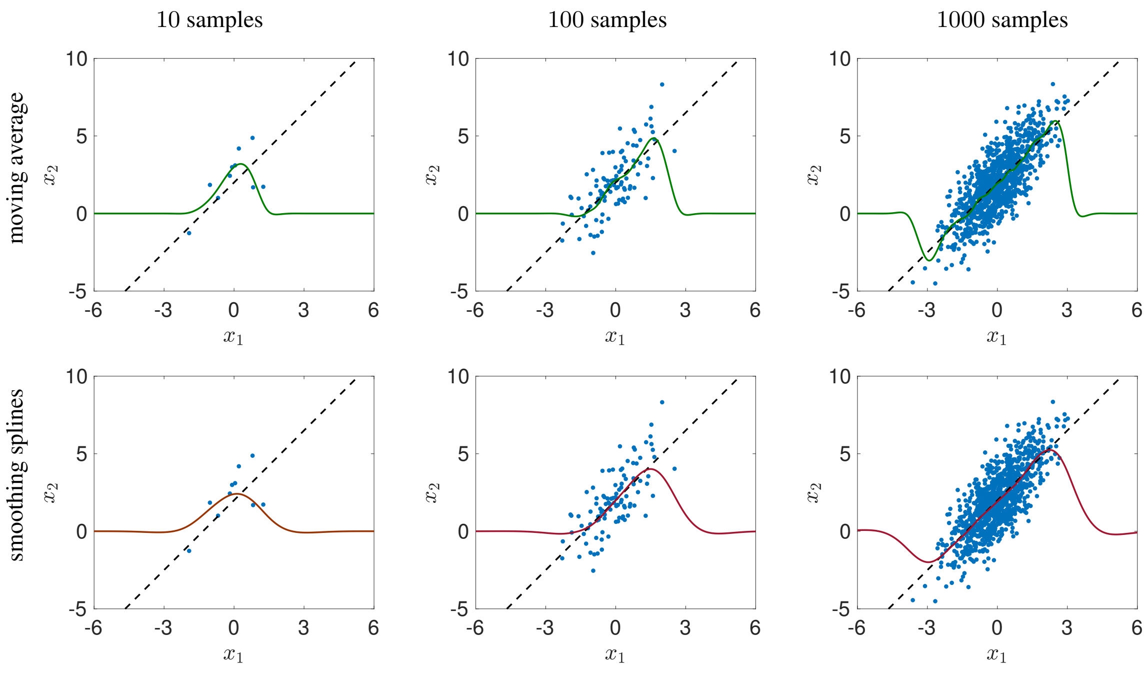


Figure 37: Numerical estimation of the conditional expectation (84) for different number of samples of (82). Shown are results obtained with moving averages and cubic smoothing splines.

then we can explicitly write (68) as

$$\frac{\partial p(x_k,t)}{\partial t} + \frac{\partial}{\partial x_k}\left(p(x_k,t) \sum_{l=1}^{r} f_{kl}^{k}(x_k) \mathbb{E}\left\{ f_{kl}^{1}(x_1)...f_{kl}^{k-1}(x_{k-1}) f_{kl}^{k+1}(x_{k+1})...f_{kl}^{n}(x_n) \middle| x_k \right\} \right) = 0. \quad (81)$$

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: 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 [15]. The choice of smoothing parameter is critical to the accuracy of the results. Specifying the smoothing parameter a priori is generally yields poor estimates [56]. Instead, cross-validation and maximum likelihood estimators can guide the choice the optimal smoothing value for the data set [66]. 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 [25]. In Figure 37 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

$$p(x_1,x_2) = \frac{1}{2\pi\sigma_1\sigma_2\sqrt{1-\rho^2}}\exp\left(-\frac{1}{2(1-\rho^2)}\left[\frac{(x_1-\mu_1)}{\sigma_1^2}\frac{(x_2-\mu_2)}{\sigma_2^2}-\frac{2\rho(x_1-\mu_1)(x_2-\mu_2)}{\sigma_1\sigma_2)}\right]\right) \tag{82}$$

with parameters $\rho = 3/4$, $\mu_1 = 0$, $\mu_2 = 2$, $\sigma_1 = 1$ $\sigma_2 = 2$. As is well known [53], given two random variables with joint PDF $p(x_1,x_2)$, the conditional expectation of x_2 given x_1 is defined as

$$\mathbb{E}\{x_2|x_1\} = \int_{-\infty}^{\infty} x_2 p(x_2|x_1)dx_2 = \frac{1}{p(x_1)}\int_{-\infty}^{\infty} x_2 p(x_1,x_2)dx_2, \tag{83}$$

where $p(x_1)$ is the marginal of $p(x_1,x_2)$ with respect to x_2. In the specific case of (82) we have

$$\mathbb{E}\{x_2|x_1\} = \mu_2 + \rho\frac{\sigma_2}{\sigma_1}(x_1-\mu_1) = 2 + \frac{3}{2}x_1. \tag{84}$$

Such conditional expectation is plotted in Figure 37 (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 (82) 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 (84) in regions where the joint PDF is very small [12]. 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 [13]. 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 (81), 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 (81).

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 38.

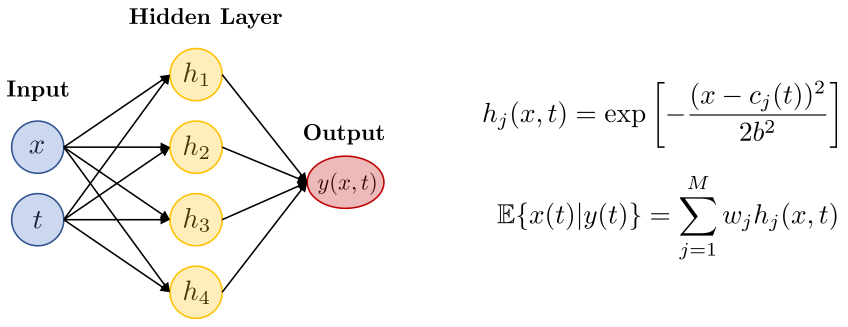

Hidden Layer
Input
Output
x
t
h_1
h_2
h_3
h_4
$y(x,t)$
$h_j(x,t) = \exp\left[-\frac{(x-c_j(t))^2}{2b^2}\right]$
$\mathbb{E}\{x(t)|y(t)\} = \sum_{j=1}^{M} w_j h_j(x,t)$

Figure 38: 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 $\mathbb{E}[x(t)|y(t)]$.

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 $m = 1, \dots, M$. 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
$$h_m(\boldsymbol{x}, t) = \exp\left(-\frac{\|\boldsymbol{x}(t) - \boldsymbol{c}_m(t)\|^2}{2b^2}\right), \tag{85}$$
where b is the *spread* (or variance) of the basis function.

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 (79)) then the output layer takes the form
$$\mathbb{E}\{y(t)|x(t)\} = \sum_{m=1}^{M} w_m h_m(x,t). \tag{86}$$

The spread of the radial basis neural network, i.e., the parameter b in (85), 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 (75) 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 39, 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 39, 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 (77), while smoothing splines and moving average regression methods require 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

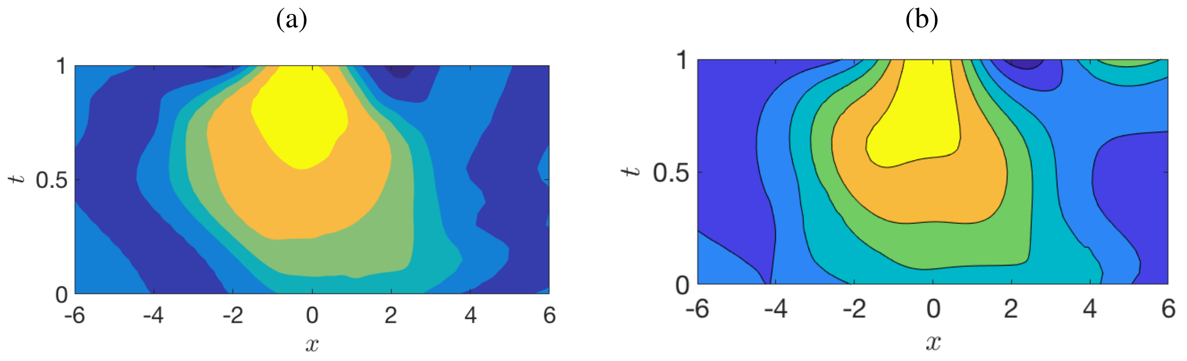


Figure 39: Approximate $\mathbb{E}\{y(t)|x(t)\}$ of dynamical system, equation (75), (a) using methods from 2.3.1 with 5000 trajectories, and (b) using a radial basis network with 25 neurons in the hidden layer and 500 training samples.

Table 29: Training time of a neural net for approximating the unclosed term in (77), depending on the availability of data.

		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

demonstrated in Table 29. 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.

2.3.2.1 Training the neural net To initialize training of the neural net, we need sample trajectories, e.g., of the system (75). 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

$$\boldsymbol{\Phi}_{ij}^C = \exp\left(-\frac{||\boldsymbol{x}_i - \boldsymbol{c}_j||_2^2}{2b^2}\right). \tag{87}$$

The transfer matrix satisfies

$$\boldsymbol{\Phi}^C \boldsymbol{w} + \boldsymbol{\beta} = \boldsymbol{y}, \tag{88}$$

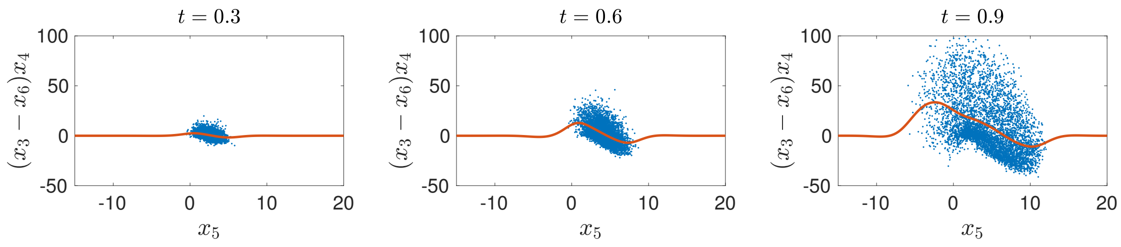


Figure 40: Data-driven smoothing spline estimation of the conditional expectations arising in the study of the Lorenz-96 dynamical system (69).

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 88 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.

2.3.3 Numerical results

2.3.3.1 Lorenz-96 system In Figure 40, we summarize the results we obtained by applying the smoothing spline conditional expectation estimator to the Lorenz-96 model introduced in (69). 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 41 we plot the PDF dynamics we obtain by solving (71) with an accurate Fourier spectral method. The conditional expectation is estimated based on 5000 sample trajectories.

2.3.3.2 Divergence-free system (75) In Figure 42, we plot the PDF dynamics we obtained by sampling (75), and then solving (79) 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 43, we plot the PDF dynamics we obtained by solving (79) with the conditional expectation estimated using a radial basis neural network and $M = 25$ neurons in the hidden layer. The network was trained with 500 sample trajectories, i.e., *an order of magnitude less than the moving average case*.
In Table 30 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 (79), 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 $\mathbb{E}\{y(t)|x(t)\}$.

2.3.4 Data-driven PDF equations as optimization constraints

Next, we consider the problem of equation-driven PDF estimation using a constrined optimization framework. To demonstrate the proposed methods, we consider the simple divergence-free ODE system (37)

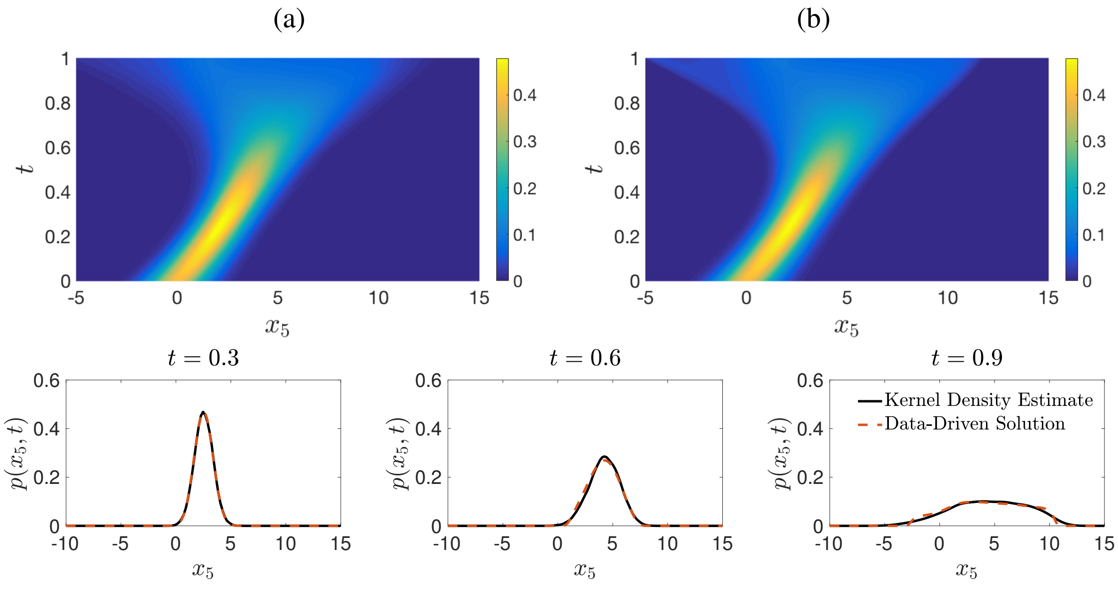


Figure 41: Estimate of the Lorenz-96 dynamical system using: (a) accurate kernel density estimate of $p_5(x_5, t)$ based on 2500 sample trajectories; and (b) numerical solution of (71) obtained by estimating $\mathbb{E}\left[\big(x_3(t) - x_6(t)\big)x_4(t)|x_5(t)\right]$ with 5000 sample trajectories.

Table 30: L^2 errors between the benchmark reduced-order PDF $p(x, t)$ at $t = 0.5$ in the system (75), and the one obtained from the reduced order equation (79), with $\mathbb{E}\{y(t)|x(t)\}$ estimated using a radial basis neural network.

		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

evolving from a random initial state, $p(x, y, t_0)$. The PDF of the state variables evolves according to the well-studied Liouville equation, (76). We obtain the marginal density function for x by integrating the joint density function with respect to y,

$$\frac{\partial p(x,t)}{\partial t} = -\frac{\partial}{\partial x}\Big(2xu(x,t)p(x,t) - p(x,t)\Big), \tag{89}$$

where $u(x, t) = \mathbb{E}[y|x, t]$. In this problem environment, the conditional expectation will act as a control signal for the PDE. Ultimately, we will design the control signal to steer the PDF in the direction of the sample data while agreeing with conditional data sampled from (37).

2.3.4.1 Discretization First, we will spatially discretize the PDE constraint using Fourier spectral methods. In this context, we seek solutions in the form $p_Q(x, t) = \sum_{k=0}^{Q} p_Q(x_k, t)\phi_k(x)$. This leads to a system

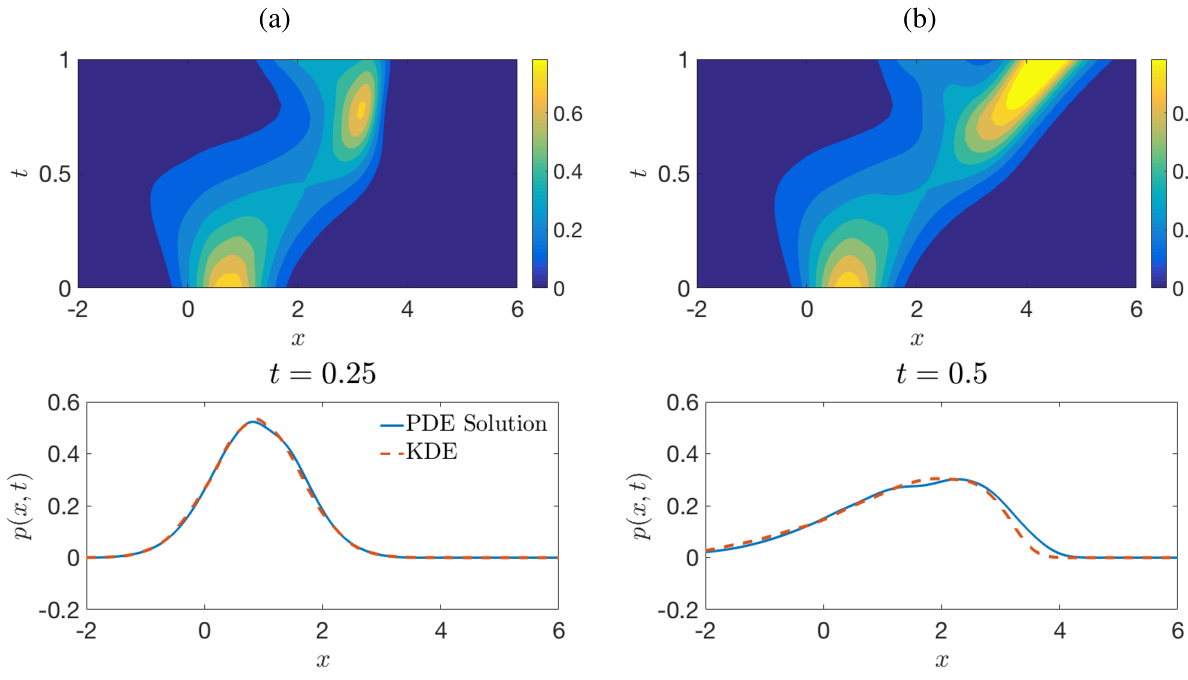


Figure 42: (a) Accurate kernel density estimate of $p(x,t)$ in equation (75), and (b) numerical solution of (79) where $\mathbb{E}\{y(t)|x(t)\}$ is estimated using the moving average regression method, as outlined in Section 2.3.1.

of Q ordinary differential equations defining p_Q.

$$\frac{\partial p_Q(x_j,t)}{\partial t} = -\sum_{k=1}^{Q} D_{jk}\Bigg(2x_k u(x_k,t)p_Q(x_k,t) - p_Q(x_k,t)\Bigg), \tag{90}$$

where D represents the $Q \times Q$ Fourier differentiation matrix. Next, we will discretize the system in time using an explicit third-order Adams-Bashforth scheme,

$$p(x_j,t_{n+1}) = p(x_j,t_n) + \frac{\Delta t}{12}\Bigg(23L(x_j,t_n) - 16L(x_j,t_{n-1}) + 5L(x_j,t_{n-2})\Bigg), \tag{91}$$

where $L(x_j,t_n) = -\sum_{k=1}^{Q} D_{jk}\Bigg(2x_k u(x_k,t_n)p(x_k,t_n) - p(x_k,t_n)\Bigg)$. This temporal discretization is desirable as it is consistent, strongly stable, and third-order accurate [18].

2.3.4.2 Cost Functional The cost functional at time t_n will be responsible for tracking two components: the PDF solution, $p(x,t_n)$, and the control signal, $u(x,t_n)$. We consider a cost functional in the form,

$$J[p(x,t_n),u(x,t_n)] = J_1[p(x,t_n),u(x,t_n)] + \big(u(x_i,t_n) - y_i\big)^2 + \lambda \int \frac{\partial^2 u(x,t_n)}{\partial x^2}\,dx. \tag{92}$$

The first term in our cost functional, $J_1[p(x,t_n),u(x,t_n)]$ represents a direct divergence approximation between the data-driven PDF estimator and sample trajectories of the underlying dynamical system. Several options for $J_1[p(x,t_n),u(x,t_n)]$ are discussed below. The second term in the cost functional determines the goodness of fit between our control, $u(x,t)$, and the data, $\{y_i\}_{i=1}^{M}$. The third and final term in J penalizes

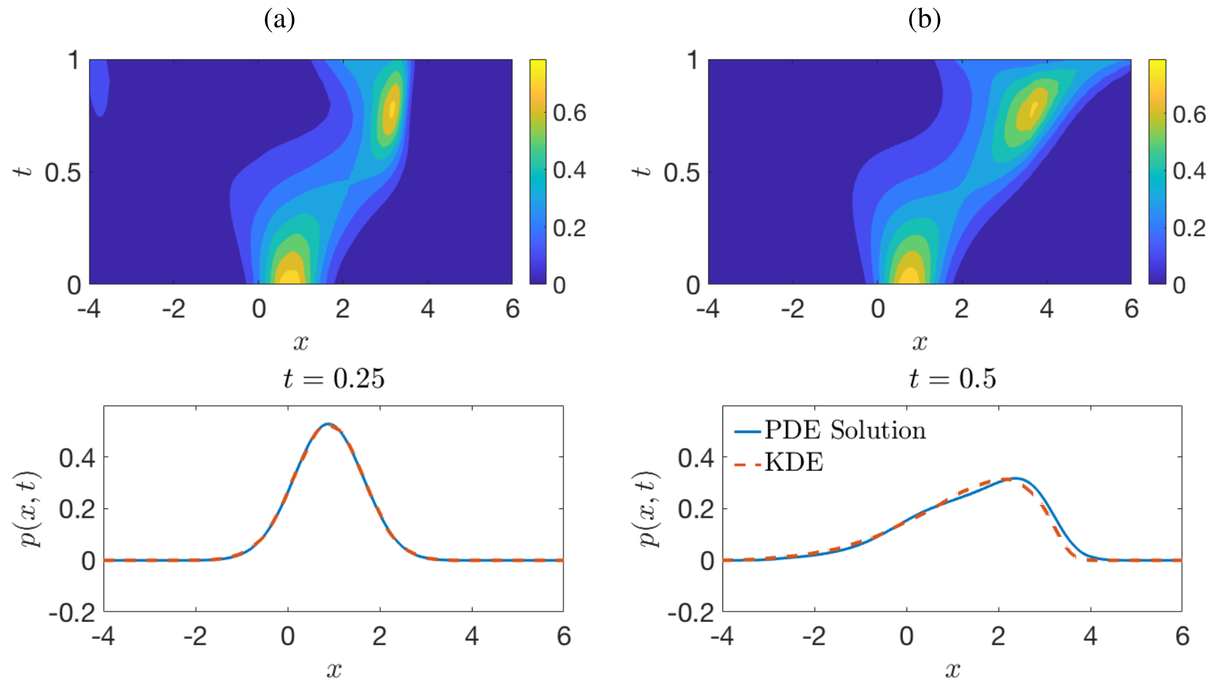


Figure 43: (a) Accurate kernel density estimate of $p(x,t)$ in equation (75), and (b) numerical solution of (79) where $\mathbb{E}\{y(t)|x(t)\}$ is estimated using the using a radial basis neural network.

roughness in the control, $u(x,t)$. Notice that the sum of the last two terms represents a cubic smoothing spline with the smoothing parameter λ determining the balance between smoothness and goodness-of-fit. Additionally, the time discretization of the problem ensures that we have an exceptional initial guess for t_n from iteration t_{n-1}. If we design J to be convex, the problem simiplifies and a unique minimum is guaranteed. There are many options for measuring the divergence between our PDF approximation and samples. We can rule out several criterion based on the problem setting. For example, the Akaike Information Criterion (AIC) can be problematic if we are working with a small sample size. In cases where the sample size is small, AIC is likely to select models with too many parameters. For the purposes of this research, AIC will lead to an overfit model. The Anderson-Darling Test is another popular criterion that is unfit for this setting. Similar to Kolmogorov-Smirnov test, the Anderson-Darling test focuses on whether two samples came from the same probability distribution. This criterion does a nice job with tails of probability distribution, but we are not interested in resolving dynamics for rare events. Moving forward, we focus our efforts on a hybridization of the Kolmogorov-Smirnov test. Such test is used to determine whether a set of samples were drawn from a given distribution. The Kolmogorov-Smirnov statistic for a set of samples, $\{x_i(t_n)\}_{i=1}^M$, at time t_n is defined

$$D = \sup_{\forall x_i} \Big\{ |F(x_i,t_n) - F_{obs}(x_i,t_n)| \Big\}, \tag{93}$$

where $F(x,t)$ represents the CDF of x obtained using our equation-driven PDF estimator and F_{obs} represents the empirical distribution of our sample set $\{x_i(t_n)\}_{i=1}^M$. For each timestep, we order the samples to obtain $\{\hat{x}_i(t_n)\}_{i=1}^M$. From the ordered dataset, we construct an empirical distribution,$F_{obs}(x,t_n) = \frac{1}{M}\sum_{i-1}^M I_{[-\infty,x]}\big(\hat{x}_i(t_n)\big)$, where $I_{[-\infty,x]}\big(\hat{x}_i(t_n)\big)$ is equal to 1 if $\hat{x}_i(t_n) \leqslant x$ and 0 otherwise. At the 95% level, the critical distance is approximately $D_{crit,0.05} = 1.36M^{-1/2}$. To circumvent direct computation of the CDF at each time step, we employ a hybrid representation of the Kolmogorov-Smirnov distance. This

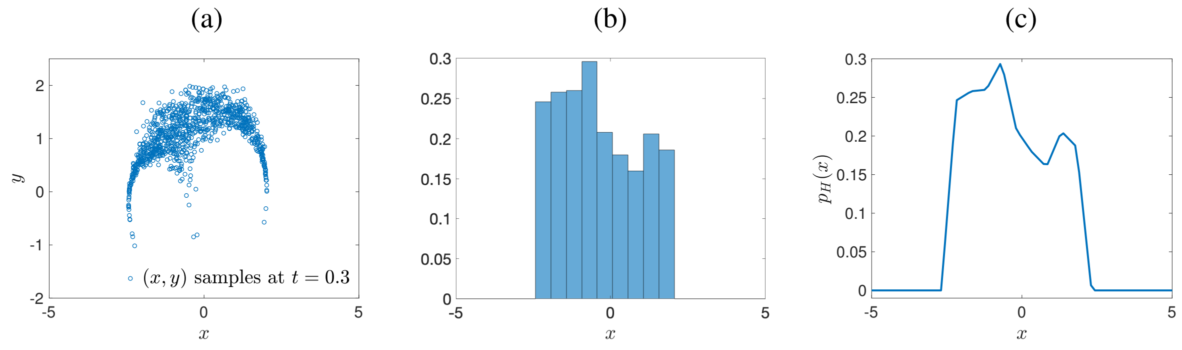


Figure 44: (a) Samples of (37) plotted at $t = 0.3$; (b) A histogram generated from the data on the left; (c) The function $p_H(x)$ that will be used in the cost functional J.

hybrid measure focuses on the divergence between two PDFs,

$$D_{hybrid} = \left| p(x, t_n) + \frac{\Delta t}{12}\Big(23L(x, t_n) - 16L(x, t_{n-1}) + 5L(x, t_{n-2})\Big) - p_H(x, t_n) \right| \tag{94}$$

where p_H is a simple approximation of the PDF from a histogram of the samples.

Instead of minimizing the supremum of D_{hybrid}, we minimize the mean integrated square error between our approximation of the PDF and $p_H(x, t)$. The hyperbolic PDE-constrained optimization problem associated with the hybrid Kolmogorov-Smirnov cost functional is

$$\min_{p,u} \quad J[p, u] = \int_{-\infty}^{\infty} [p(x, t_{n+1}) - p_H(x, t_{n+1})]^2 \, dx + \sum_{i=1}^{M} \big(u(x_i, t_{n+1}) - y_i\big)^2 + \lambda \int_{-\infty}^{\infty} \frac{\partial^2 u(x, t_{n+1})}{\partial x^2} \, dx, \tag{95}$$

$$\text{s.t.} \quad p(x_j, t_{n+1}) = p(x_j, t_n) + \frac{\Delta t}{12}\Big(23L(x_j, t_n) - 16L(x_j, t_{n-1}) + 5L(x_j, t_{n-2})\Big) \tag{96}$$

$$L(x_j, t_n) = -\sum_{k=1}^{Q} D_{jk}\Big(2x_k u(x_k, t_n) p(x_k, t_n) - p(x_k, t_n)\Big). \tag{97}$$

The cost functional $J[p, u]$ depends on a state variable, p, and a control variable, u. In the context of (89), $u(x, t)$, mimics the conditional expectation $\mathbb{E}\{y|x, t\}$. We use M to denote the number of samples used in the cost functional, N to denote the number of timesteps used in the temporal discretization, and Q to denote the number of Fourier modes used in the spatial discretization. Note that the problem at time instance t_n depends on the solution at times t_{n-1}, t_{n-2}, and t_{n-3}. This temporal dependence has the desirable effect of enforcing time-continuity in the solution of $p(x, t)$. The constrained optimization problem can be recast as an unconstrained optimization problem. By replacing $p(x_i, t_{n+1})$ with (96) and shifting indices slightly in the cost functional, we arrive at the following unconstrained problem

$$\min_{p,u} \quad J[p, u] = \int_{-\infty}^{\infty} \left[p(x, t_n) + \frac{\Delta t}{12}\Big(23L(x, t_n) - 16L(x, t_{n-1}) + 5L(x, t_{n-2})\Big) - p_H(x, t_{n+1}) \right]^2 dx$$
$$\cdots + \big(u(x_i, t_n) - y_i\big)^2 + \lambda \int \frac{\partial^2 u(x, t_n)}{\partial x^2} \, dx, \tag{98}$$

where $L(x, t_n)$ represents the right hand side of the discretized PDF equation defined in (91).

Table 31: Change of variables ensuring that the $\mathbb{E}[y|x]$ is periodic.

x	y	$p(x,t_0)$	$\mathbb{E}[y|x]$
$\hat{x}$	$\hat{y}\exp\{-c\hat{x}^2\}$	$(2\pi)^{-1/2}\exp\{-x^2/2\}$	$\rho\, x\exp\{-c\,x^2\}$
$\hat{x}$	$\hat{y}\,/\,(1+c\hat{x}^2)$	$(2\pi)^{-1/2}\exp\{-x^2/2\}$	$\rho x/(1+cx^2)$

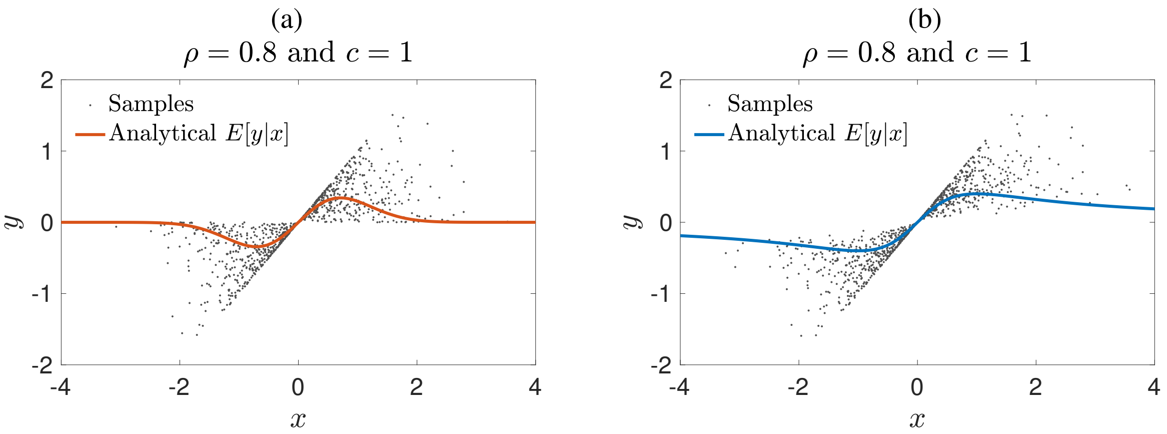


Figure 45: (a) Analytical conditional expectation, (84), plotted alongside samples associated with the exponential probability transformation; (b) Analytical conditional expectation plotted alongside samples associated with the Lorentzian probability tranformation.

2.3.4.3 Initialization We will need an approximation of $p(x,t_1)$, $p(x,t_2)$, ..., $p(x,t_k)$ to initialize the k^{th}-order Adams-Bashforth time stepping scheme. These pdf approximations are obtained using a linear multistep method (Forward Euler for $t_0+\Delta t$, and Adams-Bashforth for subsequent timesteps). Additionally, we will need an approximation of the control, $u(x,t)$ at the first k time steps. Given $p(x,y,t_0)$, we know the exact value of $u(x,t_0) = \mathbb{E}[y|x,t_0]$ for all x. However, we must be mindful of the numerical scheme we've used for spatial discretization. The control signal must be periodic with zero boundary conditions on the numerical domain of x. This requires some care in the selection of an initial condition. If we allow $p(x,y,t_0)$ to be a bivariate Gaussian, the resulting $u(x,t_0) = \mathbb{E}[y|x,t_0]$ is non-periodic (i.e. it is linear, specifically $\mathbb{E}[y|x,t_0] = \mu_y + \rho\sigma_y/\sigma_x(x-\mu_x)$ where ρ is the correlation). To find a suitable initial density, we begin by considering jointly Gaussian random variables $\hat{x}$ and $\hat{y}$ with means $\mu_x = \mu_y = 0$ and variances $\sigma_x^2 = \sigma_y^2 = 1$. To ensure $u(x,t_0) = \mathbb{E}[y|x]$ is periodic while leveraging on the convenient properties of Gaussian distributions, we consider the probability transformation from $(\hat{x},\hat{y})$ to (x,y). In Table 31 we summarize reasonable options for x and y. Both of these conditional expectations have the qualities we desire. They are continuous, differentiable, and $\mathbb{E}[y|x] \to 0$ as $x \to \pm\infty$. The parameter $c > 0$ controls how quickly the conditional expectation decays to 0. We have carefully selected the initial condition in order to leverage on the convenient sampling properties of Gaussian distributions. To generate samples for (x,y), we will start by sampling $(\hat{x},\hat{y})$ from a standard normal distribution. With $(\hat{x},\hat{y})$ available, we can use the mappings presented in the Table above to convert $(\hat{x},\hat{y})$ samples to (x,y) samples.

2.3.4.4 Numerical results Each of the results presented were generated using an initial condition based on the exponential transformation $y = \hat{y}\exp\{-cx^2\}$. The smoothing parameter, λ, is fixed at 0.25 and the value of μ in (37) is set to 5. Each simulation was run on $t \in [0,1]$. For the results presented in Figure 46, $M = 50$ samples of (37) were used and the benchmark was constructed using a density histogram with

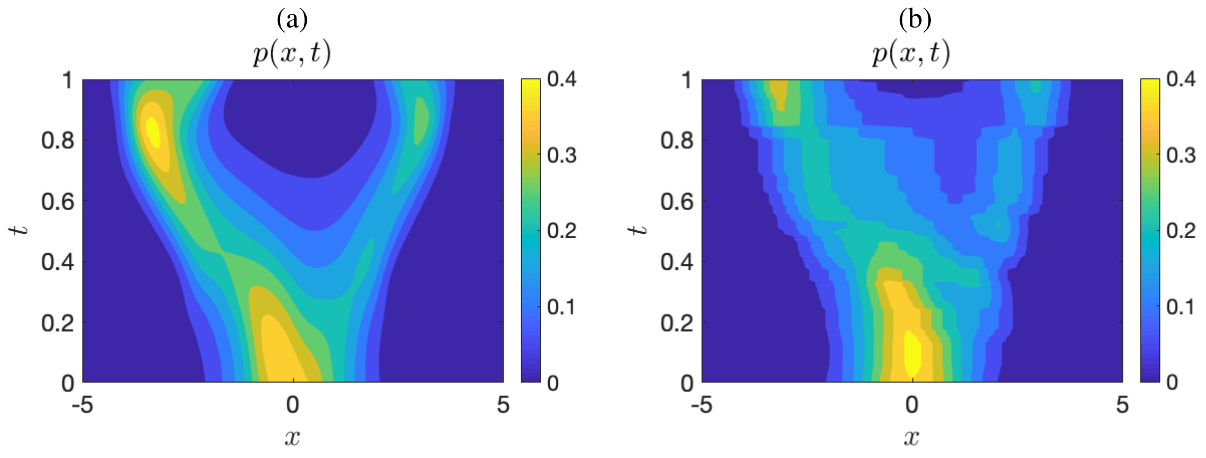


Figure 46: Kolmogorov-Smirnov Hybrid Cost Optimization Results: (a) Benchmark estimate of $p(x,t)$ generated using a traditional kernel density estimator and 5000 samples at each time step; (b) Equation-driven PDF estimate obtained by solving (98) with $M = 50$ samples.

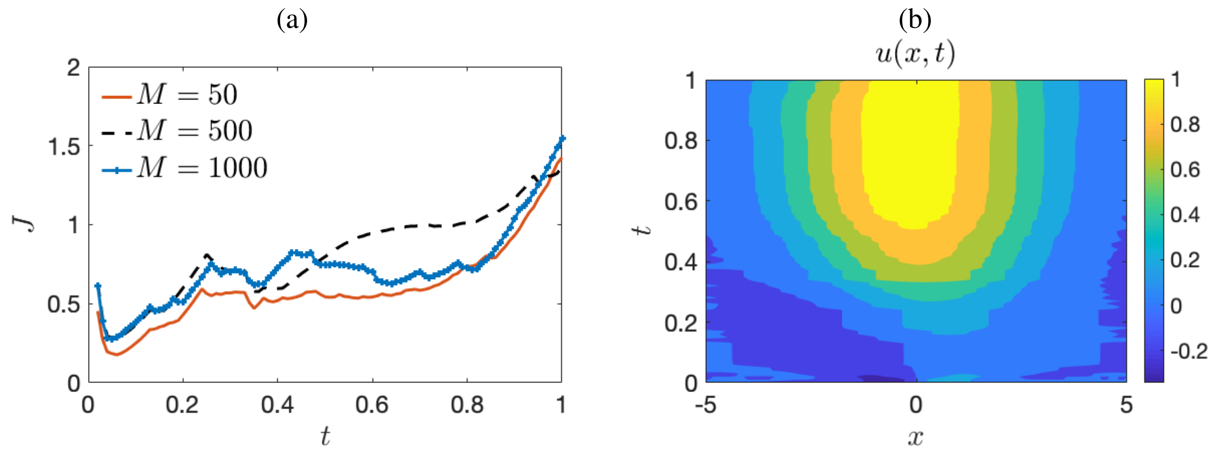


Figure 47: (a) Cost functional J versus time for $M = 50$, $M = 500$, and $M = 1000$ samples, and (b) control signal generated from (98) using 50 samples from (37).

midpoint interpolation. The control signal generated using 50 samples is plotted in Figure 47

Although the results presented in Figure 46 appear to indicate that few samples are needed for reasonable accurate equation-driven PDF estimation in an optimization framework, we must be careful about the design of our cost functional. In any case, the direct comparison between our equation-driven PDF estimator and a benchmark introduces a suite of issues. There is, of course, the question of how to construct the benchmark approximation. A benchmark that is not true to the data will cause the PDF approximation to evolve in an inaccurate maner. Moreover, if our benchmark is accurate to order S, our PDF approximation may be limited to accuracy of order S as well.

2.4 Task III-IV: Optimal control under uncertainty for high dimensional nonlinear systems

Designing efficient optimal control algorithms for stochastic dynamical systems is an extremely challenging problem. State-of-the-art probabilistic (spectral) collocation methods can handle random input vectors with dimension up to 3 or 4. Beyond such small number of random variables, the memory requirements and computational cost of such methods become unacceptably large, yielding a bottleneck that hasn't yet been overcome. In this Section, we introduce a new *potentially groundbreaking algorithm for computational optimal control under uncertainty*. The new algorithm is based on interior point methods, common sub-expression elimination, and exact gradients obtained with automatic differentiation and computational graphs [1, 4]. The new algorithm has extremely low memory requirements (see Figure 57), which means that it allows us to process a massive number of sample trajectories (in parallel) and determine the performance metric and associated optimal controls in a very efficient way. The method is based on three distinct but complementary elements:

- We implemented a multi-shooting optimal control scheme with piece-wise constant (in time) control approximation. Multi-shooting is known for its numerical stability in solving deterministic optimal control problems. We modified the deterministic algorithm and made it effective for uncertain optimal control problems. This includes imposing particle-independent continuity conditions across adjacent time segments.
- The optimization solver we developed combines an interior point method [69] with backward propagation and automatic differentiation algorithms coded in TensorFlow [1]. In this way, we can compute exact gradients of discretized cost functionals in a very efficient way, while allowing at the same time the imposition of nonlinear constraints.
- To further improve performance and memory management, we implemented a Common Subexpression Elimination (CSE) technique that can massively reduce memory consumption during computations. As we shall see in Section 2.4.4, CSE *eliminates* the growth of memory requirements with respect to the increase of the time discretization points. This feature is *essential* for solving high dimensional problems and problems with long time horizon. It also reduces the discretization error in the control approximation, since smaller time steps can be afforded.

Hereafter, we provide technical details on the proposed stochastic optimal control algorithm, and demonstrate its accuracy and effectiveness in applications to path-planning problems under uncertainty involving UGVs and UAVs.

2.4.1 Multi-shooting for optimal control problems under uncertainty

Multi-shooting optimal control algorithms have been used extensively in control of deterministic systems because of their numerical stability. To illustrate how multi-shooting works, consider a one-dimensional deterministic optimal control problem,

$$\min_{u(t)} J([x(t_f), u(t)]) \quad \text{subject to} \quad \dot{x} = f(x, t, u), \quad x(0) \quad \text{deterministic}, \tag{99}$$

where, $x \in \mathbb{R}$ (state variable), $u \in \mathbb{R}$ (control).

- The first step of the multi-shooting method is to divide time horizon $[0, t_f]$ into S sub-intervals which we will call *shooting segments*,

  $$[t_k, t_{k+1}], \qquad k = 1, \ldots, S, \tag{100}$$

  with $t_1 = 0$ and $t_{S+1} = t_f$. Each shooting segments can have different lengths. Each shooting segment is further discretized into uniform grids

  $$t_{k,j} = t_k + (j-1)h_k, \qquad j = 1, 2, \ldots, N_k, \quad k = 1, \ldots, S, \tag{101}$$

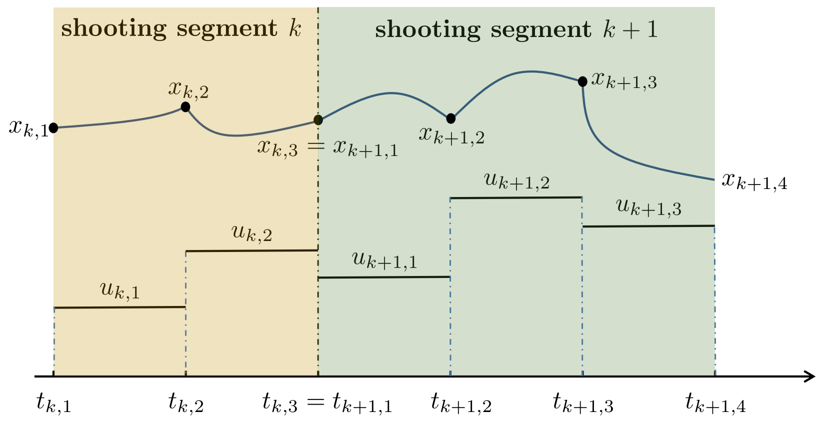


Figure 48: Sketch of the multi-shooting setting for the optimal control of one-dimensional deterministic problem.

where $h_k = \frac{t_k - t_{k-1}}{N_k - 1}$ is the time step size used in the kth shooting segment $[t_k, t_{k+1}]$. Note that different shooting segment can have different step size h_k, depending on the size of the grid N_k. The total number of time discretization points is therefore

$$N_{\text{tot}} = \sum_{k=1}^{S} N_k, \qquad \text{(total number of time-discretization points).} \tag{102}$$

- Within each shooting segment $[t_k, t_{k+1}]$, control $u(t)$ is approximated by a *piecewise constant function* over the grid $\{t_{k,j}\}_{j=1}^{N_k}$, i.e.,

  $$u(t) \;\approx\; \hat{u}_k(t) \;\triangleq\; u_{k,j}, \text{ when } t \in [t_{k,j}, t_{k,j+1}), j = 1, \ldots, N_k - 1. \tag{103}$$

  The approximation/discretization of the control variable $u(t)$ is sketched in Figure 48. The constant variables $u_{k,j}$ define part of decision vector to be determined through optimization.

- To ameliorate the known sensitivity issues of single direct shooting methods, in multi-shooting the starting point of each state trajectory withing each shooting segment $[t_k, t_{k+1}]$ (denoted as $x_{k,1}$), is also part of the optimization variables. With the piece-wise constant control approximation available ($\hat{u}_k(t)$), the state vector at any time instant can be computed by numerical integration

  $$\hat{x}(t_{k+1}) = x_{k,1} + \int_{t_k}^{t_{k+1}} f(\hat{x}(t), t, \hat{u}_k(t))dt. \tag{104}$$

  To ensure the continuity of the state across the shooting segments, we impose constraints

  $$x_{k+1,1} \;=\; \hat{x}(t_{k+1}). \tag{105}$$

To extend the basic idea of multi-shooting from one-dimensional deterministic optimal control problems to high dimensional optimal control problems under uncertainty, we introduce the following notation:

$$\boldsymbol{x}_{j,k}^{(i)} = \boldsymbol{x}^{(j)}(t_{j,k}, \omega_i) \qquad \boldsymbol{u}_{j,k} = \boldsymbol{u}(t_{j,k}) \tag{106}$$

for the discretized trajectory of the state vector and control where $i = 1, \dots, N$ labels a specific realization of state process, $j = 1, \dots, S$, labels the shooting segment $[t_j, t_{j+1}]$, and $k = 1, \dots, N_j$ labels the time instant within the jth shooting segment. We also denote the ith-sample of the full discretized state process in $[0, t_f]$ as

$$\boldsymbol{X}^{(i)} = \left\{\boldsymbol{x}^{(i)}_{1,1}, \dots \boldsymbol{x}^{(i)}_{1,N_1}, \boldsymbol{x}^{(i)}_{2,1}, \dots, \boldsymbol{x}^{(i)}_{2,N_2}, \dots\right\}, \qquad i = 1, ..., N. \tag{107}$$

Similarly, the discretized control vector is

$$\boldsymbol{U} = \{\boldsymbol{u}_{1,1}, \dots \boldsymbol{u}_{1,N_1}, \boldsymbol{u}_{2,1}, \dots, \boldsymbol{u}_{2,N_2}, \dots\}. \tag{108}$$

It is also convenient to define an additional vector collecting the sample values the process $\boldsymbol{x}(t, \omega)$ at the boundaries of the shooting segments, i.e.,

$$\boldsymbol{X}^{(i)}_b = \left\{\boldsymbol{x}^{(i)}_{1,1}, \boldsymbol{x}^{(i)}_{1,N_1}, \boldsymbol{x}^{(i)}_{2,1}, \boldsymbol{x}^{(i)}_{2,N_2} \dots, \boldsymbol{x}^{(i)}_{S,N_S}\right\}, \tag{109}$$

and the *full vector* of states and boundary values

$$\boldsymbol{X} = \left\{\boldsymbol{X}^{(1)}, \dots, \boldsymbol{X}^{(N)}\right\}, \qquad \boldsymbol{X}_b = \left\{\boldsymbol{X}^{(1)}_b, \dots, \boldsymbol{X}^{(N)}_b\right\}. \tag{110}$$

The continuity of each sample trajectory across different shooting segments can be imposed in the same manner as Eq. (105). However, such implementation would introduce a very large number of optimization constraints (one for each sample path), making the optimization problem very challenging to solve, and potentially increasing unfeasible regions. Instead, we introduce the following single constraint

$$\sum_{k=1}^{S} \sum_{l=1}^{N} \left\| \boldsymbol{x}^{(l)}_{k,N_k} - \boldsymbol{x}^{(l)}_{k+1,1} \right\|^2 = 0, \tag{111}$$

which guarantees continuity of sample trajectories across different shooting segments, while being completely agnostic on the trajectory labels. In other words, across different shooting segments the constraint (111) might yield a reshuffling of the trajectories labels, but as we know this does not affect statistical properties. Moreover, we have the set of constraints

$$\boldsymbol{x}^{(i)}_{k,N_k} = \boldsymbol{x}^{(i)}_{k,1} + \sum_{j=1}^{N_k} w_j \boldsymbol{f}\left(\boldsymbol{x}^{(i)}_{k,j}, t_{k,j}, \boldsymbol{u}_{k,j}\right), \qquad i = 1, ..., N, \tag{112}$$

which represent the discretized version of equation (104) in the vector setting (here w_j are temporal integration weights). Using this notation we formulate the following multi-shooting ensemble optimal control problem

$$\min_{\boldsymbol{U}, \boldsymbol{X}_b} J(\boldsymbol{X}, \boldsymbol{U}) \quad \text{subject to (111) and (112).} \tag{113}$$

2.4.2 Efficient gradient computation for constrained optimization

The large-scale *constrained optimization problem* (113) can be solved by using optimization algorithms based on gradient information. To this end, it is very important to be able to compute the gradient of the cost function J with respect to the decision variables $(\boldsymbol{U}, \boldsymbol{X}_b)$, with accuracy and efficiency. In this Section we provide technical details on how we implemented such gradient computation using automatic differentiation functions available in TensorFlow [1] and backward propagation. Our algorithm provides exact gradients at a negligible computational cost since all operations are performed on very efficient computational graphs.

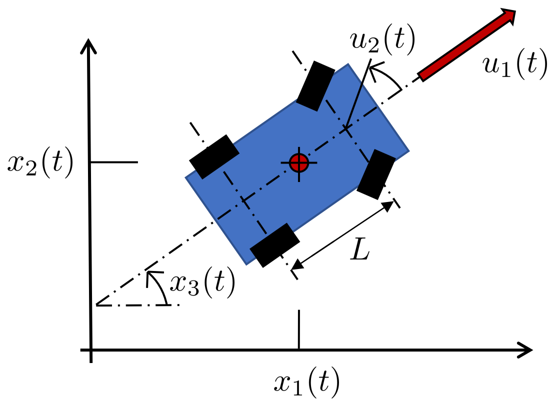


Figure 49: Model of an Unmanned Ground Vehicle (UGV).

2.4.2.1 Dataflow graphs and gradient evaluation Modern machine learning frameworks use decentralized data graphs to map computations to different nodes in a computational cluster [1]. The graphs use data as edges and computational operations as nodes. The dataflow graph is pre-compiled, optimized and stored as metadata in memory for the duration of the calculation. Using reverse automatic differentiation, the computational graph of the gradients with respect to a given cost function is simultaneously formed and similarly stored. Using these two graphs simultaneously allows the efficient computation of the gradients with respect to the decision variables. The calculations are performed using a data event driven mechanism allowing for nearly simultaneous calculations of gradients as the dynamical system is being integrated. To illustrate the main idea, consider the Euler forward scheme applied to the following dynamics representing a a real wheel driving Unmanned Ground Vehicle (UGV) (see Section 2.4.3)

$$\begin{cases} \dot{x}_1 = u_1 \cos(x_3) \\ \dot{x}_2 = u_1 \sin(x_3) \\ \dot{x}_3 = \dfrac{u_1}{L}\tan(u_2) \end{cases} \tag{114}$$

where $(x_1(t), x_2(t))$ is the location of the vehicle, $x_3(t)$ is the orientation angle, u_1 is the forward velocity, and $u_2(t)$ is the steering angle. Such scheme can be written as

$$\begin{cases} x_1(t_{k+1}) = x_1(t_k) + hu_1(t_k)\cos(x_3(t_k)) \\ x_2(t_{k+1}) = x_2(t_k) + hu_1(t_k)\sin(x_3(t_k)) \\ x_3(t_{k+1}) = x_3(t_k) + h\dfrac{u_1(t_k)}{L}\tan(u_2(t_k)) \end{cases} \tag{115}$$

The generation of a data graph in this case is particularly simple. Consider, as an example the cost functional

$$J([\boldsymbol{x}(t)]) = x_1(t_N)^2. \tag{116}$$

The gradient of J with respect to $\{u_1(t_1), \ldots, u_1(t_N)]\}$ and $\{u_2(t_1), \ldots, u_2(t_N)]\}$ can be easily computed with backward propagation (chain rule). For instance,

$$\frac{\partial J([\boldsymbol{x}(t)])}{\partial u_1(t_k)} = \frac{\partial x_1(t_N)}{\partial x_{j_1}(t_{N-1})}\frac{\partial x_{j_1}(t_{N-1})}{\partial x_{j_2}(t_{N-2})}\cdots\frac{\partial x_{j_{k+2}}(t_{k+2})}{\partial x_{j_{k+1}}(t_{k+1})}\frac{\partial x_{j_{k+1}}(t_{k+1})}{\partial u_1(t_k)} \tag{117}$$

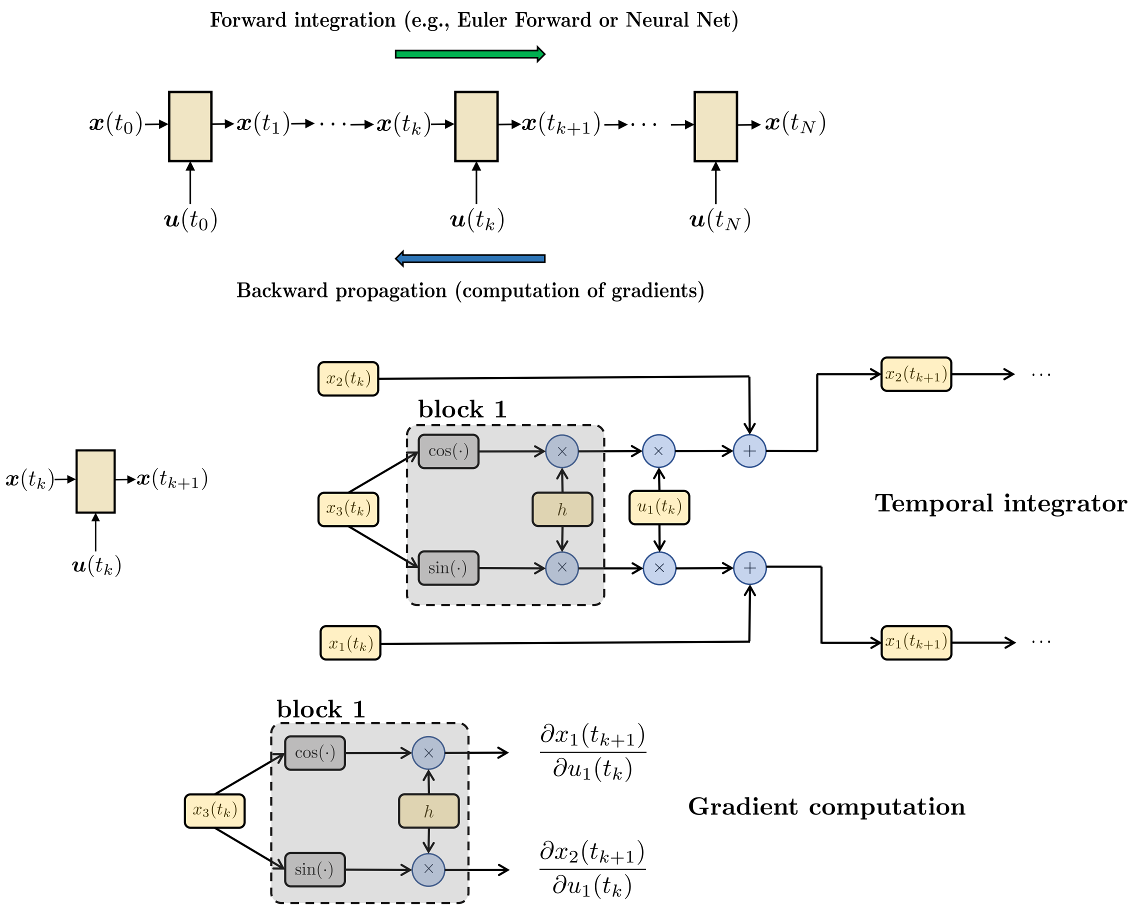


Figure 50: Portion of the data graph for the UGV model discretized with Euler forward time integration and graph the compute the kth component of the gradient of the cost functional.

As clearly seen from the computational graph sketched in Figure 50, portions of the calculations for the forward trajectory propagation as well as the backward gradient calculation are shared. Using *reactive programming*, shared portions of the graphs are computed only once, while *memoization* realizes additional substantial computational savings.

The exact savings depend on the efficiency of the graph compiler as well as the particular dynamics. Once the compiler creates and compiles the data graph in memory for a single trajectory, adding a dimension for sampling does not involve any changes to the graph or gradient calculations. By choosing a non-adaptive integration scheme, we ensure that the operations can be performed in lock step mode across all samples, and can utilize hardware with wide SIMD capabilities such as GPU, TPU or ASIC cards.

2.4.2.2 Integration of efficient gradient computation with constrained optimization solvers In the previous Section we have seen that standard high-performance machine learning packages such as TensorFlow [1], can be effectively utilized to compute gradients of cost functionals very efficiently. However, such packages lacks the ability to handle constraints. In the simplest implementation, constraints can be simply added as penalty terms in the cost functional, e.g., as we did in physics-informed neural net traning (see Section 2.2.1 and equation 35). Such simplified is not allowed in optimal control problems, since the

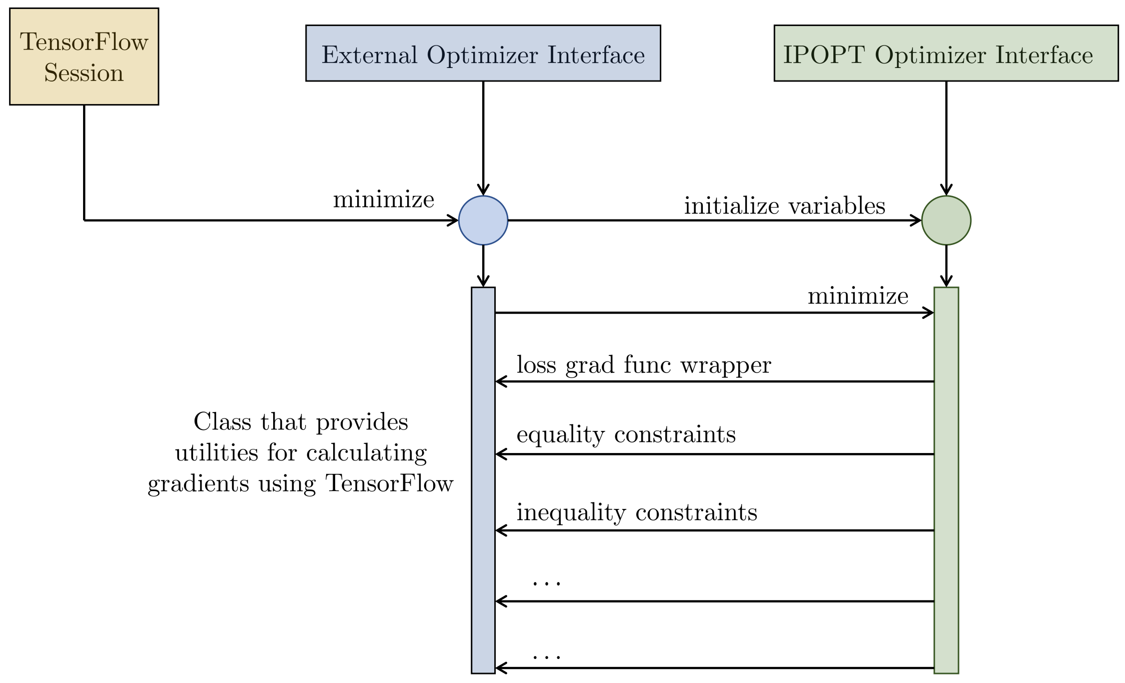

TensorFlow Session
External Optimizer Interface
IPOPT Optimizer Interface
minimize
initialize variables
minimize
loss grad func wrapper
Class that provides utilities for calculating gradients using TensorFlow
equality constraints
inequality constraints
...
...

Figure 51: UML (Unified Modeling Language) diagram for the integration of IPOPT with TensorFlow illustrating the major interface points.

constraints, which represent dynamics and physical limitations of the control systems, can only be satisfied approximately in this setting. Also, the resulting cost function may be highly nonlinear and ill-conditioned. On the other hand, sequential quadratic programming (SQP) and interior point methods have had great success in computational optimal control. Optimizers such as SNOPT [22] and IPOPT [6] are widely used in solving deterministic optimal control problems. The success of these solvers in computational optimal control is largely due to the excellent performance of numerical optimization algorithms in handling constraints. Such constraint optimization algorithms are currently not available in standard machine learning software packages. On the other hand, optimization software generally lacks effective routines to efficiently compute accurate gradients of the cost functional. Thus, we developed new software to integrate IPOPT [69], an optimizer that relies on interior point methods , with TensorFlow [1]. Such integration combines fast gradient computation in TensorFlow with the efficient constrain optimization algorithms implemented in IPOPT, making it suitable for solving high-dimensional optimal control problems under uncertainty. The integration can be done in two ways. The first one replaces the built-in optimizers in TensorFlow with IPOPT. This approach equips TensorFlow with the ability to handle constraints explicitly, which is crucial, e.g, when training physics-informed data-driven neural nets for PDFs with Liouville equations as a constraint. However, this requires labor intensive software development. The second way incorporates TensorFlow into IPOPT, in the sense that the IPOPT uses TensorFlow functions to compute the gradient of the objective functional. A third party contributed python wrapper around IPOPT [69] allows for a relatively straightforward integration.

2.4.3 Application to a UGV stochastic path planning problem

Consider the systems of equations (114), describing the dynamics of a simple unmanned ground vehicle (UGV) . The control objective is to drive the vehicle from a uniformly distributed random initial condition

$$\begin{bmatrix} x_{10} \\ x_{20} \\ x_{30} \end{bmatrix} \sim \begin{bmatrix} U(0.95, 1.05) \\ U(0.95, 1.05) \\ U(-0.05, 0.05) \end{bmatrix} \tag{118}$$

to a target located precisely at $(x_1, x_2) = (0, 0)$. The parameter L, representing the distance between the front and rear axles, is also random and following a normal distribution with mean 0.1 and variance 10^{-2}. The controls are the forward velocity, $u_1(t)$, and the steering angle, $u_2(t)$, subject to the following constraints

$$-1 \leqslant u_1(t) \leqslant 1, \tag{119}$$

$$-\frac{\pi}{4} \leqslant u_2(t) \leqslant \frac{\pi}{4}. \tag{120}$$

To represent the control objective, we define the following cost function

$$J(\boldsymbol{x}(t_f)) = \mathbb{E}\{x_1(t_f)^2 + x_2(t_f)^2\}, \tag{121}$$

where $t_f = 50$ is the final transfer time, and $\mathbb{E}\{\cdot\}$ is an expectation over all possible realizations of the initial UGV position and distance between the front and rear axles. The expectation in (121) can be approximated by computing the ensemble mean over $N = 1000$ sample trajectories. Also, the integration period $t_f = 50$ is discretized in terms of $S = 2$ shooting segments, $t \in [0, 10]$ and $t \in [10, 50]$, with $N_1 = 100$ and $N_2 = 400$ time instants respectively. Within each segment, a 4th order Runge-Kutta method with a time-step size $h = 0.1$ is used to numerically propagate the entire ensemble of realization (which can be easily vectorized and distributed across different cores).

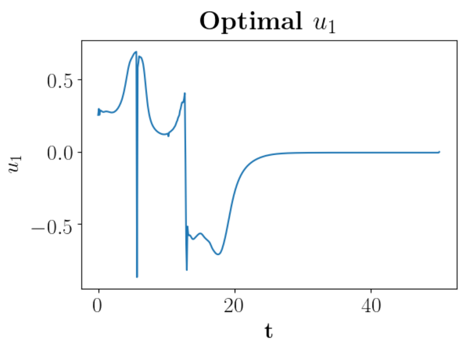
We solved the discretized constrained optimization problem mentioned above by using IPOPT [69], an open source software package for large-scale nonlinear optimization. As we mentioned before, we linked IPOPT with automatic differentiation functions coded in TensorFlow. This allowed us to compute the gradient of the objective functional *exacly*. The optimal controls we obtain for the forward velocity, $u_1(t)$ and the steering angle $u_2(t)$ are shown in Figure 52. The corresponding trajectories of the UGV under the optimal controls are shown in Figure 53. Note that the final position of the UGV is clustered around the target location of $(0, 0)$ (see Figure 53b). As a comparison, we generate a different control by minimizing the quadratic cost

$$J(u_1, u_2) = \int_0^{t_f} (u_1^2(t) + u_2^2(t))dt \tag{122}$$

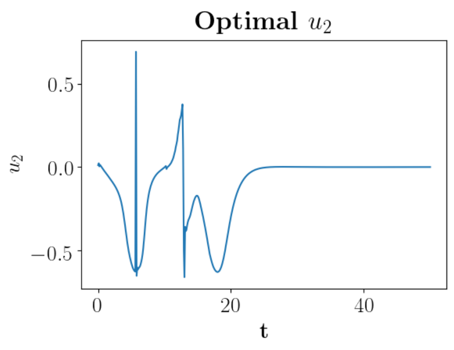
for the most probable scenario, i.e., setting the initial condition and parameter equal to their mean values

$$\begin{cases} x_1(0) = 1.0 \\ x_2(0) = 1.0 \\ x_3(0) = 0.0 \\ L = 1.0 \end{cases} \tag{123}$$

together with the final condition $(x_1(t_f), x_2(t_f)) = (0, 0)$. This approach is typically used in engineering control applications. The resulting control is defined as *nominal control*. To show that nominal controls are ineffective in the presence of uncertainty, in Figure 54 we plot the stochastic dynamics of the UGV under nominal control. By comparing Figure 53 with Figure 54, it is clear that trajectories under optimal control

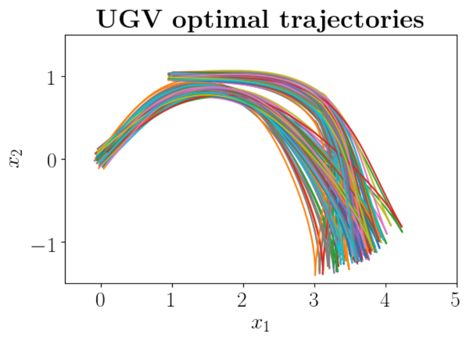


(a) Optimal forward velocity $u_1(t)$

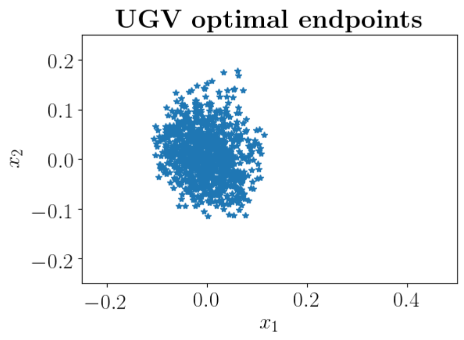


(b) Optimal steering angle $u_2(t)$

Figure 52: Optimal controls for the UGV stochastic path planning problem.

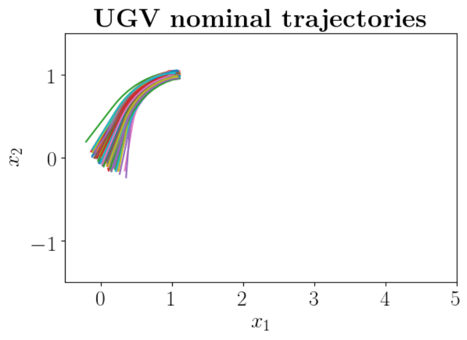


(a) Trajectories driven by optimal control

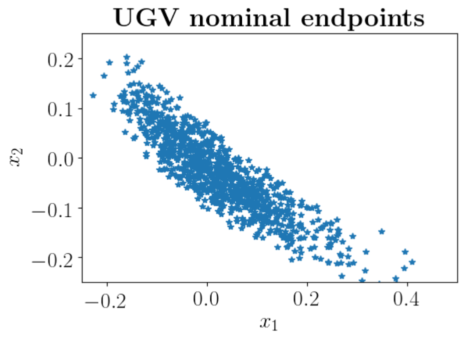


(b) Endpoints of the UGV under optimal control

Figure 53: Trajectories and endpoints with optimal control for UGV example over 1000 randomly selected samples.



(a) Trajectories driven by nominal control



(b) Endpoints of the UGV under nominal control

Figure 54: Stochastic dynamics of the UGV under nominal controls.

are more concentrated around the target location than trajectories under nominal control. It is also interesting to note that, in the case of nominal control, the trajectories keep spreading out as time advances. On the other hand, the optimal control drives the vehicle in a more complicated forward and backward motions to

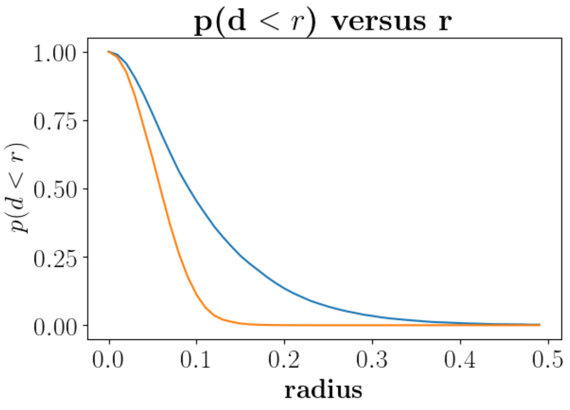
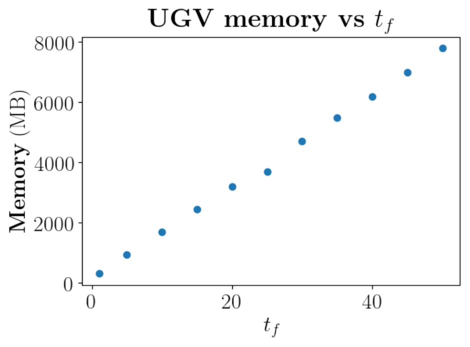
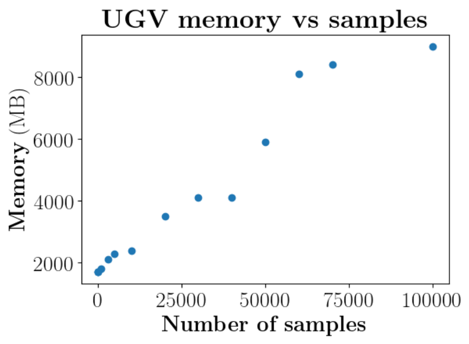


Figure 55: Probability that the UGV is found at a distance $r \geqslant d$: optimal control (orange) versus nominal control (blue).



(a) Fixed sample size ($N = 100$) and variable final time (t_f).



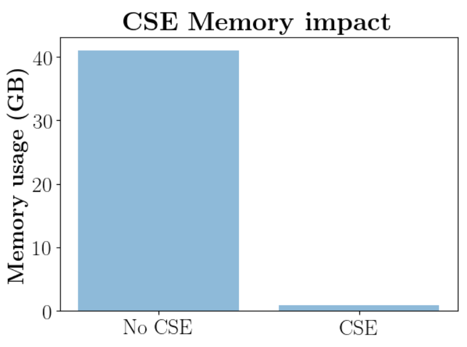
(b) Fixed final time ($t_f = 10.0$) and variable size (N).

Figure 56: Memory storage requirement for the stochastic UGV path planning problem as a function of the final time t_f and number of samples. These graphs are obtained on a Xeon v2 processor 2.3GHz with 10 cores.

mitigate the effect of uncertainty in the trajectories. In Figure 55 we plot the probability that the UGV is found at a distance $r \geqslant d$ from the target location at time $t_f = 50$. Calculations are based on 10000 Monte Carlo samples. It is seen that 90% of the trajectories endpoints are within a radius of 0.15 if we the optimal controls, while they are within a larger radius of 0.3 if we use nominal controls.

2.4.4 Common Subexpression Elimination (CSE)

We have seen in the previous Section that by combining multi-shooting, automatic differentiation, and non-linear optimization, we can develop an effective algorithm that can handle uncertainty in the UGV path planning problem. However, as shown in Figure 56, the memory storage required by such algorithm grows with both the final time t_f and the number of samples. Note that increasing the number of samples yields a memory plateau at different sample sizes. This happens because the SIMD (Single Instruction, Multiple Data) bandwith of the processing unit fills up. Utilizing processors with wide SIMD capabilities, will extent the plateau. To further extend the applicability of the algorithm we propose, especially to high dimensional



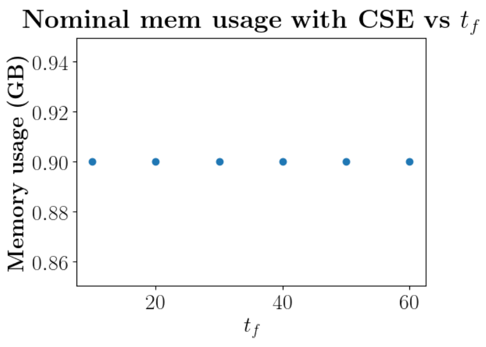


Figure 57: Memory savings obtained by using the Common Subexpression Elimination (CSE) technique in the nominal case of a thirteen-dimensional UAV example. The memory usage remains independent of the final transfer time t_f. Control results are identical with or without CSE.

stochastic problems, it is highly desirable to improve the efficiency and memory management. Noting that the integration steps constitute identical computation graphs that repeat in a daisy chained fashion, a Common Subexpression Elimination (CSE) technique can be used to *optimize memory utilization for the entire calculation*. In fact only a single step needs to be kept in memory, and the gradient calculations can be simultaneously carried forward and backwards using the chain rule.

As an example, in Figure 57 we show the effects of the Common Subexpression Elimination (CSE) in the memory storage required by the UAV stochastic path planning problem discussed in Section 2.4.5. It is seen that CSE reduces the memory storage by a factor 44.4. More importantly, the memory storage required by the algorithm *is constant with* t_f, i.e., it does not depend on the number of time instants between in $[0, t_f]$. This feature significantly broadens the applications to include higher dimensional systems, as well as control problems with long time horizon. It can also be used to improve the accuracy, since smaller step size can be afforded in piece-wise constant control approximation. Note that after application of CSE, memory utilization still depends on the number of samples. The effect can be mitigated by utilizing hardware with wide SIMD capabilities, or multiple cores.

2.4.5 Application to a stochastic path planning problem involving a fixed-wing UAV

Consider a stochastic path planning problem for a fixed-wing UAV with constant thrust. The control system is described in equation (48), where u_T is set to be zero (constant thrust). The state vector and parameters are the same as those defined in Section 2.2.8. We are interested in computing the controls $u_\alpha(t)$ and $u_\mu(t)$, i.e., the controls for the pitch and the bank angles, that can steer the UAV from an uncertain initial state (position, velocity, and other random parameters) to the final position at $(x(t_f), y(t_f), z(t_f)) = (1000, 1000, 600)$, where $t_f = 60$. The pitch and the bank angles controls are subject to the constraints

$$\begin{aligned} u_\alpha &\in [-0.05, 0.05], \\ u_\mu &\in [-0.05, 0.05]. \end{aligned} \tag{124}$$

Recently, Shaffer *et. al* considered a similar problem in the attempt to define a robust control which could mitigate the effects of aerodynamic uncertainty in the UAV planned trajectory; such uncertainty was modeled in terms of only *one random variable*, i.e., C_{x0} in Eqs. (50) and (51). The main reason for studying such simplified model was that the computational control algorithm could not handle higher-dimensional random input vectors. Indeed, the memory requirements and computational cost of such algorithm are far beyond the capabilities of a modern workstation. Next, we demonstrate that the computational optimal control algorithm we outlined in Section 2.4 can indeed handle high-dimensional random input vectors. To this end, we consider the following set of random initial conditions

$$\text{(UAV random parameters)} \qquad \begin{cases} x(0) \sim N(0.0, 0.5^2) \\ y(0) \sim N(0.0, 0.5^2) \\ z(0) \sim N(600.0, 0.5^2) \\ v(0) \sim N(27.5, 0.1^2) \\ C_{x0} \sim N(-0.0355, 0.0012^2) \\ C_{xa} \sim N(0.00292, 0.0001^2) \\ C_{z0} \sim N(-0.055, 0.001^2) \\ C_{za} \sim N(-5.578, 0.01^2) \end{cases} \tag{125}$$

Note that this includes random initial position and velocity. Note also that all four parameters defining lift and drag coefficients (50)-(51) are set to be random. To complete the specification of the problem we consider the following deterministic the initial condition for elevation, heading, pitch and bank angles

$$\begin{cases} \gamma(0) = 0 \\ \sigma(0) = \dfrac{\pi}{4} \\ \alpha(0) = -0.0088 \\ \mu(0) = 0 \end{cases} \tag{126}$$

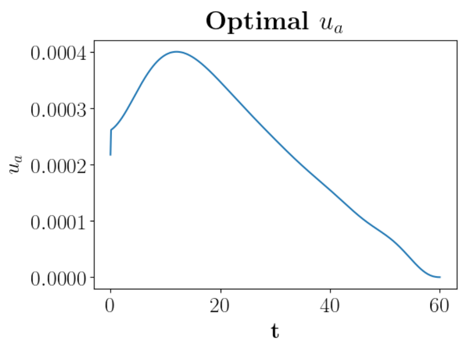
The thrust T is set to be 16.1(Newton). These parameters are the same as those used in [60]. By augmenting the parameters into the state space, the dimension of the system is 12, with 8 random variables. The objective is to design controls $u_\alpha(t)$ and $u_\mu(t))$ ($\alpha(t)$ and $\mu(t)$ are pitch and bank angles, respectively) to steer the UAV from an uncertain initial position $(x(0), y(0), z(0)$, to a deterministic final position $(x(t_f), y(t_f), z(t_f)) = (1000, 1000, 600)$ ($t_f = 60$) under uncertain dynamics modeled by 8 random variables. To represent the control objective, we construct the following cost function

$$J(\boldsymbol{x}(t_f)) = \mathbb{E}\{(x(t_f) - 1000)^2 + (y(t_f) - 1000)^2 + (z(t_f) - 600)^2\}, \tag{127}$$

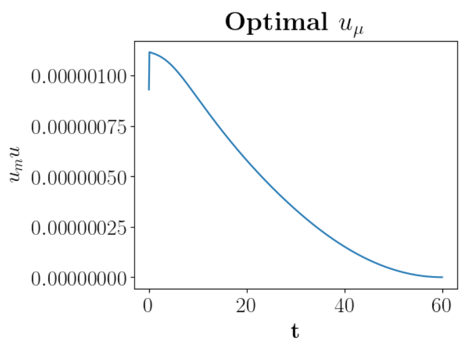
which we approximate with an ensemble mean over $N = 10000$ randomly drawn sample paths. In the simulation, a single shooting segment is used, with 4th order RK time integrator and time step size $h = 0.1$.

We solved the discretized constrained optimization problem by using IPOPT [69] linked to TensorFlow (see Section 2.4). In this way we were able to compute exact gradients of the discretized objective function. The resulting optimal controls are shown in Figure 58. To verify the the performance of the optimal controls under random initial conditions and uncertain parameters, we propagated in time the controlled dynamics of 1000 randomly generated trajectories, with initial conditions and parameters sampled according to (125). Such trajectories are shown in Figure 59. It is seen that the final positions of the UAV converge to a small region about the target location $(x(t_f), y(t_f), z(t_f)) = (1000, 1000, 600)$.

Next, let us set all random parameters in (52) equal to their mean values and compute the so-called *nominal controls*, i.e., the two functions $u_\alpha(t)$ and $u_\mu(t))$ that steer the UAV to the target location at $t_f = 60$. We would like to show that such nominal controls are not effective when we switch randomness back on. To this end, we samples 10000 realizations of (52) and propagates forward in time the dynamics of the UAV model under nominal controls. The trajectories we obtained are shown in Figure 60. By comparing Fig.59 with Fig.60, it is clear that trajectories under *optimal controls* land closer to the target location than trajectories under *nominal controls*. To quantify this improvement we define a metric that measures the

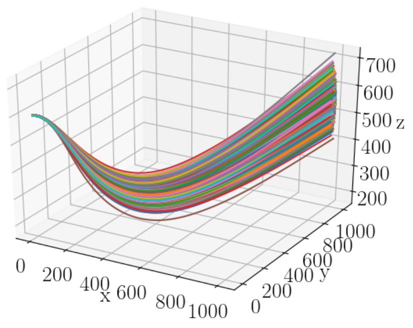


(a) Optimal control on angle of attack.

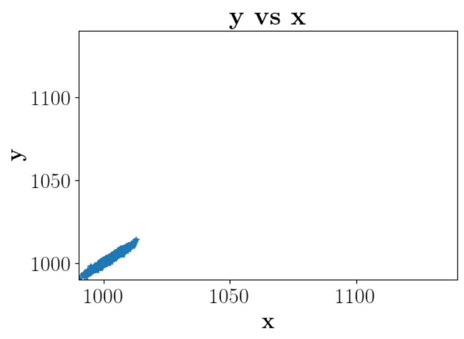


(b) Optimal control on bank angle.

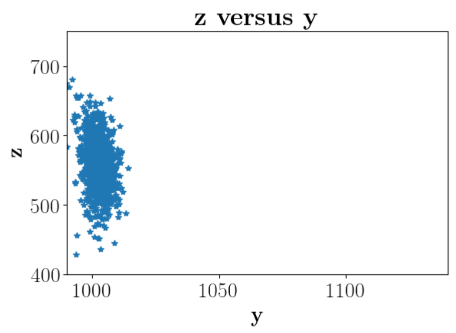
Figure 58: Stochastic path planning problem for a fixed-wing UAV with constant thrust. Shown are the optimal controls we obtain to steer the UAV from an uncertain initial position $(x(0), y(0), z(0)$, to a deterministic final position $(x(t_f), y(t_f), z(t_f)) = (1000, 1000, 600)$ $(t_f = 60)$ under uncertain dynamics modeled by 8 random variables.



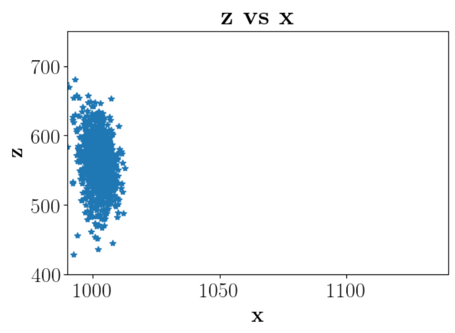
(a) Trajectories driven by optimal control



(b) x-y endpoints of the UAV under optimal control



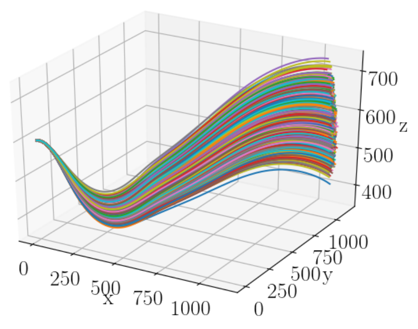
(c) y-z endpoints of the UAV under optimal control



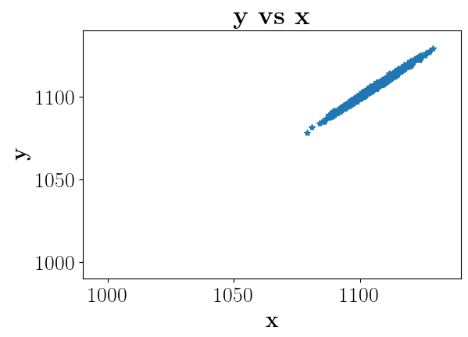
(d) x-z endpoints of the UAV under optimal control

Figure 59: Verification of the optimal controls for the stochastic path-planning problem (UAV with constant thrust).

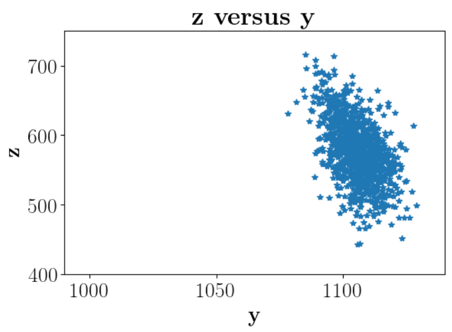
probability that the UAV is found at a distance $\geqslant d$ from the target location at final time. This probability is easy to compute once the optimal control is available. In fact, it is sufficient to plot a 1D histogram of cumulative frequencies. This is done in Figure 61, where we see that 90% of the trajectories now land within



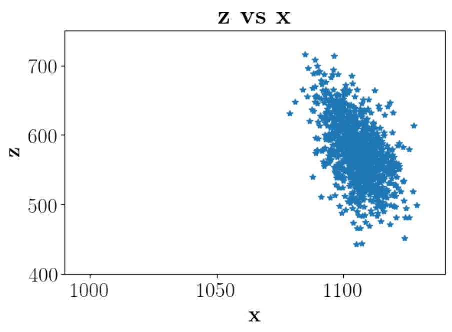
(a) Trajectories driven by nominal control



(b) x-y endpoints of the UAV under nominal control



(c) y-z endpoints of the UAV under nominal control



(d) x-z endpoints of the UAV under nominal control

Figure 60: Stochastic dynamics of the UAV under nominal controls.

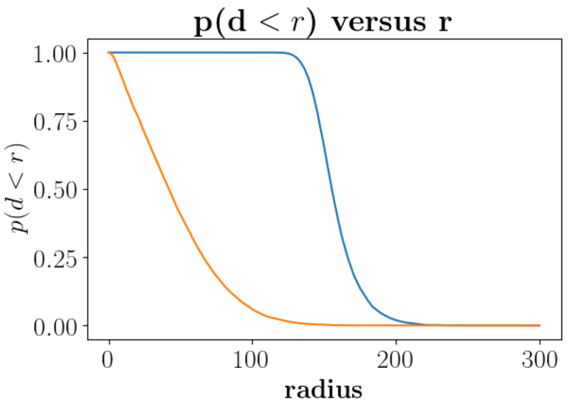


Figure 61: Probability of the UAV being located at a distance $\geqslant r$ from the target location based on 10000 Monte Carlo samples.

a radius of 150 of the objective, versus a radius of over 900 for the nominal case.

2.5 High-dimensional optimal control under uncertainty with state-space constraints

In this Section we consider a more challenging problem for the previously introduced (UAV) model (48) under uncertain initial state (position, heading angle, angle of attack, etc) and aerodynamic forces. As pointed out in [59], the system of equations (48) is valid only withing the region of the state space identified by the following equations

$$\text{(State-space constraints)} \quad \begin{cases} 13 \leqslant v(t) \leqslant 42 \\ -\dfrac{\pi}{6} \leqslant \gamma(t) \leqslant \dfrac{\pi}{6} \\ -\pi \leqslant \sigma(t) \leqslant \pi \\ 3.0 \leqslant T(t) \leqslant 35.0 \\ -\dfrac{\pi}{12} \leqslant \alpha(t) \leqslant \dfrac{\pi}{12} \end{cases} \tag{128}$$

These constraints apply over the entire interval $0 \leqslant t \leqslant t_f$. Outside this box of constraints, the model may not represent the true physical system. These constraints motivate the introduction of state-space constraints in the formulation of the optimal control problem. Without such constraints the model is invalid, and can produce unstable or un-physical behavior.

As before, the goal is to design controls $u_T(t)$, $u_\alpha(t)$ and $u_\mu(t)$) ($T(t)$, $\alpha(t)$ and $\mu(t)$ are thrust, pitch and bank angles, respectively) to steer the UAV from an uncertain initial state (position, velocity, and other random parameters) to the final position at $(x(t_f), y(t_f), z(t_f)) = (1000, 1000, 600)$, where $t_f = 60$. The thrust, pitch and the bank angles control signals, in this case, are subject to the control constraints

$$\begin{aligned} &u_T \in [-1.0, 1.0], \\ &u_\alpha \in [-0.05, 0.05], \\ &u_\mu \in [-0.05, 0.05]. \end{aligned} \tag{129}$$

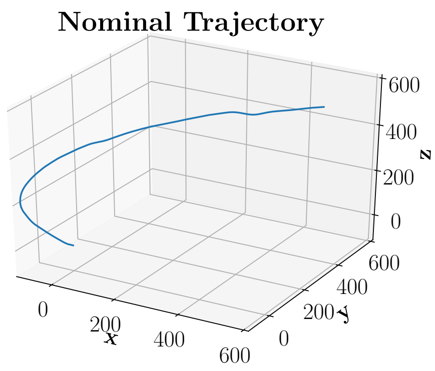
In other words, the optimal control problem we are aiming at solving has both state-space constraints (Eq. (128)) and control constraints (Eq. (129)). We consider the following random initial condition

$$\begin{cases} r \sim \mathcal{U}(0, 0.5) \qquad \theta \sim \mathcal{U}(0, \pi) \qquad \phi \sim \mathcal{U}(0, 2\pi) \\ x_0 = r\sin(\theta)\cos(\phi) \qquad y_0 = r\sin(\theta)\sin(\phi) \qquad z_0 = 600 + r\cos(\phi) \\ v_0 \sim \mathcal{U}(25.5750, 29.4250) \qquad \gamma_0 \sim \mathcal{U}(-0.05, 0.05) \qquad \sigma_0 \sim \mathcal{U}(3.1, 3.2) \\ C_{x0} \sim \mathcal{U}(-0.0380, -0.0330) \qquad C_{xa} \sim \mathcal{U}(0.0027, 0.0031) \\ C_{z0} \sim \mathcal{U}(-0.0589, -0.0548) \qquad C_{za} \sim \mathcal{U}(-5.9685, -5.1875) \end{cases} \tag{130}$$

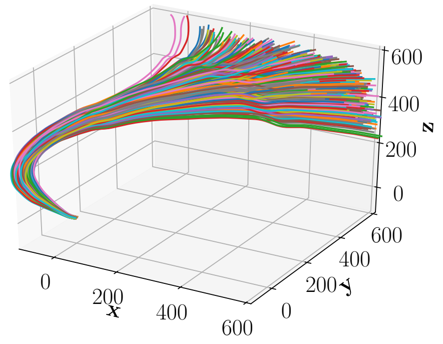
and the "nominal" (deterministic) initial state

$$\begin{cases} x_0 = 0.0 & y_0 = 0.0 \\ z_0 = 0.0 & v_0 = 27.5, \\ \gamma_0 = 0 & \sigma_0 = \pi \\ T_0 = 16.1 & \alpha_0 = -0.0088 \\ \mu_0 = 0 & C_{x0} = -0.03554 \\ C_{xa} = 0.00292 & C_{z0} = -0.055 \\ C_{za} = -5.578 \end{cases} \tag{131}$$

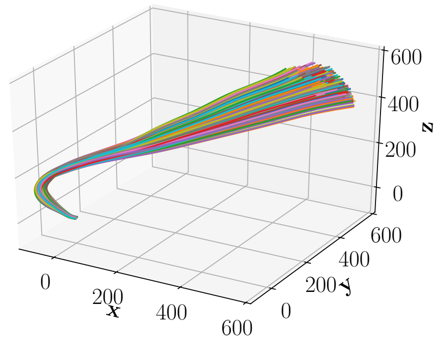
We would like to show that nominal controls are not effective when randomness is involved, even in the path-constrained setting we consider here. In particular, instead of the point-wise constraint, we implemented an



(a) Nominal trajectory.



(b) Nominal trajectories under uncertainty.



(c) Optimal trajectories under uncertainty.

Figure 62: Effects of uncertainty in the initial conditions on the nominal control and the mitigating effect of the application of an uncertainty optimal control under *ensemble state-space constraints*.

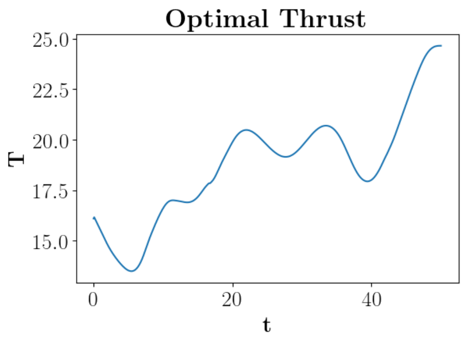
ensemble path constraint of the form

$$\text{(Ensemble path constraints)} \quad \begin{cases} 13 \leqslant \mathbb{E}\{v(t)\} \leqslant 42 \\ -\dfrac{\pi}{6} \leqslant \mathbb{E}\{\gamma(t)\} \leqslant \dfrac{\pi}{6} \\ -\pi \leqslant \mathbb{E}\{\sigma(t)\} \leqslant \pi \\ 3.0 \leqslant \mathbb{E}\{T(t)\} \leqslant 35.0 \\ -\dfrac{\pi}{12} \leqslant \mathbb{E}\{\alpha(t)\} \leqslant \dfrac{\pi}{12} \end{cases} \tag{132}$$

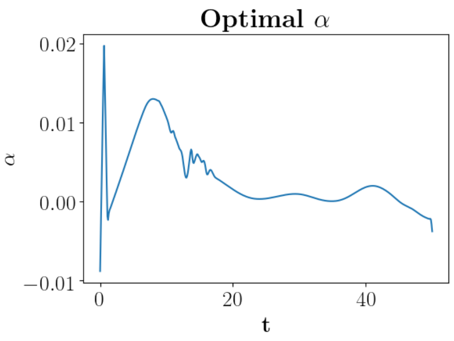
To reduce uncertainty in the optimal path planning problem, we consider the following cost functional

$$J(\boldsymbol{x}(t_f)) = \mathbb{E}\{(x(t_f)-500)^2+(y(t_f)-500)^2+(z(t_f)-500)^2\}+\frac{q}{2}\int_0^{t_f}(u_T^2(t)+u_\alpha^2(t)+u_\mu^2(t))dt, \tag{133}$$

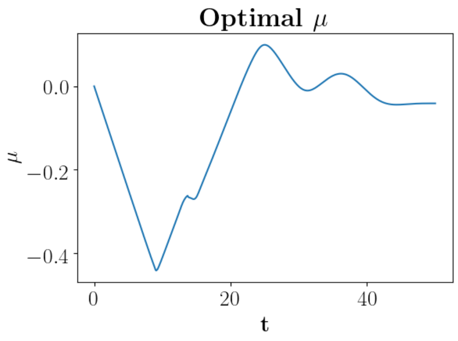
where $q = 0.01$, and we minimize it relative to $u_T(t)$, $u_\alpha(t)$ and $u_\mu(t)$ subject to the dynamics (48), the ensemble path constraints (132) and the control constraints (129). To this end, we employ $N = 1000$ randomly drawn sample paths. In the numerical simulation, we use a single shooting segment, with 3rd order RK time integrator and time step size $h = 0.1$. The final transfer time is experimentally increased to $t_f = 70.0$ using the requirement that the altitude of the drone needs to be positive (above ground) while maneuvering under uncertainty. Transfer times below $t_f = 70.0$ that do not fulfill this requirement in our numerical experiments. In Figure 62 and Figure 64 we demonstrate that trajectories under *optimal controls*



(a) Optimal thrust.



(b) Optimal angle of attack.



(c) Optimal bank angle.

Figure 63: Optimal controls we obtain to steer the UAV from an uncertain initial state to the target $(x(t_f), y(t_f), z(t_f)) = (500, 500, 500)$ $(t_f = 50)$, by minimizing the objective function (133) under ensemble path constraints (132) and control constraints (129).

land closer to the target location than trajectories under *nominal controls*. We solved the discretized constrained optimization problem by using IPOPT [69] linked to TensorFlow (see Section 2.4). In this way we were able to compute exact gradients of the discretized objective function. The resulting optimal controls are shown in Figure 63. To verify the the performance of the optimal controls under random initial conditions and uncertain parameters, we propagated in time the controlled dynamics of $N = 1000$ randomly generated trajectories, with initial conditions and parameters sampled according to (130). Such trajectories are shown in Figure 59. It is seen that the final positions of the UAV converge to a smaller region about the target location $(x(t_f), y(t_f), z(t_f)) = (500, 500, 500)$.

2.5.0.1 Common sub-expression elimination (CSE) for optimal control with path constraints

Optimal control with path constraints introduce the necessity of calculating gradients of the state variables with respect to the control function at intermediate times. These gradients can be calculated by simply storing the previously discarded intermediate values of the derivatives with respect to the control as illustrated in Table 32.

CSE-II has memory requirements scaling as $\mathcal{O}(NdmN_t^2)$, where N is the number of samples, d is the dimension of the system, m is the dimension of the control, and N_t is the number of discrete points used in the temporal discretization. The memory requirements can be reduced further by subdividing the problem into smaller batches.

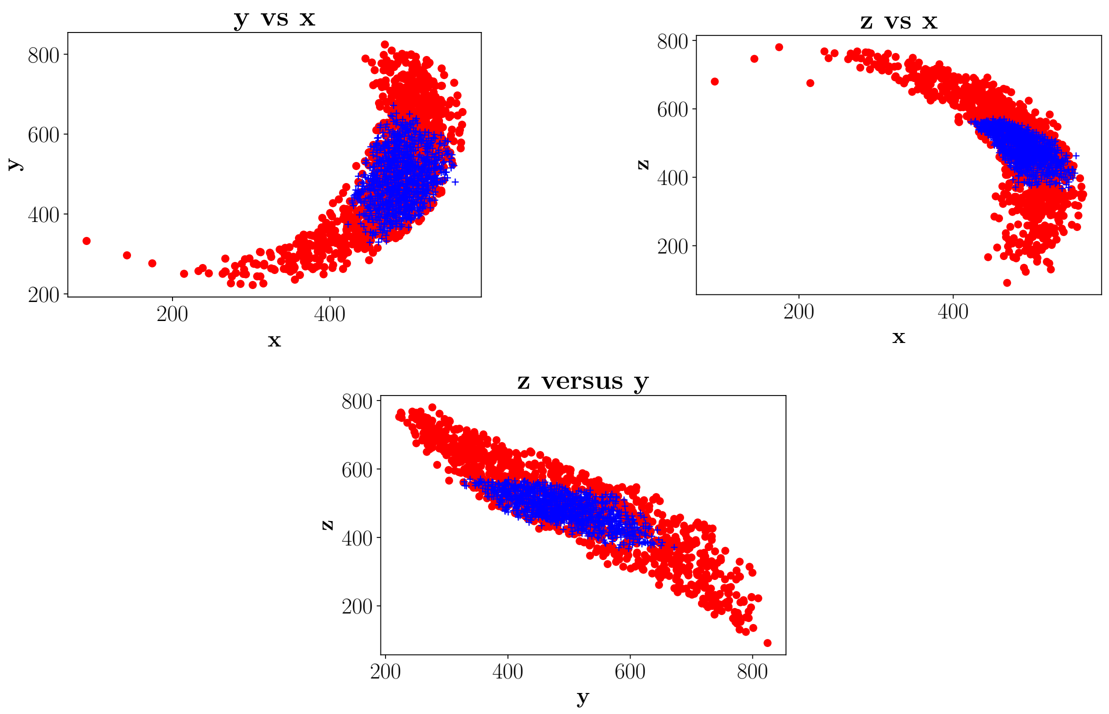


Figure 64: Verification of the optimal controls for the UAV stochastic path-planning problem under ensemble path constraints (132), and control constraints (129): nominal endpoints are in red, optimal endpoints are in blue.

Table 32: CSE-II algorithm

Step	Calculate Jacobian	Calculate control adjoint	Single Step Backpropagate	Store	Discard
1		$\frac{\partial x_{k,1}^{(i)}}{\partial u_{k,0}}$		$\frac{\partial x_{k,1}^{(i)}}{\partial u_{k,0}}$	
2	$\frac{\partial x_{k,2}^{(i)}}{\partial x_{k,1}^{(i)}}$	$\frac{\partial x_{k,2}^{(i)}}{\partial u_{k,1}}$	$\frac{\partial x_{k,2}^{(i)}}{\partial x_{k,1}^{(i)}} \frac{\partial x_{k,1}^{(i)}}{\partial u_{k,0}}$	$\frac{\partial x_{k,2}^{(i)}}{\partial u_{k,1}}, \frac{\partial x_{k,2}^{(i)}}{\partial u_{k,0}}$	$\frac{\partial x_{k,2}^{(i)}}{\partial x_{k,1}^{(i)}}$
3	$\frac{\partial x_{k,3}^{(i)}}{\partial x_{k,2}^{(i)}}$	$\frac{\partial x_{k,3}^{(i)}}{\partial u_{k,2}}$	$\frac{\partial x_{k,3}^{(i)}}{\partial x_{k,2}^{(i)}} \frac{\partial x_{k,2}^{(i)}}{\partial u_{k,1}}, \frac{\partial x_{k,3}^{(i)}}{\partial x_{k,2}^{(i)}} \frac{\partial x_{k,2}^{(i)}}{\partial u_{k,0}}$	$\frac{\partial x_{k,3}^{(i)}}{\partial u_{k,2}}, \frac{\partial x_{k,2}^{(i)}}{\partial u_{k,1}}, \frac{\partial x_{k,3}^{(i)}}{\partial u_{k,0}}$	$\frac{\partial x_{k,3}^{(i)}}{\partial x_{k,2}^{(i)}}$
...	...	...	...	...	...

2.6 Semi-stochastic optimization applied to open-loop control

It is known that for many problems with high-dimensional uncertainty, we require many sample trajectories to compute robust open-loop controls. Since the memory requirements of the problem scale with the number of samples (see Figure 56b), optimizing using the entire sample set can be inefficient or even cause the program to crash. Moreover, it is difficult to guess a priori how many samples will be needed to sufficiently characterize the uncertainty in the problem. In the machine learning community, this problem is typically dealt with by using stochastic optimization. Such methods are almost exclusively first order methods based on stochastic gradient descent [8]. For the purpose of path planning, however, we require second order optimizers which explicitly allow for bounds and arbitrary constraints. We consider a compromise between these two extremes by applying the simple idea from 2.2.5. We can optimize in multiple rounds, and in each round r generate a sample batch

$$\mathcal{S}_r = \left\{ \boldsymbol{x}_0^{(i)} \right\}_{i=1}^{|\mathcal{S}_r|}. \tag{134}$$

Like collocation points for physics informed neural networks, the samples $\boldsymbol{x}_0^{(i)}$ are essentially free to generate – they require no numerical integration. Thus we can generate a new batch $\mathcal{S}_r$ at each round r with negligible computational cost, and the batch size $|\mathcal{S}_r|$ can be selected to balance control robustness with computational efficiency.

Recently, [5] and [7] proposed multi-batch and progressive batch L-BFGS methods similar to what we have used for PINNs. The primary difference is that these methods samples a new batch $\mathcal{S}_r$ is sampled at every iteration or every few iterations, whereas we allow the optimizer to converge given a single batch. This means that we lose some of the nice generalization properties of stochastic optimization retained by [5] and [7], but we are able to immediately implement bounded and constrained optimization without working out extensive new methods and theorems.

2.6.1 Development of convergence test and sample size selection scheme

For a successful implementation of constrained progressive batch optimization, we will need useful stopping criteria and sample size selection schemes. To start, assume that the internal optimizer (for example L-BFGS-B [10], SLSQP [43], or IPOPT [69]) converges in round r. If we suppose that none of the constraints are active (dealing with active constraints is a significantly more complicated matter), then the first order necessary optimality condition holds. That is, the discretized control $\boldsymbol{u}_r$ satisfies

$$\|\nabla_{\boldsymbol{u}} J_{\mathcal{S}_r}(\boldsymbol{x}(t_f), \boldsymbol{u}_r)\| \ll 1, \tag{135}$$

where $J_{\mathcal{S}_r}(\boldsymbol{x}(t_f), \boldsymbol{u}_r)$ denotes the empirical cost estimated using the batch $\mathcal{S}_r$:

$$J_{\mathcal{S}_r}(\boldsymbol{x}(t_f), \boldsymbol{u}_r) = \mathbb{E}_{\boldsymbol{x}_0^{(i)} \in \mathcal{S}_r} J\left(\boldsymbol{\Phi}\left(\boldsymbol{x}_0^{(i)}, t_f\right), \boldsymbol{u}_r\right) = \frac{1}{|\mathcal{S}_r|} \sum_{i=1}^{|\mathcal{S}_r|} J\left(\boldsymbol{\Phi}\left(\boldsymbol{x}_0^{(i)}, t_f\right), \boldsymbol{u}_r\right). \tag{136}$$

Since $\mathcal{S}_r$ is sampled from $p_0(\boldsymbol{x}_0)$, it follows that

$$\mathbb{E}_{\mathcal{S}_r}\{J_{\mathcal{S}_r}(\boldsymbol{x}(t_f), \boldsymbol{u}_r)\} = J(\boldsymbol{x}(t_f), \boldsymbol{u}_r), \tag{137}$$

$$\text{and} \quad \mathbb{E}_{\mathcal{S}_r}\{\nabla_{\boldsymbol{u}} J_{\mathcal{S}_r}(\boldsymbol{x}(t_f), \boldsymbol{u}_r)\} = \nabla_{\boldsymbol{u}} J(\boldsymbol{x}(t_f), \boldsymbol{u}_r), \tag{138}$$

where $J(\boldsymbol{x}(t_f), \boldsymbol{u}_r)$ is the true cost and $\mathbb{E}_{\mathcal{S}_r}\{\cdot\}$ denotes the population average taken over all possible batches $\mathcal{S}_r$ sampled from $p_0(\boldsymbol{x}_0)$. Note that neither of these quantitities is directly computable.

The most immediate way to test convergence is to approximate $\nabla_{\boldsymbol{u}} J(\boldsymbol{x}(t_f), \boldsymbol{u}_r)$ by $\nabla_{\boldsymbol{u}} J_{\mathcal{V}_r}(\boldsymbol{x}(t_f), \boldsymbol{u}_r)$, where $\mathcal{V}_r$ is a validation data set with $|\mathcal{V}_r| \gg |\mathcal{S}_r|$. Then one might simply ask if

$$\|\nabla_{\boldsymbol{u}} J_{\mathcal{V}_r}(\boldsymbol{x}(t_f), \boldsymbol{u}_r)\| < \epsilon, \tag{139}$$

for a small positive parameter ϵ. This and other similar tests may be prohibitively expensive for large $|\mathcal{V}_r|$ or uninformative if $|\mathcal{V}_r|$ is too small. More importantly, this test provides no clear guidance in choosing $|\mathcal{V}_r|$, or $|\mathcal{S}_{r+1}|$ should the test fail. A similar test was proposed in [55], where we check if

$$\left\|\nabla_{\boldsymbol{u}} J_{\mathcal{S}_{r-1}}(\boldsymbol{x}(t_f), \boldsymbol{u}_{r-1}) - \nabla_{\boldsymbol{u}} J_{\mathcal{S}_r}(\boldsymbol{x}(t_f), \boldsymbol{u}_r)\right\| < \epsilon. \tag{140}$$

If eq. (140) is satisfied for a choice of small ϵ, then this indicates that the sequence $\{\boldsymbol{u}_r\}$ converges to a control $\boldsymbol{u}^*$ which is optimal for the true cost. This method still leaves us in the dark on how to properly select $|\mathcal{S}_r|$.

Instead, we now propose a more general method along the lines of [7] which provides information on how to choose the sample size $|\mathcal{S}_{r+1}|$ and is applicable to general stochastic optimization problems. The idea is simple: since we already assume (135) holds, then it should suffice to check if the population variance is small. That is, we require

$$\mathrm{Var}_{\mathcal{S}_r}\left\{\|\nabla_{\boldsymbol{u}} J_{\mathcal{S}_r}(\boldsymbol{x}(t_f), \boldsymbol{u}_r)\|\right\} \leqslant \epsilon \left\|\nabla_{\boldsymbol{u}} J(\boldsymbol{x}(t_f), \boldsymbol{u}_r)\right\|, \tag{141}$$

for $\epsilon \in (0,1)$. Evaluating the left hand side is computationally infeasible, but as in [7], we bound it by the sample variance:

$$\begin{aligned}\mathrm{Var}_{\mathcal{S}_r}\left\{\|\nabla_{\boldsymbol{u}} J_{\mathcal{S}_r}(\boldsymbol{x}(t_f), \boldsymbol{u}_r)\|\right\} \leqslant & \frac{1}{|\mathcal{S}_r|}\mathrm{Var}_{\boldsymbol{x}_0^{(i)}\in\mathcal{S}_r}\left\{\left\|\nabla_{\boldsymbol{u}} J\left(\boldsymbol{\Phi}\left(\boldsymbol{x}_0^{(i)}, t_f\right), \boldsymbol{u}_r\right)\right\|\right\} \\ \approx & \frac{1}{|\mathcal{S}_r'|}\mathrm{Var}_{\boldsymbol{x}_0^{(i)}\in\mathcal{S}_r'}\left\{\left\|\nabla_{\boldsymbol{u}} J\left(\boldsymbol{\Phi}\left(\boldsymbol{x}_0^{(i)}, t_f\right), \boldsymbol{u}_r\right)\right\|\right\},\end{aligned} \tag{142}$$

where $\mathcal{S}_r' \subseteq \mathcal{S}_r$.[5] We make this second approximation because TensorFlow and other popular automatic differentiation packages do not provide efficient ways to compute gradients of individual samples for large batches. We still cannot compute the right side of (141) directly, so we approximate it by the sample gradient and arrive at the following condition:

$$\mathrm{Var}_{\boldsymbol{x}_0^{(i)}\in\mathcal{S}_r'}\left\{\left\|\nabla_{\boldsymbol{u}} J\left(\boldsymbol{\Phi}\left(\boldsymbol{x}_0^{(i)}, t_f\right), \boldsymbol{u}_r\right)\right\|\right\} \leqslant \epsilon|\mathcal{S}_r'| \left\|\nabla_{\boldsymbol{u}} J_{\mathcal{S}_r}(\boldsymbol{x}(t_f), \boldsymbol{u}_r)\right\|. \tag{143}$$

If (143) is satisfied, then it is likely that $\|\nabla_{\boldsymbol{u}} J(\boldsymbol{x}(t_f), \boldsymbol{u}_r)\|$ is small. In other words, the solution $\boldsymbol{u}_r$ should satisfy the first order optimality conditions for the true cost $J(\boldsymbol{x}(t_f), \boldsymbol{u})$, so we can stop optimization. On the other hand, when (143) does not hold, it provides us guidance in choosing the sample size $|\mathcal{S}_{r+1}|$ for the next optimization round. If we assume that the gradient variance doesn't change significantly by choosing a larger sample. That is, if we assume

$$\frac{\mathrm{Var}_{\boldsymbol{x}_0^{(i)}\in\mathcal{S}_{r+1}}\left\{\left\|\nabla_{\boldsymbol{u}} J\left(\boldsymbol{\Phi}\left(\boldsymbol{x}_0^{(i)}, t_f\right), \boldsymbol{u}_r\right)\right\|\right\}}{\|\nabla_{\boldsymbol{u}} J_{\mathcal{S}_{r+1}}(\boldsymbol{x}(t_f), \boldsymbol{u}_r)\|} \approx \frac{\mathrm{Var}_{\boldsymbol{x}_0^{(i)}\in\mathcal{S}_r'}\left\{\left\|\nabla_{\boldsymbol{u}} J\left(\boldsymbol{\Phi}\left(\boldsymbol{x}_0^{(i)}, t_f\right), \boldsymbol{u}_r\right)\right\|\right\}}{\|\nabla_{\boldsymbol{u}} J_{\mathcal{S}_r}(\boldsymbol{x}(t_f), \boldsymbol{u}_r)\|}, \tag{144}$$

then the appropriate choice of sample size $|\mathcal{S}_{r+1}|$ is such that

$$|\mathcal{S}_{r+1}| \geqslant \frac{\mathrm{Var}_{\boldsymbol{x}_0^{(i)}\in\mathcal{S}_r'}\left\{\left\|\nabla_{\boldsymbol{u}} J\left(\boldsymbol{\Phi}\left(\boldsymbol{x}_0^{(i)}, t_f\right), \boldsymbol{u}_r\right)\right\|\right\}}{\epsilon\left\|\nabla_{\boldsymbol{u}} J_{\mathcal{S}_r}(\boldsymbol{x}(t_f), \boldsymbol{u}_r)\right\|}. \tag{145}$$

The convergence test (143) and batch size selection scheme (145) we have derived are quite general in the sense that they are not specific to solving our uncertain optimal control problem. For example, we could immediately apply these ideas to optimizing physics informed neural networks by replacing some of our hand-tuned heuristics. We do not do this here, but rather focus on the uncertain optimal control problem.

[5] When $|\mathcal{S}_r|$ itself is too small to provide useful statistical information, say $|\mathcal{S}_r| < M$, we generate a new data set $\mathcal{S}_r'$ with $|\mathcal{S}_R'| = M$ samples.

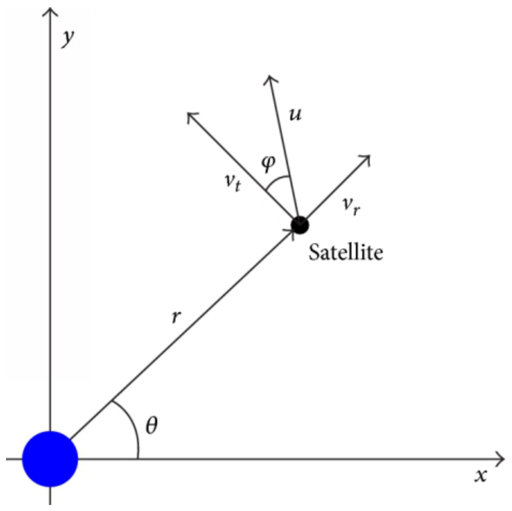


Figure 65: Dynamics of the scaled low-thrust satellite (146), from [62].

2.6.2 Numerical results

We now demonstrate the viability of our method for solving a simple low-dimensional problem. Consider the following scaled model of a satellite with low thrust from [58], [62]:

$$\begin{cases} \dot{r} = v_r, \\ \dot{\theta} = \dfrac{v_t}{r}, \\ \dot{v}_r = \dfrac{v_t^2}{r} - \dfrac{1}{r^2} + u \sin\phi, \\ \dot{v}_t = -\dfrac{v_r v_t}{r} + u\cos\phi. \end{cases} \tag{146}$$

Here r and θ are the scaled radius and angle with respect to the origin (orbited body, e.g. Earth), v_r is the radial velocity, and v_t is the tangential velocity. u and ϕ are the control variables; $u \in [0, 0.01]$ is the thrust acceleration and ϕ is the thrust angle. See Figure 65 for a graphical representation of the coordinates. Suppose the initial conditions are distributed according to

$$\begin{cases} r(t=0) = r_0(\omega) \sim \mathcal{N}(1, 0.01^2), \\ \theta(t=0) = \theta_0(\omega) \sim \mathcal{N}(0, 0.01^2), \\ v_r(t=0) = v_{r0}(\omega) \sim \mathcal{N}(0, 0.001^2), \\ v_t(t=0) = v_{t0}(\omega) \sim \mathcal{N}(1, 0.001^2). \end{cases} \tag{147}$$

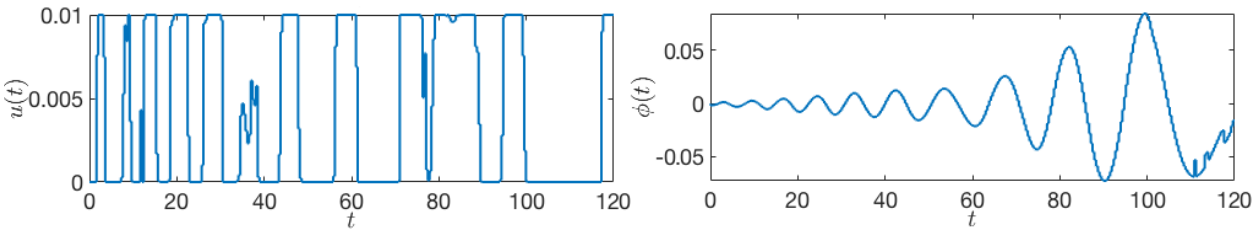
We ask that the controller take the satellite to a new orbit, while penalizing fuel consumption:

$$J(\boldsymbol{u}) = \mathbb{E}\left\{(r(t_f) - 4)^2 + (v_r(t_f) - 0)^2 + (v_t(t_f) - 0.5)^2\right\} + w\int_0^{t_f} u(t)dt, \tag{148}$$

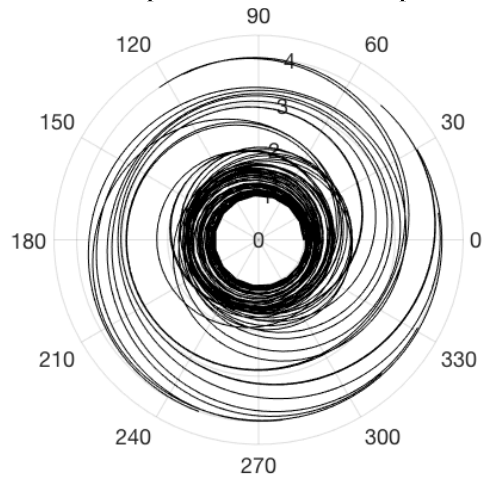
where $t_f = 120$ and w is a scalar weight which we set at $w = 10^{-3}$. We discretize the problem with RK4 in a single shooting segment containing $N_t = 300$ time intervals. Validation loss is computed against $N_v = 10^4$ samples. In Table 33, we show the results of our tests of different optimizers for solving this problem. Figure 66 shows the computed optimal controls and trajectories.

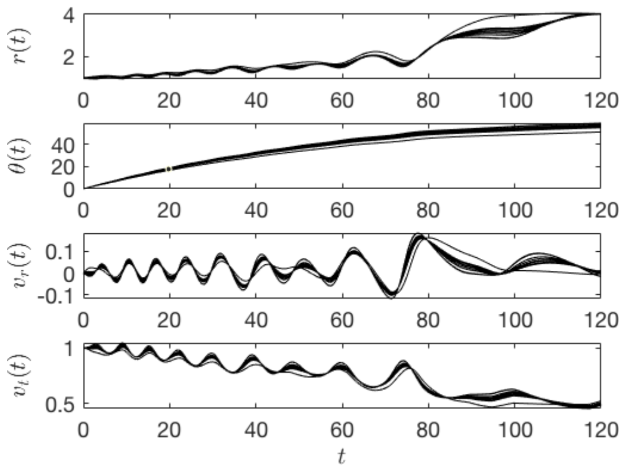
Table 33: Comparison of optimizer configurations for minimizing (148) subject to the dynamics (146) and initial condition uncertainty (147).

Optimizer	batch size	validation loss	compute time
SLSQP	$\lvert\mathcal{S}\rvert = 1$ (nominal)	8.08 e–02	423 s
SLSQP	$\lvert\mathcal{S}\rvert = 10^5$ (hardware limit)	7.57 e–04	471 s
SLSQP	$\lvert\mathcal{S}_r\rvert = \{32, \dots\}$ (progressive using (145))	8.17 e–04	481 s



(a) Optimal control profiles for the satellite problem with uncertainty. Note the roughly “bang-bang” structure found.





(b) A random selection of ten trajectories in (r, θ) phase space.

(c) State trajectories of ten random initial conditions.

Figure 66: Optimal controls and states for the satellite problem under initial condition uncertainty.

Because this is a relatively low-dimensional problem, we can get away with using only a few samples. However, when we attempted to use full batches with $\lvert\mathcal{S}\rvert > 10^5$, our hardware (NVIDIA RTX2080Ti GPU) ran out of memory, and hence the largest batch size reported is $\lvert\mathcal{S}\rvert = 10^5$. For completeness, we also attempted to solve the problem with Adam [42], a popular variant of stochastic gradient descent. Besides not being able to incorporate bounds on $u(t)$ without changing the structure of the control (for example, by using a limiting function to explicitly incorporate bounds), Adam completely failed for this problem, causing the cost function to diverge to numerical infinity. The failure of popular stochastic optimizers and the limits of hardware memory further motivate the need for a progressive batching method like we have proposed.

2.7 Verification and Validation based on the extended Pontryagin's minimum principle

We also studied a method for determining convergence based on evaluating the necessary conditions for optimality given in [54]. This involves integrating the adjoint equations backwards in time, which may be made computationally efficient using automatic differentiation. Let us consider a problem of the form

$$\begin{cases} \min\limits_{\boldsymbol{u}(t)} & \mathbb{E}\left\{F(\boldsymbol{x}(t_f,[\boldsymbol{u}]))\right\} + \displaystyle\int_0^{t_f} r(\boldsymbol{u})dt \\ \text{s.t.} & \dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x},\boldsymbol{u}), \qquad \boldsymbol{x}(0) = \boldsymbol{x}_0(\omega) \sim p(\boldsymbol{x}_0) \\ & \boldsymbol{u}^- \leqslant \boldsymbol{u} \leqslant \boldsymbol{u}^+ \qquad \text{(box constraints on the control)} \end{cases} \tag{149}$$

We now define the Hamilton's function of the constrained problem:

$$H(\boldsymbol{u},\boldsymbol{x},\boldsymbol{\lambda}) = r(\boldsymbol{u}) + \boldsymbol{\lambda}\cdot\boldsymbol{f}(\boldsymbol{x},\boldsymbol{u}). \tag{150}$$

From this we obtain Hamilton's equations

$$\begin{cases} \dot{\boldsymbol{x}} = \dfrac{\partial H}{\partial \boldsymbol{\lambda}} = \boldsymbol{f}(\boldsymbol{x},\boldsymbol{u}), \qquad \boldsymbol{x}(t=0) = \boldsymbol{x}_0(\omega), \\ \dot{\boldsymbol{\lambda}} = -\dfrac{\partial H}{\partial \boldsymbol{x}} = -\dfrac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\cdot\boldsymbol{\lambda}, \quad \boldsymbol{\lambda}(t_f) = \dfrac{\partial F(\boldsymbol{x}(t_f,[\boldsymbol{u}]))}{\partial \boldsymbol{x}}. \end{cases} \tag{151}$$

In theory, the optimal control $\boldsymbol{u}^*(t)$ for the true cost $J([\boldsymbol{u}])$ can be computed by

$$\boldsymbol{u}^*(t) = \underset{\boldsymbol{u}(t)}{\operatorname{argmin}}\, \mathbb{E}\{H(\boldsymbol{u},\boldsymbol{x},\boldsymbol{\lambda})\} \tag{152}$$

at all $t \in [0, t_f]$. Although how to solve such a problem in practice remains an open question, this offers us a way to measure the optimality of a given control. To see this, suppose we solve (149) by approximating $J([\boldsymbol{u}])$ with the sample average of $|\mathcal{S}|$ random initial conditions to obtain a candidate control $\hat{\boldsymbol{u}}(t)$. We can then sample a (large) data set $\left\{\boldsymbol{x}_0^{(i)}\right\}$, $i = 1,\dots,|\mathcal{V}_r|$, from $p_0(\boldsymbol{x}_0)$ and propagate this forward and backward through Hamilton's equations (151) to obtain $\left\{\boldsymbol{\lambda}^{(i)}(t)\right\}$ for the candidate control $\hat{\boldsymbol{u}}(t)$ (see Figure 67). With $\left\{\boldsymbol{\lambda}^{(i)}(t)\right\}$ available, we can evaluate

$$\boldsymbol{u}^*(t) = \underset{\boldsymbol{u}(t)}{\operatorname{argmin}}\, \mathbb{E}\{H(\hat{\boldsymbol{u}},\boldsymbol{x}(t,[\hat{\boldsymbol{u}}]),\boldsymbol{\lambda}(t,[\hat{\boldsymbol{u}}])\} \tag{153}$$

and check whether this control is close to $\hat{\boldsymbol{u}}(t)$. Note that the $\boldsymbol{u}^*(t)$ we compute is *not* the true optimal, since it is estimated using the Hamiltonian computed using $\hat{\boldsymbol{u}}(t)$, but if $\|\hat{\boldsymbol{u}}(t) - \boldsymbol{u}^*(t)\|$ is small then $\hat{\boldsymbol{u}}(t)$ satisfies the necessary conditions for optimality. As a simple example, we use a model of a two-wheeled UGV with differential drive given by

$$\begin{cases} \dot{x}_1 = Ru_1\cos(x_3), \\ \dot{x}_2 = Ru_1\sin(x_3), \\ \dot{x}_3 = Ru_2, \\ \dot{R} = 0. \end{cases} \tag{154}$$

Here R is the radius of the wheel, which we take to be uncertain. See Figure 68 for an illustration. Let the controls be bounded by $|u_1(t)| \leqslant 1$ and $|u_2(t)| \leqslant 1$. Suppose we have initial condition and parameter uncertainty defined by

$$\begin{cases} x_1(t=0) = x_{10}(\omega) \sim \mathcal{U}(-0.05, 0.05), \\ x_2(t=0) = x_{20}(\omega) \sim \mathcal{U}(-0.05, 0.05), \\ x_3(t=0) = x_{30}(\omega) \sim \mathcal{U}(-0.05, 0.05), \\ R = R(\omega) \sim \mathcal{U}(1, 1.5), \end{cases} \tag{155}$$

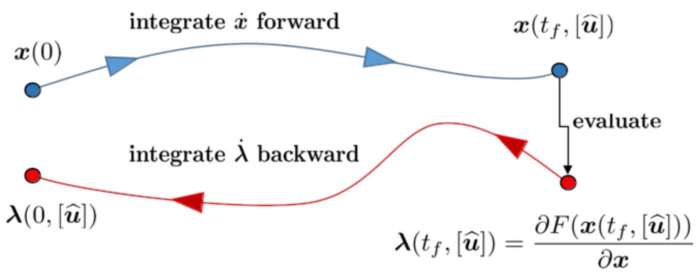

integrate $\dot{x}$ forward
$\boldsymbol{x}(0)$
$\boldsymbol{x}(t_f,[\widehat{\boldsymbol{u}}])$
evaluate
integrate $\dot{\lambda}$ backward
$\boldsymbol{\lambda}(0,[\widehat{\boldsymbol{u}}])$
$\boldsymbol{\lambda}(t_f,[\widehat{\boldsymbol{u}}]) = \frac{\partial F(\boldsymbol{x}(t_f,[\widehat{\boldsymbol{u}}]))}{\partial \boldsymbol{x}}$

Figure 67: Schematic of the forward-backward integration process for validation and verification of optimality of computed controls.

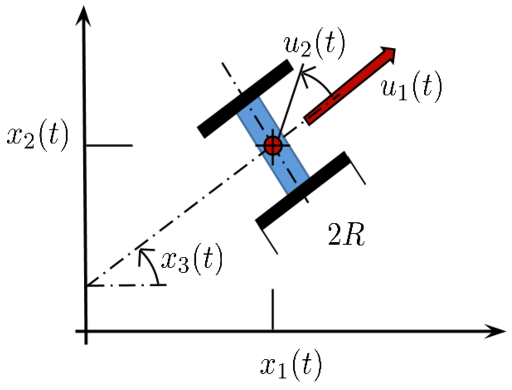


Figure 68: Dynamics of the two-wheeled UGV model (154).

and define the cost function as

$$J(\boldsymbol{u}) = \frac{1}{2}\mathbb{E}\left\{(x_1(t_f)-3)^2 + (x_2(t_f)-3)^2\right\} + \frac{w}{2}\int_0^{t_f}\left(u_1^2(t)+u_2^2(t)\right)dt, \tag{156}$$

where $t_f = 10$ and $w = 10^{-2}$. This asks the controller to drive the UGV to the point $(3,3)$ while minimizing control effort. We use Euler discretization with $N_t = 2000$ time instances and $|\mathcal{S}| = 1000$ samples to compute our candidate control. For validation and verification, we analytically derive the Hamiltonian for this system and obtain

$$H(\boldsymbol{u},\boldsymbol{x},\boldsymbol{\lambda}) = \frac{1}{2}\left(u_1^2+u_2^2\right) + \lambda_1 R u_1\cos(x_3) + \lambda_2 R u_1 \sin(x_3) + \lambda_3 R u_2. \tag{157}$$

The adjoint system is given by

$$\begin{cases} \dot{\lambda}_1 = 0, \\ \dot{\lambda}_2 = 0, \\ \dot{\lambda}_3 = \lambda_1 R u_1 \sin(x_3) - \lambda_2 R u_1 \cos(x_3). \end{cases} \tag{158}$$

We follow the process described above to measure the optimality of the computed solution, which we plot in Figure 69.

2.8 Task IV: Applications of data-driven optimal control strategies

As we anticipated in the quarter-by-quarter breakdown of the proposed research activities, we studied several nonlinear models involving swarms of attacking/defending drones, as well as disease propagation models

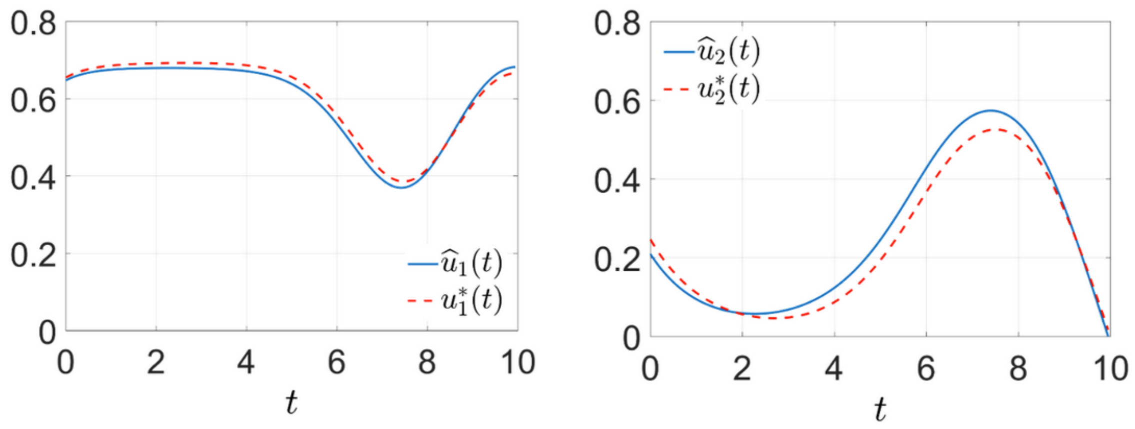


Figure 69: Evaluation of the compute control $\hat{\boldsymbol{u}}(t)$ by comparison to $\boldsymbol{u}^*(t)$ from (153) using 10^7 samples.

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.

2.8.1 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 [67] 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 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 70 diagrams the attrition interactions among HVU, attackers and defenders.

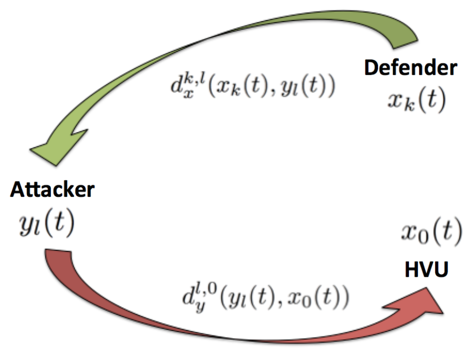


Figure 70: Interception of HVU-focused attack. $d_x^{k,l}(\boldsymbol{x}_k(t), \boldsymbol{y}_l(t))$ is the damage rate of the k-th defender against the l-th attacker; $d_y^{l,0}(\boldsymbol{y}_l(t), \boldsymbol{x}_0(t))$ is the damage rate of the l-th attacker against the HVU.

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

$$\dot{\boldsymbol{x}}_k(t) = \begin{pmatrix} \dot{x}_{k,1}(t) \\ \dot{x}_{k,2}(t) \\ \dot{x}_{k,3}(t) \end{pmatrix} = \begin{pmatrix} v_D \sin(x_{k,3}(t)) \\ v_D \cos(x_{k,3}(t)) \\ u_k(t) \end{pmatrix} = \boldsymbol{f}_k(\boldsymbol{x}_k(t), \boldsymbol{u}_k(t)), \qquad k = 1, \ldots, K \tag{159}$$

where v_D is the modulus of the defender velocity. The control $u_k(t) \in \mathbb{R}$ is subject to the constraint $u_{min} \leqslant u_k(t) \leqslant u_{max}$. 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:

$$\dot{\boldsymbol{y}}_l = \begin{pmatrix} \dot{y}_{l,1}(t, \boldsymbol{\omega}) \\ \dot{y}_{l,2}(t, \boldsymbol{\omega}) \end{pmatrix} = v_A \frac{\boldsymbol{H}^l(\boldsymbol{R}^l(t, \boldsymbol{\omega}))}{\|\boldsymbol{H}^l(\boldsymbol{R}^l(t, \boldsymbol{\omega}))\|}, \tag{160}$$

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

$$\boldsymbol{H}^l(\boldsymbol{R}^l(t, \boldsymbol{\omega})) = \begin{pmatrix} H_1^l(\boldsymbol{R}^l(t, \boldsymbol{\omega})) \\ H_2^l(\boldsymbol{R}^l(t, \boldsymbol{\omega})) \end{pmatrix}, \tag{161}$$

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

$$\boldsymbol{R}^l(t, \boldsymbol{\omega}) = \left(\boldsymbol{R}^{l,0}(t, \boldsymbol{\omega}), \boldsymbol{R}^{l,1}(t, \boldsymbol{\omega}), \ldots, \boldsymbol{R}^{l,K}(t, \boldsymbol{\omega})\right), \quad \boldsymbol{R}^{l,k}(t, \boldsymbol{\omega}) = \begin{pmatrix} y_{l,1}(t, \boldsymbol{\omega}) - x_{k,1}(t) \\ y_{l,2}(t, \boldsymbol{\omega}) - x_{k,2}(t) \end{pmatrix}. \tag{162}$$

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

$$\boldsymbol{H}^l(\boldsymbol{R}^l) = \sum_{k=0}^{K} \alpha^{l,k} \left(\|\boldsymbol{R}^{l,k}\|\right) \frac{\boldsymbol{R}^{l,k}}{\|\boldsymbol{R}^{l,k}\|} \tag{163}$$

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

$$\begin{cases} \alpha^{l,0}\left(\|\boldsymbol{R}^{l,0}\|\right) = -\frac{1}{\sigma_0} e^{\frac{-\|\boldsymbol{R}^{l,0}\|}{\sigma_0}} \\ \alpha^{l,k}\left(\|\boldsymbol{R}^{l,k}\|\right) = \frac{1}{\sigma_k} e^{\frac{-\|\boldsymbol{R}^{l,k}\|}{\sigma_k}}. \end{cases} \tag{164}$$

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

$$d_x^{k,l}(\boldsymbol{x}_k(t), \boldsymbol{y}_l(t, \boldsymbol{\omega})) = \lambda_D \left[1 + \frac{\left[\theta^{k,l}\right]^2}{3}\right]^{-3/2} \cdot \Phi\left(\frac{-r^{k,l}(t)}{\sigma_D}\right) \tag{165}$$

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

$$d_y^{l,0}(\boldsymbol{y}_l(t, \boldsymbol{\omega}), \boldsymbol{x}_0) = \lambda_A \Phi\left(\frac{-r^{l,0}(t)}{\sigma_A}\right), \tag{166}$$

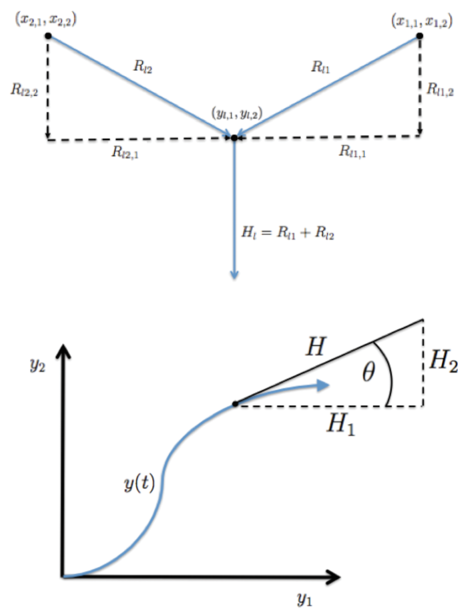


Figure 71: Averaged herding heading example for one attacker and two defenders

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 [67] that the probability of survival of the l-th attacker satisfies differential equation

$$\frac{dQ_l(t,\boldsymbol{\omega})}{dt} = -Q_l(t,\boldsymbol{\omega}) \sum_{k=1}^{K} d_x^{k,l}(\boldsymbol{x}_k(t), \boldsymbol{y}_l(t,\boldsymbol{\omega})) \tag{167}$$

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

$$\frac{dP_0(t,\boldsymbol{\omega})}{dt} = -P_0(t,\boldsymbol{\omega}) \sum_{l=1}^{L} Q_l(t,\boldsymbol{\omega}) d_y^{l,0}(\boldsymbol{y}_l(t,\boldsymbol{\omega}), \boldsymbol{x}_0), \tag{168}$$

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

2.8.1.3 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 probability of survival, i.e., minimizes

$$J = \int_\Omega \left[1 - P_0(t_f, \boldsymbol{\omega})\right] \phi(\boldsymbol{\omega}) d\boldsymbol{\omega} \tag{169}$$

subject to

$$\begin{cases} \dot{\boldsymbol{x}}_k(t) = \boldsymbol{f}_k(\boldsymbol{x}_k(t), u_k(t)), & \boldsymbol{x}_k(0) = \boldsymbol{x}_{k0}, \\ \dot{\boldsymbol{y}}_l(t,\boldsymbol{\omega}) = \boldsymbol{g}_l(\boldsymbol{x}_1(t), \ldots, \boldsymbol{x}_K(t), \boldsymbol{y}_l(t,\boldsymbol{\omega})), & \boldsymbol{y}_l(0,\boldsymbol{\omega}) = \boldsymbol{\gamma}_l(\boldsymbol{\omega}), \\ \dot{Q}_l(t,\boldsymbol{\omega}) = -Q_l(t,\boldsymbol{\omega}) \displaystyle\sum_{k=1}^{K} d_x^{k,l}(\boldsymbol{x}_k(t), \boldsymbol{y}_l(t,\boldsymbol{\omega})), & Q_l(0,\boldsymbol{\omega}) = 1, \\ \dot{P}_0(t,\boldsymbol{\omega}) = -P_0(t,\boldsymbol{\omega}) \displaystyle\sum_{l=1}^{L} Q_l(t,\boldsymbol{\omega}) d_y^{l,0}(\boldsymbol{y}_l(t,\boldsymbol{\omega}), \boldsymbol{x}_0), & P_0(0,\boldsymbol{\omega}) = 1, \end{cases} \tag{170}$$

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 34: Definition of all phase variables and parameters appearing in the nonlinear system (173).

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

The HUV attach interception problem is ideal for testing the proposed data-driven optimal control architecture. In particular, in [67] 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.

2.8.2 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 34. 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

$$\frac{ds_i}{dt} = -s_i \sum_{j=1}^{n} A_{ij} B_{ij} x_j, \tag{171}$$

$$\frac{dx_i}{dt} = s_i \sum_{j=1}^{n} A_{ij} B_{ij} x_j - \gamma x_i, \tag{172}$$

$$\frac{dr_i}{dt} = \gamma x_i, \tag{173}$$

where $s_i + x_i + r_i = 1$, and $i = 1, \dots, n$. 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 (173) 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.

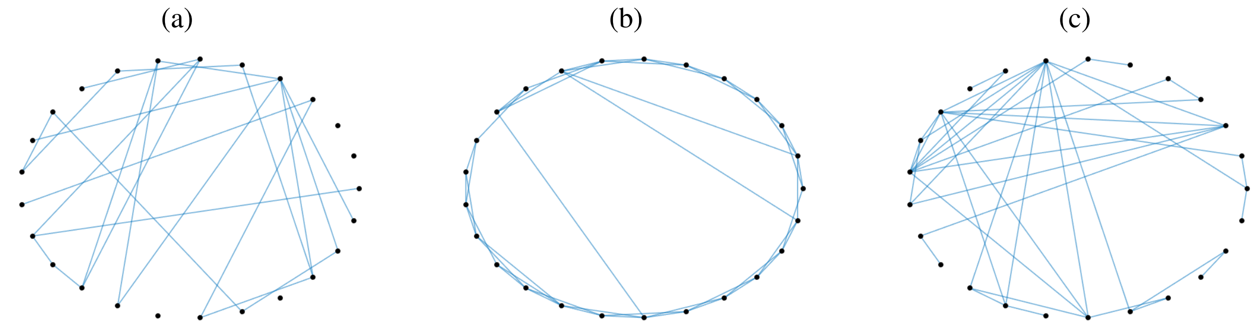


Figure 72: Sample configurations of a Poisson random graph (a), a Watts-Strogatz random graph (b), and an exponential random graph (c) with $N = 25$ vertices.

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

$$p_k = \binom{n-1}{k} p^k (1-p)^{n-1-k} \tag{174}$$

  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

$$p_k = \sum_{n=0}^{\min(k-K/2,K/2)} \binom{K/2}{n} (1-\beta)^n \beta^{K/2-n} \frac{(\beta K/2)^{k-K/2-n}}{(k-K/2-n)!} e^{-\beta K/2}, \tag{175}$$

  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

$$p_k = (1-e^{-\lambda})e^{-\lambda k}, \tag{176}$$

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

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

$$B_{ij} = \begin{cases} 0, & (i \vee j) \text{ practiced safe sex} \vee (i \vee j) \text{ is protected} \\ & \vee\ (i \wedge j) \text{ are susceptible} \vee (i \wedge j) \text{ are infected} \\ 1-(1-\beta)^{\eta}, & \text{otherwise} \end{cases}. \tag{177}$$

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 (173) are

$$s_i(0) = 1 - c/n, \qquad x_i(0) = c/n, \qquad r_i(0) = 0. \tag{178}$$

2.9 Solution of Hamilton-Jacobi-Bellman equations with physics-informed neural networks

In Q1-Q4, we focused our effort on developing computational algorithms for *open-loop* control under uncertainty. Such open-loop controls were designed to explicitly account for and mitigate uncertainty, but for implementation in real systems one would still prefer *closed-loop (feedback) controls* which are inherently robust to initial condition uncertainty and disturbances. To calculate optimal feedback controls, in Q5 we studied the possibility of solving the Hamilton-Jacobi-Bellman (HJB) equation. Such equation is a hard-to-solve nonlinear PDE in n dimensions (n is the dimension of the dynamical system) plus time, that arises naturally from the optimal control problem and allows to compute directly feedback controls (once the solution is know). The HJB equation has been challenging the computational mathematics community for decades. One of the main difficulties is related its high-dimensionality. Classical numerical discretizations based on tensor product representations cannot be in practice, since the number of degrees of freedom grows exponentially fast with the dimension of the system. There is an extensive literature on methods of finding approximate solutions for HJB equations. We have no intention to give a full review of existing results except for a short list of some related publications, [3, 11, 17, 20, 32, 36, 38, 47, 50, 61]. In addition to suffering from the curse of dimensionality, most existing methods suffer one or more of the following drawbacks: the accuracy of the solution cannot be verified for general systems; the solution is valid only in a small neighborhood of a point; or the system model must have certain special algebraic structure, the solution to the HJB equation is rough function in the presence of optimization constraints.

In [37, 38], semi-global solutions to HJB equations are computed by constructing sparse discretization of the state space, then solving a two-point boundary value problem (TPBVP) to obtain the solution at each point on the grid. We previously studied the TPBVP in a different context in Section 2.7. Critically, the TPBVP is *causality-free*, i.e. the solution at each grid point is computed without using the value of the solution at other nearby points. Causality-free algorithms have been used to solve various types of PDEs, such as conservation laws [39], HJ and HJB equations [31, 14], and semilinear parabolic PDEs [29]. The causality-free property is achieved by means of Hopf formula and convex optimization in [14], backward stochastic differential equations in [29], and Pontryagin's minimum principle in [38]. Causality-free algorithms are attractive because the computation does not depend on a grid, and hence can be applied to high dimensional problems. Some causality-free methods are slow for online computation, but they are perfectly parallel so can be used to generate large data sets offline. Such data sets can then be used to construct faster solutions such as sparse grid interpolation [37, 38] or neural networks, as we do here.

In this Section, we develop a new computational method for solving high-dimensional HJB equations and for calculating fully nonlinear optimal feedback controls. Although the problem we solve is somewhat different than in our previous work, our computational algorithm utilizes many of the same elements in a different context. At the core of our method is a type of physics-informed neural network (PINN), first discussed in Section 2.2, trained to model the value function which is the solution to the HJB PDE. Data to train the NN is generated by solving the TPBVP derived from Pontryagin's minimum principle. In addition, we apply the semi-stochastic optimization technique introduced in Section 2.6 to generate data adaptively. With the aid of the partially-trained NN, we use this strategy to start from a tiny initial data set and progressively build up large, enriched data sets.

We demonstrate the effectiveness of the PINN-HJB technique we developed by applying it to the optimal attitude control of a rigid-body satellite equipped with momentum wheels. The associated HJB equation has $n = 6$ spatial dimensions and $m = 3$ control signals . This example allows us to demonstrate several advantages and potential capabilities of the method we propose, including solving HJB equations with high dimensions with validated levels of accuracy, computationally efficient NN-based feedback control for real-time applications, and progressive generation of rich data sets. Similar to [38], our method is semi-global, i.e., the solution is computed and validated over an entire region rather than a local neighborhood around an operating point. Further, we achieve a comparable level of accuracy to that in [38], but require far fewer data points to do so.

2.9.1 A causality-free method for HJB

Consider the open-loop optimal control problem

$$\begin{cases} \underset{\boldsymbol{u}(t)}{\text{minimize}} \quad F(\boldsymbol{x}(t_f)) + \displaystyle\int_{t_0}^{t_f} L(t, \boldsymbol{x}, \boldsymbol{u}) dt \\ \text{subject to} \quad \dot{\boldsymbol{x}} = \boldsymbol{f}(t, \boldsymbol{x}, \boldsymbol{u}), \qquad \boldsymbol{x}(0) = \boldsymbol{x}_0 \end{cases} \tag{179}$$

Here $\boldsymbol{x}(t) : [t_0, t_f] \to \mathcal{X} \subseteq \mathbb{R}^n$ is the state of the system, $\boldsymbol{u}(t) : [t_0, t_f] \to \mathcal{U} \subseteq \mathbb{R}^m$ is the control, $\boldsymbol{f}(t, \boldsymbol{x}, \boldsymbol{u}) : [t_0, t_f] \times \mathcal{X} \times \mathcal{U} \to \mathbb{R}^n$ is the vector field (Lipschitz continuous), $F(\boldsymbol{x}(t_f)) \in \mathbb{R}$ is the terminal cost, and $L(t, \boldsymbol{x}, \boldsymbol{u}) \in \mathbb{R}$ is the running cost. For simplicity we let the final time t_f be fixed. This identifies a well-defined control time horizon $[t_0, t_f]$. For a given initial condition $\boldsymbol{x}(t_0) = \boldsymbol{x}_0$, many methods exist to compute the optimal open-loop solution

$$\boldsymbol{u}^* = \boldsymbol{u}^*(t). \tag{180}$$

For instance one can use the multi-shooting algorithm we developed in Q3-Q4 (see Section 2.4.1). It is well-known, however, that open-loop controls are not robust to model uncertainty or disturbances, and must be recomputed for each new initial condition in a deterministic setting. For slowly evolving systems, it is possible to use open-loop model predictive control [21] by recomputing (180) within small time intervals in $[t_0, t_f]$, but for most applications one typically desires optimal feedback controls

$$\boldsymbol{u}^* = \boldsymbol{u}^*(t, \boldsymbol{x}). \tag{181}$$

which solves the close-loop (feedback) optimal control problem

$$\begin{cases} \underset{\boldsymbol{u}(t,\boldsymbol{x})}{\text{minimize}} \quad F(\boldsymbol{x}(t_f)) + \displaystyle\int_{t}^{t_f} L(\tau, \boldsymbol{x}, \boldsymbol{u}) d\tau \\ \text{subject to} \quad \dot{\boldsymbol{x}}(\tau) = \boldsymbol{f}(\tau, \boldsymbol{x}, \boldsymbol{u}) \\ \qquad\qquad\quad \boldsymbol{x}(t) = \boldsymbol{x} \end{cases} \tag{182}$$

Using dynamic programming, it can be shown that computing the optimal feedback control (181) using (182) is equivalent to solving the following Hamilton-Jacobi-Bellman (HJB) equation

$$\begin{cases} \dfrac{\partial V(t,\boldsymbol{x})}{\partial t} + \min\limits_{\boldsymbol{u}(t,\boldsymbol{x})} \{L(\tau,\boldsymbol{x},\boldsymbol{u}) + \nabla V(t,\boldsymbol{x}) \cdot \boldsymbol{f}(\tau,\boldsymbol{x},\boldsymbol{u})\} = 0 \\ V(t_f,\boldsymbol{x}) = F(\boldsymbol{x}) \end{cases} \tag{183}$$

The field $V(t,\boldsymbol{x})$ is the optimal cost of (182) and is called the value function. Since Eq. (183) is a partial differential equation (PDE) in n dimensions plus time, the size of the discretized equation increases exponentially with n, making even moderately high dimensional problems computationally intractable.

In [37, 38], the authors make efforts to overcome the curse of dimensionality by avoiding direct discretization the solution of the HJB equation. Instead, they construct a sparse grid of initial conditions and, for each point in the grid, solve the open-loop optimal control problem (179) as a TPBVP. Then they interpolate the cost and apply Pontryagin's Maximum Principle (PMP) to obtain the feedback control. However, even using a sparse grid collocation method the number of points grows as

$$O\left(N(\log N)^{n-1}\right), \tag{184}$$

where n is the state dimension and N is the number of points in each dimension. Thus, using the method of characteristics on sparse grids [37, 38], could yield a prohibitively large number of BVPs, especially in high-dimensions. In addition, even with the solution available, evaluating the feedback control requires interpolation of the n-dimensional cost. This can be expensive and inaccurate for large n, making such feedback implementation infeasible. Hereafter, we adopt the same TPBVP and its computational algorithm in [37, 38], but instead of sparse grid interpolation we use data to train a NN to approximate the value function $V(t,\boldsymbol{x})$.

2.9.2 Dynamic programming and two-point boundary value problems for feedback control

Let us provide a brief overview of the connection between dynamic programming and the TPBVP, as well as techniques to solve it numerically. Following the standard procedure in optimal control, we first define the Hamiltonian

$$H(t,\boldsymbol{x},\boldsymbol{\lambda},\boldsymbol{u}) = L(t,\boldsymbol{x},\boldsymbol{u}) + \boldsymbol{\lambda}^T \boldsymbol{f}(t,\boldsymbol{x},\boldsymbol{u}), \tag{185}$$

where $\boldsymbol{\lambda}(t) : [t_0, t_f] \to \mathbb{R}^n$ is the co-state. It is well-known that the optimal control satisfies

$$\boldsymbol{u}^*(t,\boldsymbol{x},\boldsymbol{\lambda}) = \arg\min_{\boldsymbol{u}} H(t,\boldsymbol{x},\boldsymbol{\lambda},\boldsymbol{u}). \tag{186}$$

Next, we define the value function

$$V(t,\boldsymbol{x}) = \inf_{\boldsymbol{u}} \left\{ F(\boldsymbol{x}(t_f)) + \int_t^{t_f} L(\tau,\boldsymbol{x},\boldsymbol{u}) d\tau \right\}. \tag{187}$$

The value function satisfies the HJB equation (183), or equivalently

$$\begin{cases} \dfrac{\partial V(t,\boldsymbol{x})}{\partial t} + H^*\left(t,\boldsymbol{x},\nabla V(t,\boldsymbol{x})\right) = 0, \\ V(t_f,\boldsymbol{x}) = F(\boldsymbol{x}), \end{cases} \tag{188}$$

where $H^*(t,\boldsymbol{x},\boldsymbol{\lambda}) = H(t,\boldsymbol{x},\boldsymbol{\lambda},\boldsymbol{u}^*)$ is the Hamiltonian evaluated at the optimal control (assuming it is available). If (188) can be solved, then the optimal control can be computed by substituting

$$\boldsymbol{\lambda}(t) = \nabla V(t,\boldsymbol{x}) \tag{189}$$

into (186), i.e.,

$$\boldsymbol{u}^*(t,\boldsymbol{x}) = \boldsymbol{u}^*\left(t,\boldsymbol{x},\nabla V\right) = \arg\min_{\boldsymbol{u}} H\left(t,\boldsymbol{x},\nabla V,\boldsymbol{u}\right). \tag{190}$$

This means that with ∇V available, the optimal feedback control is obtained as the solution of a (usually straightforward) optimization problem.

To implement (190), we need an efficient way to calculate the value function and its gradient. Rather than solving the full HJB equation (188) on a grid in the state space, we leverage the fact that its characteristics curves evolve according to

$$\begin{cases} \dot{\boldsymbol{x}}(\tau) = \dfrac{\partial H}{\partial \boldsymbol{\lambda}} = \boldsymbol{f}(\tau,\boldsymbol{x},\boldsymbol{u}^*(\tau,\boldsymbol{x},\boldsymbol{\lambda})), \\ \dot{\boldsymbol{\lambda}}(\tau) = -\dfrac{\partial H}{\partial \boldsymbol{x}}(\tau,\boldsymbol{x},\boldsymbol{\lambda},\boldsymbol{u}^*(\tau,\boldsymbol{x},\boldsymbol{\lambda})), \\ \dot{v}(\tau) = L(\tau,\boldsymbol{x},\boldsymbol{u}^*(\tau,\boldsymbol{x},\boldsymbol{\lambda})), \end{cases} \tag{191}$$

with two-point split boundary conditions

$$\begin{cases} \boldsymbol{x}(t_0) = \boldsymbol{x}_0, \\ \boldsymbol{\lambda}(t_f) = \nabla F(\boldsymbol{x}(t_f)), \\ v(t_0) = 0. \end{cases} \tag{192}$$

For any given initial condition $\boldsymbol{x}_0$, the optimal control and value function *along that characteristic* are then given by

$$\boldsymbol{u}^*(\tau,\boldsymbol{x}) = \boldsymbol{u}^*(\tau;\boldsymbol{x}_0,\boldsymbol{\lambda}), \qquad V(t_0,\boldsymbol{x}_0) = v(t_f) + F(\boldsymbol{x}(t_f)). \tag{193}$$

The necessary condition (191-192) is well-known in optimal control theory. However, numerically solving such TPBVP numerically is often very difficult. So far, there is no general algorithm that is reliable and fast enough for real-time applications. However, in our approach the real-time computation is done by a pre-trained NN. In practice, we solve the TPBVP *offline* to generate a data set which we use to train and validate NN. The computational algorithm to solve (191-192) is based on a four-point Lobatto IIIA discretization. This is a collocation formula which provides a solution that is fifth-order accurate (see [41] for more detail). But the algorithm is highly sensitive to the initial guess: there is no guarantee of convergence with an arbitrary initial guess. Furthermore, convergence is increasingly dependent on a good initial guess as we increase the length of the time interval.

To overcome this difficulty, to generate the initial data set we employed the time-marching trick from [37, 38] in which the solution grows from an initially short time interval to the final time t_f. More specifically, we choose a time sequence

$$t_0 < t_1 < t_2 < \cdots < t_k = t_f,$$

in which t_1 is small. For the short time interval $[t_0, t_1]$, the TPBVP solver converges given most initial guesses near the initial state $\boldsymbol{x}$. Then, the resulting trajectory is rescaled over the longer time interval $[t_0, t_2]$. The rescaled trajectory is used as the initial guess to find a solution of the TPBVP for $t_0 \leqslant t \leqslant t_2$. We repeat this process until $t_k = t_f$, at which we obtain the full solution. To achieve convergence, it is necessary to tune the time sequence $\{t_i\}_{1\leqslant i \leqslant t_k}$ while maintaining acceptable efficiency. Computing many solutions in this ways becomes expensive, which means that generating the large data sets needed to efficiently train a NN can be hard to obtain. Therefore, we begin by generating a small data set and adaptively adding more points during training. The key to doing this efficiently is simulating the system dynamics, using the partially-trained NN to close the loop. This quickly provides good guesses for the optimal state $\boldsymbol{x}^*(\tau)$ and co-state $\boldsymbol{\lambda}^*(\tau)$ over the entire time interval $[t_0, t_f]$, so that we can immediately solve (191-192) for all of

$[t_0, t_f]$. Details are presented in Section 2.9.3, and numerical comparisons between this method and the time-marching trick are given in Section 2.9.5.

2.9.3 Neural network (NN) approximation of the value function

2.9.3.1 Physics-informed learning of the value function The method we propose to compute the solution to the HJB equation relies on modeling the value function (187) over the domain $\mathcal{X} \subset \mathbb{R}^n$. This is done using PINNs similar to those use in Section 2.2. To this end, let us denote by $V(\cdot)$ the value function which we want to approximate, and let $V^{NN}(\cdot)$ be its NN representation, and denote by $\boldsymbol{\theta}$ the parameters (weights and biases) of the NN which need to be optimized. We again use hyperbolic tangent function as activation function in all the hidden layers. By solving the TPBVP (191-192) for a set of (randomly sampled) initial conditions, we get a data set of the form

$$\mathcal{D} = \left\{ \left(\left(t^{(i)}, \boldsymbol{x}^{(i)} \right), V^{(i)} \right) \right\}_{i=1}^{N_d}, \tag{194}$$

where $\left(t^{(i)}, \boldsymbol{x}^{(i)}\right)$ are the inputs, $V^{(i)} = V\left(t^{(i)}, \boldsymbol{x}^{(i)}\right)$ are the outputs to be modeled, and $i = 1, 2, \ldots, N_d$ are the indices of the data points. This data set is obtained by solving many instances of the two-point BVP (191) with randomly sampled initial conditions. In the naïve implementation, the NN is trained by solving the nonlinear regression problem,

$$\min_{\boldsymbol{\theta}} \frac{1}{N_d} \sum_{i=1}^{N_d} \left[V^{(i)} - V^{NN}\left(t^{(i)}, \boldsymbol{x}^{(i)}\right) \right]^2. \tag{195}$$

Motivated by the success of using PINNs for approximating data-driven solutions of PDF equations, we expect that we can improve on the rudimentary loss function in (195) by incorporating information about the underlying dynamics. Although it is possible to impose the HJB equation (188) by means of a residual penalty term as was done in Section 2.2 (see also [61]), the residual must be evaluated over a large number of collocation points and can be rather expensive to compute. Instead, we take the simpler approach of modeling the co-state $\boldsymbol{\lambda}(t; \boldsymbol{x})$ along with the value function. Specifically, we know that the co-state must satisfy (189), so we encourage the NN to minimize

$$\left\| \boldsymbol{\lambda}(t; \boldsymbol{x}) - \nabla V^{NN}(t, \boldsymbol{x}) \right\|, \tag{196}$$

where $\nabla V^{NN}(\cdot)$ is the gradient of the NN representation of the value function with respect to the state. Co-state data $\boldsymbol{\lambda}(t, \boldsymbol{x})$ is obtained for each trajectory as a natural product of solving the TPBVP (191-192). Hence, we have the augmented data set

$$\mathcal{D}^{\boldsymbol{\lambda}} = \left\{ \left(\left(t^{(i)}, \boldsymbol{x}^{(i)} \right), \left(V^{(i)}, \boldsymbol{\lambda}^{(i)} \right) \right) \right\}_{i=1}^{N_d}, \tag{197}$$

where $\boldsymbol{\lambda}^{(i)} = \boldsymbol{\lambda}\left(t^{(i)}; \boldsymbol{x}^{(i)}\right)$. We now define the *physics-informed learning problem*,

$$\min_{\boldsymbol{\theta}} \mathcal{L}\left(\boldsymbol{\theta}; \mathcal{D}^{\boldsymbol{\lambda}}\right), \quad \text{where} \quad \mathcal{L}\left(\boldsymbol{\theta}; \mathcal{D}^{\boldsymbol{\lambda}}\right) = \text{loss}_V\left(\boldsymbol{\theta}; \mathcal{D}^{\boldsymbol{\lambda}}\right) + \mu \text{loss}_{\boldsymbol{\lambda}}\left(\boldsymbol{\theta}; \mathcal{D}^{\boldsymbol{\lambda}}\right). \tag{198}$$

In (198) $\mu \geqslant 0$ is a scalar weight, the loss with respect to data is

$$\text{loss}_V\left(\boldsymbol{\theta}; \mathcal{D}^{\boldsymbol{\lambda}}\right) = \frac{1}{N_d} \sum_{i=1}^{N_d} \left[V^{(i)} - V^{NN}\left(t^{(i)}, \boldsymbol{x}^{(i)}\right) \right]^2, \tag{199}$$

and the co-state gradient loss regularization term is defined as

$$\text{loss}_{\boldsymbol{\lambda}}\left(\boldsymbol{\theta};\mathcal{D}^{\boldsymbol{\lambda}}\right)=\frac{1}{N_d}\sum_{i=1}^{N_d}\left\|\boldsymbol{\lambda}^{(i)}-\nabla V^{NN}\left(t^{(i)},\boldsymbol{x}^{(i)}\right)\right\|^2. \tag{200}$$

We randomly partition the given data set (197) into a training set $\mathcal{D}^{\boldsymbol{\lambda}}_{\text{train}}$ and validation set $\mathcal{D}^{\boldsymbol{\lambda}}_{\text{val}}$. During optimization, the loss functions (199) and (200) are calculated with respect to the training data $D^{\boldsymbol{\lambda}}_{\text{train}}$. We then evaluate the network's performance against the validation data $\mathcal{D}^{\boldsymbol{\lambda}}_{\text{val}}$, which we did not observe during the training phase. We emphasize that a NN trained to minimize (198) learns not just to fit the value data, but it is rewarded for doing so in a way that respects the underlying structure of the problem. This physics-informed regularization takes the known problem structure into account, so it is preferable to the usual L_1 or L_2 regularization, which are based on the (heuristic) principle that simpler representations of data are likely to generalize better. Furthermore, by using information about the co-state we make the most use out of a potentially small data set.

2.9.3.2 Closed-loop system Once the NN is trained, evaluating ∇V^{NN} at new inputs $(t,\boldsymbol{x})$ points is extremely efficient – and since we minimized the gradient loss (200) during training, we also expect ∇V^{NN} to approximate the exact gradient well. At runtime, whenever the feedback control needs to be computed, we evaluate $\nabla V^{NN}(t,\boldsymbol{x})$ using our physics-informed neural net, and then solve (190) based on this approximation. For many problems of interest, the optimization problem (190) admits a semi-analytic solution. In particular, for the important class of control affine systems with running cost convex in $\boldsymbol{u}$, we can solve (190) analytically. To show this, suppose that the system dynamics can be written in the form

$$\dot{\boldsymbol{x}}=\boldsymbol{f}(t,\boldsymbol{x})+\boldsymbol{g}(t,\boldsymbol{x})\boldsymbol{u}(t,\boldsymbol{x}), \tag{201}$$

where $\boldsymbol{g}(t,\boldsymbol{x})$ is a matrix $n\times m$. Further, suppose that the running cost is of the form

$$L(t,\boldsymbol{x},\boldsymbol{u})=h(t,\boldsymbol{x})+\boldsymbol{u}^T\boldsymbol{W}\boldsymbol{u}, \tag{202}$$

for some function $h(t,\boldsymbol{x})$ and some positive-definite weight matrix $\boldsymbol{W}\in\mathbb{R}^{m\times m}$. The Hamiltonian (185) corresponding the the affine control system (201) can be written as

$$H(t,\boldsymbol{x},\boldsymbol{\lambda},\boldsymbol{u})=h(t,\boldsymbol{x})+\boldsymbol{u}^T\boldsymbol{W}\boldsymbol{u}+\boldsymbol{\lambda}^T\boldsymbol{f}(t,\boldsymbol{x})+\boldsymbol{\lambda}^T\boldsymbol{g}(t,\boldsymbol{x})\boldsymbol{u}. \tag{203}$$

Next we apply the Pontryagin stationary principle, which requires that

$$\boldsymbol{0}=\left.\frac{\partial H}{\partial\boldsymbol{u}}\right|_{\boldsymbol{u}=\boldsymbol{u}^*}=2\boldsymbol{W}\boldsymbol{u}^*+\boldsymbol{g}^T(t,\boldsymbol{x})\boldsymbol{\lambda}. \tag{204}$$

Letting $\boldsymbol{\lambda}=\nabla V(t,\boldsymbol{x})$ and solving for $\boldsymbol{u}^*$ gives us the feedback optimal control in an analytic form

$$\boldsymbol{u}^*(t,\boldsymbol{x})=-\frac{1}{2}\boldsymbol{W}^{-1}\boldsymbol{g}^T(t,\boldsymbol{x})\nabla V(t,\boldsymbol{x}). \tag{205}$$

Plugging this expression into the forward/backward system (191) yields an explicit two point boundary value problem which can be solved using the methods outlined in the next Section. Alternatively, we can consider the HJB equation (188) with Hamiltonian (203) evaluated at the optimal feedback signal (205). This makes (188) explicit, and it yields a nonlinear evolution equation for the value function $V(t,\boldsymbol{x})$ which can be solved, e.g., using numerical tensor methods (see Section 2.1).

2.9.4 Adaptive sampling and fast solution of the two-point BVP

Generating just a single data point $\left\{t^{(i)}, \boldsymbol{x}^{(i)}, V^{(i)}, \boldsymbol{\lambda}^{(i)}\right\}$ requires solving the TPBVP (191), which is not a straightforward task. Hence, it is difficult to generate large data sets by repeatedly solving (191). On the other hand, we need lots of data points to adequately represent the value function $V(t, \boldsymbol{x})$, especially in high dimensions. Hence, we developed a new approach that allows us to train the NN with small data sets and simultaneously generate new data points in a smart way. We accomplished effective training with small data sets by incorporating information about the co-state as discussed in Section 2.9.3, but also by combining progressive batching with an efficient adaptive sampling technique.

To illustrate the method, suppose we are given a small initial data set, denoted by $\mathcal{D}_1$. We begin by training a low-fidelity NN model of the value function. It is not necessary to use first-order optimization methods like SGD or Adam, since for small data sets and NNs second-order optimization methods like L-BFGS tend to converge more quickly. After allowing the internal optimizer to converge, we apply (143) to estimate how many samples are needed for convergence. In particular, the convergence condition in case can be written as

$$\operatorname{Var}_{\left(t^{(i)}, \boldsymbol{x}^{(i)}\right) \in \mathcal{D}_r}\left\{\left\|\nabla_{\boldsymbol{\theta}} \mathcal{L}\left(\boldsymbol{\theta}_r ;\left(t^{(i)}, \boldsymbol{x}^{(i)}\right)\right)\right\|\right\} \leqslant \epsilon\left|\mathcal{D}_r\right|\left\|\nabla_{\boldsymbol{\theta}} \mathcal{L}\left(\boldsymbol{\theta}_r ; \mathcal{D}_r\right)\right\|, \tag{206}$$

where $\nabla_{\boldsymbol{\theta}}(\cdot)$ is the gradient with respect to the NN parameters, $\boldsymbol{\theta}$, r is the training round, $\mathcal{L}(\cdot)$ is the physics-informed loss defined in eq. (198), $\epsilon > 0$ is a parameter, and $|\mathcal{D}_r|$ is the cardinality (number of elements) of $\mathcal{D}_r$. If failed, the test leads to a criterion for selecting the next round's sample size $|\mathcal{D}_{r+1}|$, which should satisfy

$$\left|\mathcal{D}_{r+1}\right| \geqslant \frac{\operatorname{Var}_{\left(t^{(i)}, \boldsymbol{x}^{(i)}\right) \in \mathcal{D}_r}\left\{\left\|\nabla_{\boldsymbol{\theta}} \mathcal{L}\left(\boldsymbol{\theta}_r ;\left(t^{(i)}, \boldsymbol{x}^{(i)}\right)\right)\right\|\right\}}{\epsilon\left\|\nabla_{\boldsymbol{\theta}} \mathcal{L}\left(\boldsymbol{\theta}_r ; \mathcal{D}_r\right)\right\|}. \tag{207}$$

This allows us to generate only the minimum number of samples necessary to successfully complete the training of the NN The estimates (206) and (207) are developed by assuming a uniform sampling from the domain. However, in practice all the data we generate will be new, so we can choose to generate new data where it is needed most. We shall call this strategy *adaptive sampling*. The condition "where it is needed most" for generating new data can be interpreted in many ways. In the following, we simply sample points where $\left\|\nabla V^{NN}\right\|$ is large. In fact, regions of the phase space where of the value function has large gradients usually benefit from having more data to learn from. Moreover, this sampling scheme is easy to implement and efficient. Suppose we aim at generating $N_{r+1} = |\mathcal{D}_{r+1}| - |\mathcal{D}_r|$ new data points with locations denoted by

$$\left\{t^{(i)}, \boldsymbol{x}^{(i)}\right\}_{i=|\mathcal{D}_r|+1}^{|\mathcal{D}_{r+1}|}. \tag{208}$$

To choose where to put these points, we first randomly sample a large set of $N_c > N_{r+1}$ candidate initial locations from the phase space $\mathcal{X}$. Then a quick pass through the NN can give us the predicted gradient at each candidate point,

$$\left\{\nabla V^{NN}\left(0, \boldsymbol{x}^{(i)}\right)\right\}_{i=1}^{N_c}. \tag{209}$$

We then simply pick the N_{r+1} points with largest norm. For each of these points, we then simulate the system dynamics using the partially-trained NN as the closed-loop controller. In most all cases, this yields a solution which is fairly close to the optimal state $\boldsymbol{x}^*(t)$ and co-state $\boldsymbol{\lambda}^*(t; \boldsymbol{x})$. By using this solution as an initial guess for the BVP solver, we then quickly obtain a solution to the TPBVP (191-192) for the full time interval $[0, t_f]$ without the need for the time-marching trick proposed in [37, 38] (see Section 2.9.1).

This algorithm enables us to build up a rich and informative data set, from which we develop a high-fidelity model of $V(\cdot)$ at reduced computational cost. Moreover, this data set is not constrained to be within a small neighborhood of some nominal trajectory - it will contain points from the entire domain of $\mathcal{X}$, and we can adjust the process as described to generate more data near complicated features of the value function. As we progressively refine the NN model, we can enforce stricter convergence tolerances for the internal optimizer, as well as adjust hyperparameters like the gradient loss weight μ or the number of terms in the L-BFGS Hessian approximation. As the NN is already partially-trained, fewer iterations are needed for convergence.

2.9.4.1 Correction to derivation of convergence conditions We regretfully made some errors in the derivation of the convergence conditions and sample size selection scheme developed previously in Section 2.6.1. While in practice, the same results can be obtained with some different hyperparameter tunings, for future work it is necessary to build our algorithms on a solid mathematical foundation. We present a revised derivation here in the context of training PINNs for solving HJB equations.

To start, suppose that the internal optimizer (e.g. L-BFGS) converges in optimization round r. Let $\bar{\mathcal{D}}^r_{\text{train}}$ be the available training data set and $\boldsymbol{\theta}_r$ denote the NN parameters at the end of this round. Given convergence of the internal optimizer, the first order necessary condition for optimality holds, so

$$\left\| \nabla_{\boldsymbol{\theta}} \mathcal{L} \left(\boldsymbol{\theta}_r ; \bar{\mathcal{D}}^r_{\text{train}} \right) \right\| \ll 1. \tag{210}$$

Here $\mathcal{L}(\cdot)$ is the physics-informed loss defined in eq. (198), and $\nabla_{\boldsymbol{\theta}} \mathcal{L}(\cdot)$ is its gradient with respect to the NN parameters $\boldsymbol{\theta}$. For true first order optimality, we would like the gradient to be small when evaluated over the entire continuous domain of interest, $[t_0, t_f] \times \mathcal{X}$. In other words, we want

$$\left\| \nabla_{\boldsymbol{\theta}} \mathcal{L} \left(\boldsymbol{\theta}_r ; [t_0, t_f] \times \mathcal{X} \right) \right\| \ll 1, \tag{211}$$

where the Monte Carlo sums in eqs. (199-200) become integrals in the limit as the size of the data set approaches infinity.

The simplest way to see if (211) holds is to generate a validation data set $\bar{\mathcal{D}}_{\text{val}}$. Then using the fact that $\nabla_{\boldsymbol{\theta}} \mathcal{L} \left(\boldsymbol{\theta}_r ; \bar{\mathcal{D}}_{\text{val}} \right) \to \nabla_{\boldsymbol{\theta}} \mathcal{L} \left(\boldsymbol{\theta}_r ; [t_0, t_f] \times \mathcal{X} \right)$ in the limit as $\left| \bar{\mathcal{D}}_{\text{val}} \right| \to \infty$, one checks if, for example,

$$\left\| \nabla_{\boldsymbol{\theta}} \mathcal{L} \left(\boldsymbol{\theta}_r ; \bar{\mathcal{D}}_{\text{val}} \right) \right\| < \epsilon, \tag{212}$$

for some small parameter $\epsilon > 0$. Convergence tests like (212) are standard in machine learning, and are useful for measuring generalization accuracy. But for many practical problems including optimal control problems studied in this work, it may be too expensive to generate a large-enough validation data set. More importantly, such tests provides no clear guidance in selecting the sample size $\left| \bar{\mathcal{D}}^{r+1}_{\text{train}} \right|$ should they not be satisfied.

We do use validation tests to quantify model accuracy after training is complete. Indeed, the ability to empirically validate solutions is a key benefit of the causality-free approach. However, for the purpose of determining convergence between training rounds, we propose a different statistically motivated test which provides information on choosing $\left| \bar{\mathcal{D}}^{r+1}_{\text{train}} \right|$. The idea is simple: since we already assume (135) holds, then to ensure that (211) is also satisfied, it suffices to check that the population variance is relatively small.

To motivate this more rigorously, we observe that – by design – for any sample set $\bar{\mathcal{D}}$, the sample gradient $\nabla_{\boldsymbol{\theta}} \mathcal{L} \left(\boldsymbol{\theta}_r ; \bar{\mathcal{D}} \right)$ is an unbiased estimator for the true gradient, $\nabla_{\boldsymbol{\theta}} \mathcal{L} \left(\boldsymbol{\theta}_r ; [t_0, t_f] \times \mathcal{X} \right) = \mathbb{E} \left[\nabla_{\boldsymbol{\theta}} \mathcal{L} \left(\boldsymbol{\theta}_r ; (t, \boldsymbol{x}) \right) \right]$. That is,

$$\mathbb{E}_{\bar{\mathcal{D}}} \left[\nabla_{\boldsymbol{\theta}} \mathcal{L} \left(\boldsymbol{\theta}_r ; \bar{\mathcal{D}} \right) \right] = \nabla_{\boldsymbol{\theta}} \mathcal{L} \left(\boldsymbol{\theta}_r ; [t_0, t_f] \times \mathcal{X} \right), \tag{213}$$

where $\mathbb{E}_{\bar{\mathcal{D}}}[\cdot]$ denotes the population mean over all possible sample sets $\bar{\mathcal{D}}$ with fixed size $|\bar{\mathcal{D}}|$. Intuitively, this means that if (210) holds, then on average we also have (211), as desired. But we must control the mean square error (MSE) of the estimator, which is given by

$$\begin{aligned}\text{MSE}\left[\nabla_{\boldsymbol{\theta}}\mathcal{L}\left(\boldsymbol{\theta}_r;\bar{\mathcal{D}}\right)\right] :=& \mathbb{E}_{\bar{\mathcal{D}}}\left[\left\|\nabla_{\boldsymbol{\theta}}\mathcal{L}\left(\boldsymbol{\theta}_r;\bar{\mathcal{D}}\right) - \nabla_{\boldsymbol{\theta}}\mathcal{L}\left(\boldsymbol{\theta}_r;[t_0,t_f]\times\mathcal{X}\right)\right\|^2\right] \\ =& \mathbb{E}_{\bar{\mathcal{D}}}\left[\sum_{m=1}^{M}\left(\frac{\partial\mathcal{L}}{\partial\theta_m}(\boldsymbol{\theta}_r;\bar{\mathcal{D}}) - \frac{\partial\mathcal{L}}{\partial\theta_m}(\boldsymbol{\theta}_r;[t_0,t_f]\times\mathcal{X})\right)^2\right] \end{aligned} \tag{214}$$

Now by linearity of the expectation we have

$$\text{MSE}\left[\nabla_{\boldsymbol{\theta}}\mathcal{L}\left(\boldsymbol{\theta}_r;\bar{\mathcal{D}}\right)\right] = \sum_{m=1}^{M}\text{Var}\left[\frac{\partial\mathcal{L}}{\partial\theta_m}(\boldsymbol{\theta}_r;\bar{\mathcal{D}})\right] \tag{215}$$

Then by construction of the loss,

$$\text{MSE}\left[\nabla_{\boldsymbol{\theta}}\mathcal{L}\left(\boldsymbol{\theta}_r;\bar{\mathcal{D}}\right)\right] = \sum_{m=1}^{M}\text{Var}\left[\frac{1}{|\bar{\mathcal{D}}|}\sum_{i=1}^{|\bar{\mathcal{D}}|}\frac{\partial\mathcal{L}}{\partial\theta_m}\left(\boldsymbol{\theta}_r;\left(t^{(i)},\boldsymbol{x}^{(i)}\right)\right)\right] \tag{216}$$

Using the fact that the samples $\left(t^{(i)},\boldsymbol{x}^{(i)}\right)$ are i.i.d., we get

$$\begin{aligned}\text{MSE}\left[\nabla_{\boldsymbol{\theta}}\mathcal{L}\left(\boldsymbol{\theta}_r;\bar{\mathcal{D}}\right)\right] =& \frac{1}{|\bar{\mathcal{D}}|^2}\sum_{m=1}^{M}\sum_{i=1}^{|\bar{\mathcal{D}}|}\text{Var}\left[\frac{\partial\mathcal{L}}{\partial\theta_m}\left(\boldsymbol{\theta}_r;(t,\boldsymbol{x})\right)\right] \\ =& \frac{1}{|\bar{\mathcal{D}}|^2}\sum_{m=1}^{M}|\bar{\mathcal{D}}|\,\text{Var}\left[\frac{\partial\mathcal{L}}{\partial\theta_m}\left(\boldsymbol{\theta}_r;(t,\boldsymbol{x})\right)\right] \\ =& \frac{1}{|\bar{\mathcal{D}}|}\sum_{m=1}^{M}\text{Var}\left[\frac{\partial\mathcal{L}}{\partial\theta_m}\left(\boldsymbol{\theta}_r;(t,\boldsymbol{x})\right)\right], \end{aligned} \tag{217}$$

where M is the number of parameters $\boldsymbol{\theta}$. If the estimation error is small, then the sample mean is likely to be a good approximation of the true mean. Hence we expect that $\|\nabla_{\boldsymbol{\theta}}\mathcal{L}\left(\boldsymbol{\theta}_r;[t_0,t_f]\times\mathcal{X}\right)\|$ is also small, as desired. To this end, we require

$$\text{MSE}\left[\nabla_{\boldsymbol{\theta}}\mathcal{L}\left(\boldsymbol{\theta}_r;\bar{\mathcal{D}}\right)\right] \leqslant \epsilon\left\|\nabla_{\boldsymbol{\theta}}\mathcal{L}\left(\boldsymbol{\theta}_r;[t_0,t_f]\times\mathcal{X}\right)\right\|_1, \tag{218}$$

for some parameter $\epsilon > 0$.

In practice, evaluation of (218) is computationally intractable, but we can approximate the true population variance terms on the left hand side by the sample variance[6] taken over all data $\left(t^{(i)},\boldsymbol{x}^{(i)}\right) \in \bar{\mathcal{D}}^r_{\text{train}}$, which we denote by $\text{Var}_{\bar{\mathcal{D}}^r_{\text{train}}}[\cdot]$:

$$\text{MSE}\left[\nabla_{\boldsymbol{\theta}}\mathcal{L}\left(\boldsymbol{\theta}_r;\bar{\mathcal{D}}\right)\right] \approx \frac{1}{\left|\bar{\mathcal{D}}^r_{\text{train}}\right|}\sum_{m=1}^{M}\text{Var}_{\bar{\mathcal{D}}^r_{\text{train}}}\left[\frac{\partial\mathcal{L}}{\partial\theta_m}\left(\boldsymbol{\theta}_r;\left(t^{(i)},\boldsymbol{x}^{(i)}\right)\right)\right]. \tag{219}$$

Similarly, we approximate the true gradient on the right hand side by the sample gradient and arrive at the following practical convergence criterion:

$$\sum_{m=1}^{M}\text{Var}_{\bar{\mathcal{D}}^r_{\text{train}}}\left[\frac{\partial\mathcal{L}}{\partial\theta_m}\left(\boldsymbol{\theta}_r;\left(t^{(i)},\boldsymbol{x}^{(i)}\right)\right)\right] \leqslant \epsilon\left|\bar{\mathcal{D}}^r_{\text{train}}\right|\left\|\nabla_{\boldsymbol{\theta}}\mathcal{L}\left(\boldsymbol{\theta}_r;\bar{\mathcal{D}}^r_{\text{train}}\right)\right\|_1. \tag{220}$$

[6]In practice, computing a large number of individual gradients can be expensive, so we often evaluate the sample variance over a smaller subset of the training data.

If (220) is satisfied, then it is likely that $\|\nabla_{\boldsymbol{\theta}} \mathcal{L}\left(\boldsymbol{\theta}_r; [t_0, t_f] \times \mathcal{X}\right)\|$ is small. In other words, the solution $\boldsymbol{\theta}_r$ should satisfy the first order optimality conditions evaluated over the entire domain, so we can stop optimization. Satisfaction of (220) does not imply that the trained model is good – merely that seeing more data would probably not improve it significantly. On the other hand, when the criterion is not met, it still guides us in selecting the next sample size $\left|\bar{\mathcal{D}}^{r+1}_{\text{train}}\right|$. To see this, suppose that the sample variance doesn't change significantly by increasing the size of the data set, i.e.

$$\frac{\sum_{m=1}^{M} \operatorname{Var}_{\bar{\mathcal{D}}^{r+1}_{\text{train}}}\left[\frac{\partial \mathcal{L}}{\partial \theta_m}\left(\boldsymbol{\theta}_r; \left(t^{(i)}, \boldsymbol{x}^{(i)}\right)\right)\right]}{\left\|\nabla_{\boldsymbol{\theta}} \mathcal{L}\left(\boldsymbol{\theta}_r; \bar{\mathcal{D}}^{r+1}_{\text{train}}\right)\right\|_1} \approx \frac{\sum_{m=1}^{M} \operatorname{Var}_{\bar{\mathcal{D}}^{r}_{\text{train}}}\left[\frac{\partial \mathcal{L}}{\partial \theta_m}\left(\boldsymbol{\theta}_r; \left(t^{(i)}, \boldsymbol{x}^{(i)}\right)\right)\right]}{\left\|\nabla_{\boldsymbol{\theta}} \mathcal{L}\left(\boldsymbol{\theta}_r; \bar{\mathcal{D}}^{r}_{\text{train}}\right)\right\|_1}. \tag{221}$$

Then the appropriate choice of $\left|\bar{\mathcal{D}}^{r+1}_{\text{train}}\right|$ to satisfy (220) after the next round is such that

$$\left|\bar{\mathcal{D}}^{r+1}_{\text{train}}\right| \geqslant \frac{\sum_{m=1}^{M} \operatorname{Var}_{\bar{\mathcal{D}}^{r}_{\text{train}}}\left[\frac{\partial \mathcal{L}}{\partial \theta_m}\left(\boldsymbol{\theta}_r; \left(t^{(i)}, \boldsymbol{x}^{(i)}\right)\right)\right]}{\epsilon \left\|\nabla_{\boldsymbol{\theta}} \mathcal{L}\left(\boldsymbol{\theta}_r; \bar{\mathcal{D}}^{r}_{\text{train}}\right)\right\|_1}. \tag{222}$$

2.9.5 Application to rigid body satellite rotation

As a practical test of of NN-HJB solver, we consider the six-dimensional rigid body model of a satellite studied by Kang and Wilcox [37, 38]. Using the method of characteristics on sparse grids, they demonstrate the feasibility of interpolating the value function $V(t = 0, \boldsymbol{x})$ in six dimensions, and use this for model predictive feedback control (MPC) [21] of the nonlinear system. In [38], this is accomplished even for the under-actuated case. Hereafter, we provide a comparison of the results in [38] with the results of the proposed NN-HJB method. To this end, we first describe in more detail the feedback control system. Let $\{\boldsymbol{e}_1, \boldsymbol{e}_2, \boldsymbol{e}_3\}$ be an inertial frame of orthonormal vectors and $\{\boldsymbol{e}'_1, \boldsymbol{e}'_2, \boldsymbol{e}'_3\}$ be a body frame. The state of the satellite is then written as $\boldsymbol{x} = \begin{pmatrix}\boldsymbol{v} & \boldsymbol{\omega}\end{pmatrix}^T$. Here $\boldsymbol{v}$ represents the Euler angles,

$$\boldsymbol{v} = \begin{pmatrix}\phi & \theta & \psi\end{pmatrix}^T, \tag{223}$$

in which ϕ, θ, and ψ are the angles of rotation around $\boldsymbol{e}'_1$, $\boldsymbol{e}'_2$, and $\boldsymbol{e}'_3$, respectively, and $\boldsymbol{\omega}$ is the angular velocity relative to the body frame,

$$\boldsymbol{\omega} = \begin{pmatrix}\omega_1 & \omega_2 & \omega_3\end{pmatrix}^T. \tag{224}$$

The state dynamics are given by

$$\begin{pmatrix}\dot{\boldsymbol{v}} \\ \boldsymbol{J}\dot{\boldsymbol{\omega}}\end{pmatrix} = \begin{pmatrix}\boldsymbol{E}(\boldsymbol{v})\boldsymbol{\omega} \\ \boldsymbol{S}(\boldsymbol{\omega})\boldsymbol{R}(\boldsymbol{v})\boldsymbol{h} + \boldsymbol{B}\boldsymbol{u}\end{pmatrix}. \tag{225}$$

Here $\boldsymbol{E}(\boldsymbol{v}), \boldsymbol{S}(\boldsymbol{\omega}), \boldsymbol{R}(\boldsymbol{v}) : \mathbb{R}^3 \to \mathbb{R}^{3\times 3}$ are matrix-valued functions defined as

$$\boldsymbol{E}(\boldsymbol{v}) = \begin{pmatrix}1 & \sin\phi\tan\theta & \cos\phi\tan\theta \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi/\cos\theta & \cos\phi/\cos\theta\end{pmatrix}, \qquad \boldsymbol{S}(\boldsymbol{\omega}) = \begin{pmatrix}0 & \omega_3 & -\omega_2 \\ -\omega_3 & 0 & \omega_1 \\ \omega_2 & -\omega_1 & 0\end{pmatrix}, \tag{226}$$

and

$$\boldsymbol{R}(\boldsymbol{v}) = \begin{pmatrix}\cos\theta\cos\psi & \cos\theta\sin\psi & -\sin\theta \\ \sin\phi\sin\theta\cos\psi - \cos\phi\sin\psi & \sin\phi\sin\theta\sin\psi + \cos\phi\cos\psi & \cos\theta\sin\phi \\ \cos\phi\sin\theta\cos\psi + \sin\phi\sin\psi & \cos\phi\sin\theta\sin\psi - \sin\phi\cos\psi & \cos\theta\cos\phi\end{pmatrix}, \tag{227}$$

$\boldsymbol{J} \in \mathbb{R}^{3\times 3}$ is a combination of the inertia matrices of the momentum wheels and the rigid body without wheels, $\boldsymbol{h} \in \mathbb{R}^3$ is the total constant angular momentum of the system, and $\boldsymbol{B} \in \mathbb{R}^{3\times m}$ is a constant matrix where m is the number of momentum wheels. To control the system, we apply a torque in each of the momentum wheels. Such torque represents the feedback control of the system, which we denote as $\boldsymbol{u}(t, \boldsymbol{v}, \boldsymbol{\omega}) : [t_0, t_f] \times \mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{R}^m$. In particular, we consider the fully-actuated case with $m = 3$ momentum wheels. Following [38], we set

$$\boldsymbol{B} = \begin{pmatrix} 1 & \frac{1}{20} & \frac{1}{10} \\ \frac{1}{15} & 1 & \frac{1}{10} \\ \frac{1}{10} & \frac{1}{15} & 1 \end{pmatrix}, \qquad \boldsymbol{J} = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 4 \end{pmatrix}, \qquad \boldsymbol{h} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}. \tag{228}$$

The optimal control problem is

$$\begin{cases} \underset{\boldsymbol{u}(t,\boldsymbol{x})}{\text{minimize}} & \displaystyle\int_0^{20} L(\boldsymbol{v}, \boldsymbol{\omega}, \boldsymbol{u}) d\tau + \frac{1}{2}\|\boldsymbol{v}(t_f)\|^2 + \frac{1}{2}\|\boldsymbol{\omega}(t_f)\|^2 \\ \text{subject to} & \dot{\boldsymbol{v}} = \boldsymbol{E}(\boldsymbol{v})\boldsymbol{\omega} \\ & \boldsymbol{J}\dot{\boldsymbol{\omega}} = \boldsymbol{S}(\boldsymbol{\omega})\boldsymbol{R}(\boldsymbol{v})\boldsymbol{h} + \boldsymbol{B}\boldsymbol{u} \end{cases} \tag{229}$$

where

$$L(\boldsymbol{v}, \boldsymbol{\omega}, \boldsymbol{u}) = \frac{1}{2}\|\boldsymbol{v}\|^2 + \frac{10}{2}\|\boldsymbol{\omega}\|^2 + \frac{1}{4}\|\boldsymbol{u}\|^2. \tag{230}$$

We sample N_d initial values $\{\boldsymbol{x}^{(i)}\}_{i=1}^{N_d}$ from the domain

$$D_2 = \left\{ \boldsymbol{v}, \boldsymbol{\omega} \in \mathbb{R}^3 \middle| -\frac{\pi}{3} \leqslant \phi, \theta, \psi \leqslant \frac{\pi}{3}, \text{ and } -\frac{\pi}{4} \leqslant \omega_1, \omega_2, \omega_3 \leqslant \frac{\pi}{4} \right\}, \tag{231}$$

and solve the two-point BVP (191) at each initial value $\boldsymbol{x}^{(i)}$, $i = 1, \ldots, N_d$, using a four-stage Lobatto IIIA integration method [41]. It is important to note that in [38], this problem is solved at $t = 0$. Hence, we generate data only for $V(0, \boldsymbol{v}, \boldsymbol{\omega})$ and $\boldsymbol{\lambda}(0; \boldsymbol{v}, \boldsymbol{\omega})$. This means that the system is controlled for the whole time interval $t \in [0, 20]$ by $\boldsymbol{u}^{NN}(0, \boldsymbol{v}, \boldsymbol{\omega})$. Consequently, the control is implemented as model-predictive-control (MPC) rather than time-dependent optimal control. In other words, at each time t when the integrator needs to evaluate the control, instead of computing $\boldsymbol{u}^{NN}(t, \boldsymbol{v}, \boldsymbol{\omega})$, we assume $t = 0$ and return $\boldsymbol{u}^{NN}(0, \boldsymbol{v}, \boldsymbol{\omega})$. This is justifiable because the optimal control problem (229) is time-invariant.

2.9.5.1 Learning the value function Here we present numerical results of our implementation of a NN for learning the value function of the rigid body rotation problem (229). We experiment with changing the number of training data and the weight μ on the value gradient loss term (200), and compare the results to those obtained in [38]. Notably, we demonstrate that we can match the accuracy of the sparse grid characteristics method using a data set which is over **40 times smaller**, and at minimal additional computational cost.

To this end, we implemented a simple feed-forward NN in TensorFlow [1] and trained it to approximate $V(0, \boldsymbol{v}, \boldsymbol{\omega})$. Many different configurations of numbers of hidden layers and neurons work for this problem, but we found that three hidden layers with 64 neurons in each hidden layer produce reasonably accurate results. We optimize using the SciPy interface for the L-BFGS optimizer [33, 10]. For validation, we generate a data set containing $|\mathcal{D}_{\text{val}}^{\boldsymbol{\lambda}}| = 10,000$ data points, and keep this fixed throughout all the tests. As a baseline, the sparse grids characteristic method with $|G_{\text{sparse}}^{13}| = 44,698$ grid points achieves a relative mean absolute error (RMAE) of 7.3×10^{-4} on this data set. This error metric is defined as

$$\text{RMAE}(\boldsymbol{\theta}; \mathcal{D}) = \frac{1}{|\mathcal{D}|} \sum_{i=1}^{|\mathcal{D}|} \left| \frac{V^{NN}\left(t^{(i)}, \boldsymbol{x}^{(i)}\right) - V^{(i)}}{V^{(i)}} \right|. \tag{232}$$

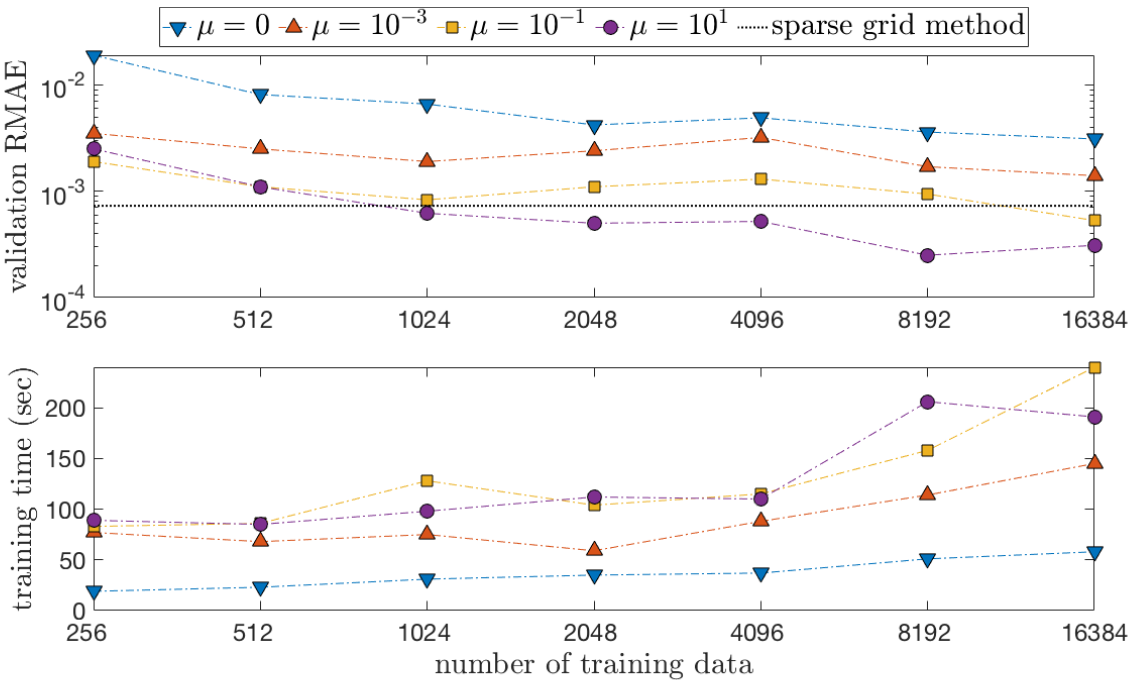


Figure 73: Accuracy measured as relative mean absolute error (RMAE) and computation time, depending on data set size and for a range of weights μ.

Figure 73 displays the results of a series of tests in which we fix the value gradient loss weight μ and vary the size of the training data set. In this experiment, we do not do adaptive sampling: training is conducted only for one round using the full data set provided, which is fixed for each different value of μ. However, we find that we need only a small amount of data to get reasonable results. We highlight that with just 1024 data points, we can match the accuracy of the sparse grid characteristics method which uses $\left|G^{13}_{\text{sparse}}\right| = 44,698$ points. With 8192 data points, the trained NN is three times as accurate as the sparse grid method. In addition, these NNs need only a couple minutes to train, so minimal computational effort beyond data generation is required. This level of accuracy for small data sets is only reached by properly tuning μ. In particular, we are unable reach a level of accuracy comparable to that of the sparse grid method by pure regression – i.e., by minimizing the loss (195). For this problem, we can improve accuracy by about an order of magnitude by choosing μ around 10^{-1} to 10^{1}. Finding a good choice of μ is problem-dependent, which generally dictates the ratio of sizes of the two loss terms (199) and (200), as well as the size of the data set and other hyperparameters. In addition, results can vary depending on randomly drawn data sets and NN parameter initializations. Even so, since training is fast, it is usually not difficult to find good hyperparameter settings and NN initializations.

2.9.5.2 Adaptive sampling results Performing a systematic study of the adaptive sampling and model refinement technique we proposed in Section 2.9.4 is rather difficult, since a successful implementation depends on various hyperparameter settings (which can change each optimization round) as well as random events, since data points are chosen partially in a random way and the randomly-initialized NN training problem is highly non-convex. For this reason, in this Section we report on a few conservative results which we feel can demonstrate the potential of the proposed method. Figure 74 displays the progress of the validation error during training when using adaptive sampling starting from a random initial data set with

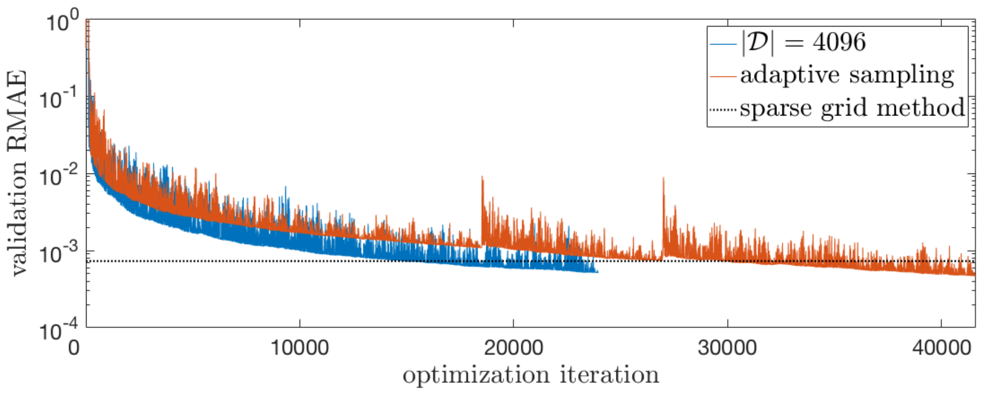


Figure 74: Progress of adaptive sampling and model refinement for training a NN to model the HJB value function, compared to training on a fixed data set $\mathcal{D}$ and the sparse grid characteristics method.

$|\mathcal{D}_1| = 512$ points. This is the same data set with 512 points used in Figure 73. We also keep the gradient loss weight constant and set to $\mu = 10^1$, set the convergence tolerance in (206) to $\epsilon = 10$ to compensate for the large gradients, and we use $|\mathcal{D}_2| = 1024$ and $|\mathcal{D}_3| = 4096$ data points in the following rounds. The data set sizes are selected to be the smallest power of 2 which is larger than the number of points recommended by Eq. (207). Results are compared to the progress when training on a fixed data set with $|\mathcal{D}| = 4096$ points (the same as used in Figure 73). To fully realize the potential of the method, one needs to adjust hyperparameters like μ, ϵ, and internal optimizer parameters in each round. How to do this in an effective and efficient way remains a topic for future research. Still, even with a naïve implementation, the final accuracy is just as good as that obtained when training on a fixed data set with $|\mathcal{D}| = 4096$, even though training started with only $|\mathcal{D}_1| = 64$ points.

Next, we study convergence of the BVP solver. First, we use the time-marching trick discussed in Section 2.9.1. Second, instead of time-marching we take initial guesses generated by simulating the closed-loop dynamics with a NN controller and solve the full TPBVP (191). Results are given in Tables 35 and 36. For the tests, we use 1,000 initial conditions with the largest predicted gradient norm, $\left\|V_{\boldsymbol{x}}^{NN}\right\|$, picked from a set of 100,000 randomly sampled candidate points. The set of initial conditions is fixed for all tests. When we attempt to solve the TPBVP with no time-marching (the line $k = 1$ in Table 35), the proportion of convergent solutions is extremely small. This obviates the need for good initial guesses. As shown in Table 35, we reliably obtain solutions for this problem when we use at least $k = 4$ time intervals. We note that the initial conditions are purposefully chosen to be difficult – if we simply take random samples from the domain D_2, the proportion of convergent solutions increases considerably. We see from the results presented in Table 36, using initial guesses generated by even a poorly-trained NN (18% validation RMAE), the chance of convergence is just as high as when using $k = 2$ time intervals for time-marching. Moreover, we find that by training even slightly more accurate NNs we easily get over 95% convergence. We also see that these solutions take less time than equivalently reliable solutions obtained with time-marching. While the difference is insignificant for individual data points, when building large data sets this can result in significant time savings. Together, these results suggest that the proposed adaptive sampling and model refinement framework has the potential to be useful in solving other, more difficult problems.

2.9.5.3 Closed-loop feedback control performance Lastly, we show that the trained NN is capable of stabilizing the nonlinear system dynamics. The control is calculated using Eq. (205) to get

$$\boldsymbol{u}^{NN}(t, \boldsymbol{v}, \boldsymbol{\omega}) = -2\left(\boldsymbol{J}^{-1}\boldsymbol{B}\right)^T V_{\boldsymbol{\omega}}^{NN}(t, \boldsymbol{v}, \boldsymbol{\omega}). \tag{233}$$

Table 35: Convergence of the BVP solution when using the time-marching trick, depending on the number of time steps.

number of time steps	% BVP convergence	mean solution time
1	0.8%	0.70 s
2	45.4%	0.64 s
4	93.0%	0.60 s
8	97.6%	0.72 s

Table 36: Convergence of the BVP solution when using an initial guess generated by NNs.

$\lvert\mathcal{D}\rvert$	μ	validation RMAE	% BVP convergence	mean solution time
64	0	1.8×10^{-1}	54.7%	0.61 s
64	10^{-4}	1.8×10^{-2}	95.4%	0.57 s
64	10^{1}	1.2×10^{-2}	97.9%	0.55 s
512	0	8.1×10^{-3}	98.8%	0.55 s
512	10^{-4}	5.1×10^{-3}	98.6%	0.54 s
512	10^{1}	1.1×10^{-3}	98.9%	0.55 s

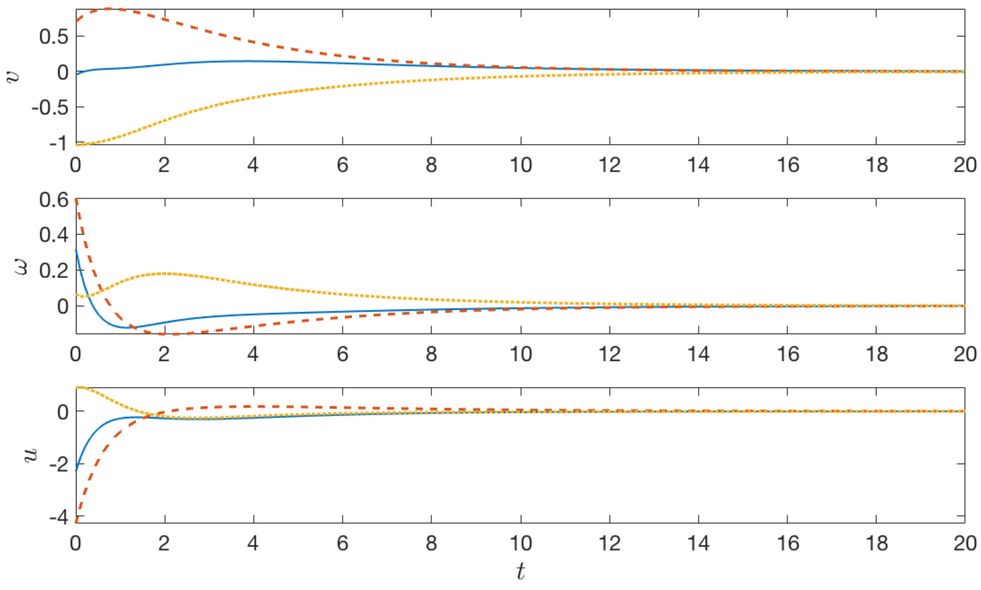


Figure 75: Typical closed-loop trajectory of the fully-actuated rigid body satellite system, controlled with model predictive feedback control generated by a neural network. Solid blue: ϕ, ω_1, and u_1. Dashed orange: θ, ω_2, and u_2. Dotted yellow: ψ, ω_3, and u_3.

Since $\boldsymbol{J}$ and $\boldsymbol{B}$ are constant matrices, we pre-compute the product $-2\left(\boldsymbol{J}^{-1}\boldsymbol{B}\right)^T$. Hence evaluation of the control requires only a forward pass through the computational graph of the gradient $V_{\boldsymbol{\omega}}^{NN}$ and a matrix multiplication. Recall that we are implementing MPC, so the control is actually computed as $\boldsymbol{u} = \boldsymbol{u}^{NN}(0, \boldsymbol{v}, \boldsymbol{\omega})$ for all times $t \in [0, 20]$. In Figure 75, we plot a typical closed-loop trajectory from a randomly sampled initial condition, clearly showing asymptotic stabilization. We also note that short computation time is critical for implementation in real systems, and this is achieved here as each evaluation of the control takes only $O(10^{-7})$ seconds.

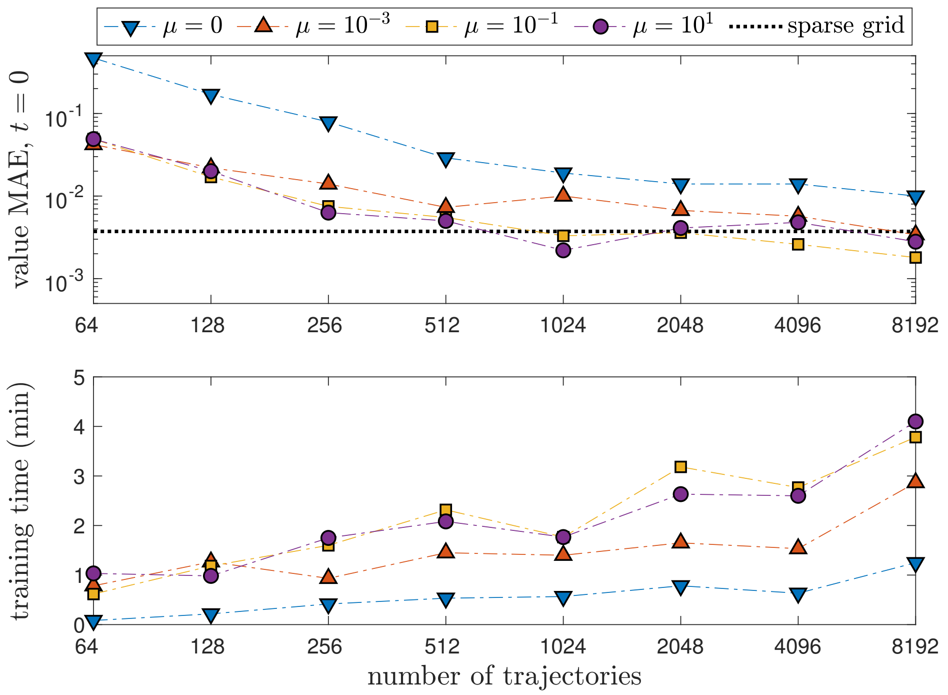


Figure 76: Validation accuracy and training time of NNs for modeling initial time value function $V(0, \boldsymbol{v}, \boldsymbol{\omega})$ of the rigid body optimal attitude control problem (229).

2.9.6 Learning time-dependent value functions

In this section, we compare the performance of NNs which model the initial time value function, $V(0, \boldsymbol{v}, \boldsymbol{\omega})$, and the full time-dependent value function, $V(t, \boldsymbol{v}, \boldsymbol{\omega})$, of the rigid body rotation problem (229). In Section 2.9.5, we previously concentrated on learning the initial time value function to facilitate comparison with the sparse grid characteristics method [37, 38]. We now demonstrate that learning the value function with time dependence presents no difficulties for our method, and in fact, can be more data-efficient in the sense that fewer BVPs need to be solved.

For comparison, we construct a validation data set with $\left|\bar{\mathcal{D}}_{\text{val}}\right| = 1,000$ data points (at $t = 0$), and keep this fixed throughout all the tests. As a baseline, the sparse grid characteristic method with $\left|G^{13}_{\text{sparse}}\right| = 44,698$ grid points achieves a mean absolute error (MAE) of 3.7×10^{-3} on this data set. The MAE is defined as

$$\text{MAE}(\boldsymbol{\theta}; \mathcal{D}) = \frac{1}{|\mathcal{D}|} \sum_{i=1}^{|\mathcal{D}|} \left| V^{NN}\left(t^{(i)}, \boldsymbol{x}^{(i)}\right) - V^{(i)} \right|. \tag{234}$$

We use this error metric instead of the RMAE defined previously because the magnitude of the value function can be very small for larger t, and even if the NN is quite accurate and good for control purposes, the RMAE can still be shockingly large. Thus the MAE provides a better error metric in this case.

First we train NNs to learn the initial time value function only, as we did in Section 2.9.5. Thus while each integrated BVP yields a time series with about 100 data points, we use only the initial point of each. Thus the number of integrated trajectories is equal to the size of the data set. Figure 76 displays the numerical with the new validation data set and error metric. We recover similar results: best results are obtained for μ between 10^{-1} and 10^{1}; with 1024 trajectories we match the accuracy of the sparse grid method; and with 8192 points the HJB-NN method is twice as accurate.

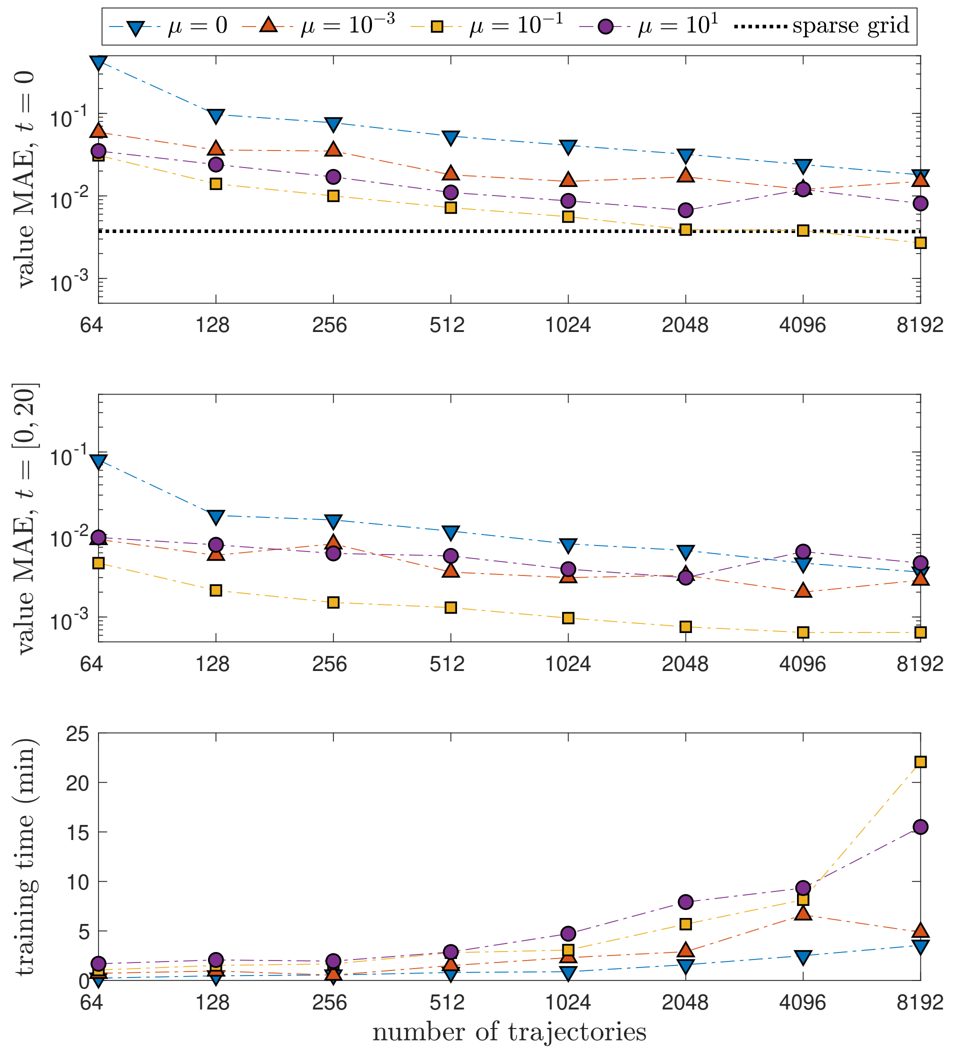


Figure 77: Validation accuracy and training time of NNs for modeling the time-dependent value function $V(t, \boldsymbol{v}, \boldsymbol{\omega})$ of (229).

2.9.6.1 Time-dependent results We now train NNs to model the value function of (229) over the whole time interval $t \in [0, 20]$. These NNs take t, $\boldsymbol{v}$, and $\boldsymbol{\omega}$ as inputs, and output $V(t, \boldsymbol{v}, \boldsymbol{\omega})$. A surprising outcome of our experiments is that including time *improves* data-efficiency. That is, fewer trajectories need to be integrated to train NNs with performance comparable to NNs approximating only $V(0, \boldsymbol{v}, \boldsymbol{\omega})$.

For training data, we keep the entire time series generated by the BVP solver. As each time series consists of about 100 points on average, the resulting data sets can be rather large. Thus we train on randomly selected subsets of the data with 16 times as many points as the number of trajectories integrated, corresponding to about 20% of the available data. Preliminary tests indicated that training on subsets of this size yields results which are comparable to those obtained when training on the full data set, but at reduced computational cost. For validation, we use the set of full trajectories corresponding to the validation data generated for Figure 76. The NN architecture is identical to that used before, except with an additional input dimension for time. Although more specialized architectures like recurrent NNs have been developed for modeling time series, we find that standard feedforward NNs do well for this task and are able to make fast predictions, which is essential for feedback control.

In Figure 77, we plot experimental results for learning the value function with time dependence. Training time for these NNs is generally longer than those in Figure 76 because the data sets are much larger, but the difference is not too extreme. The MAE for predicting the value function for validation data is given over all $t \in [0, 20]$ and at $t = 0$ for comparison with Figure 76. Again with the help of physics-informed learning, we obtain NN models which are accurate over the whole time interval trained. This time, best results are obtained with $\mu = 10^{-1}$. Interestingly, we find that the accuracy in predicting $V(0, \boldsymbol{v}, \boldsymbol{\omega})$ is comparable to that of a NN trained specifically for this purpose. This equivalence is especially true for the smallest data sets tested, which suggests that when little data is available, it can be more efficient to train a NN to learn the full time-dependent value function.

Remark 2.9 *A NN which models the time-dependent value function can be used either for optimal control or MPC: at each time t when the control needs to be computed, such a NN can predict either $\boldsymbol{u}^{NN}(t, \boldsymbol{x})$ for optimal control, or assume $t = 0$ and return $\boldsymbol{u}^{NN}(0, \boldsymbol{x})$ for MPC instead. Thus there is no disadvantage to training a NN to learn the time-dependent value function. Furthermore, if a higher-fidelity model for MPC is desired, one can simply use the time-dependent NN to gather enough data to train a NN to approximate $V(0, \boldsymbol{x})$ only.*

2.9.7 Robust control of a nonlinear advection-reaction-diffusion PDE

In this section we study both open-loop control and Deep Neural Net based feedback of the nonlinear PDE, under uncertainty. The problem is identical to the system developed in section 2.4 but with uncertainty in the initial condition. The problem is restated briefly here in a concise form.

$$\begin{cases} \dfrac{\partial \psi}{\partial t} = \psi \dfrac{\partial \psi}{\partial x} + \dfrac{1}{5}\dfrac{\partial^2 \psi}{\partial x^2} + \dfrac{3}{2}\psi e^{-\psi/10} + I_\Omega(x)u(t), \\ \psi(t, -1) = \psi(t, 1) = 0, \\ \psi(0, x) = \psi_0(x), \end{cases} \tag{235}$$

where $\psi(x, t) : [-1, 1] \times [0, t_f] \mapsto \mathbb{R}$ is a random field that is a functional of ψ_0 (random initial condition) and $u(t) : [0, t_f] \to \mathbb{R}$ (control). Note that $u(t)$ is actuated only on the spatial domain $\Omega = [-0.5, -0.2]$, which is the support of the indicator function $I_\Omega(x)$. We aim at determining the optimal control $u(t)$ that minimizes the cost functional

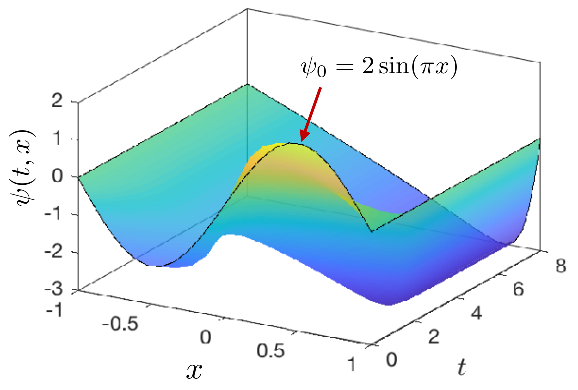
$$J([\psi], [u]) = \frac{1}{2}\mathbb{E}\left\{\int_{-1}^{1} \psi(t_f, x)^2 dx + 2\int_0^{t_f}\int_{-1}^{1} \psi(\tau, x)^2 dx d\tau\right\} + \frac{1}{20}\int_0^{t_f} u(\tau)^2 d\tau, \qquad t_f = 8, \tag{236}$$

where $\mathbb{E}\{\cdot\}$ is an expectation over the probability distribution of the initial state $\psi_0(x)$. To this end, we first discretize (235) in space using a Chebyshev pseudo-spectral method [63]. This yields the semi-discrete form

$$\begin{cases} \dot{\boldsymbol{\psi}} = \boldsymbol{\psi} \circ \boldsymbol{D}\boldsymbol{\psi} + \dfrac{1}{5}\boldsymbol{D}^2\boldsymbol{\psi} + \dfrac{3}{2}\boldsymbol{\psi} \circ e^{-\boldsymbol{\psi}/10} + \boldsymbol{I}_\Omega u(t), \\ \boldsymbol{\psi}(0) = \boldsymbol{\psi}_0, \end{cases} \tag{237}$$

where $\boldsymbol{\psi}(t) = [\psi(t, x_1), ..., \psi(t, x_n)]^T$ collects the values of the solution $\psi(t, x)$ at the (inner) Chebyshev nodes

$$x_j = \cos\left(\frac{j\pi}{n+1}\right) \qquad j = 1, ..., n. \tag{238}$$



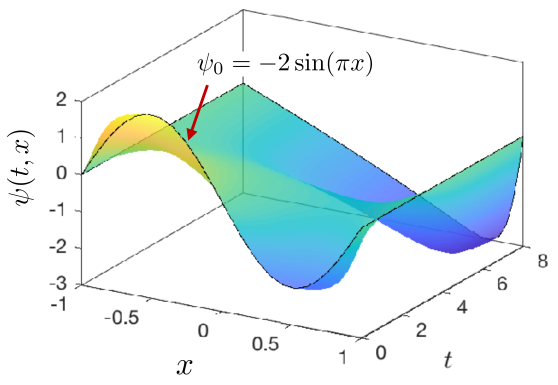
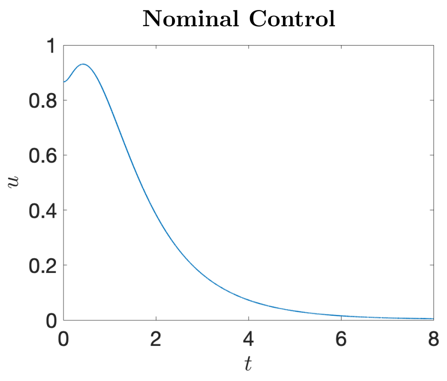


Figure 78: Solution samples of the initial-boundary value problem (235) corresponding to different initial conditions, with control $u(t) = 0$ (uncontrolled dynamics).



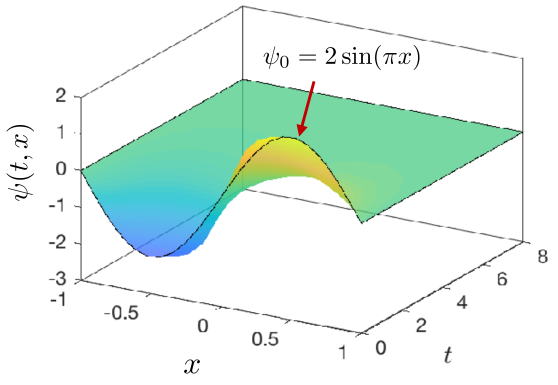


Figure 79: Open loop control of the nonlinear PDE (235). Shown is the nominal control minimizing (236) for the deterministic initial condition $\psi_0 = 2\sin(\pi x)$ and the corresponding solution dynamics. It is seen that the control $u(t)$ sends $\psi(t,x)$ to zero after a small transient.

In equation (237), the circle "$\circ$" denotes element-wise multiplication (Hadamard product), $\boldsymbol{I}_\Omega$ is the discrete indicator function, while $\boldsymbol{D}$ and $\boldsymbol{D}^2 \in \mathbb{R}^{n\times n}$ are, respectively, the first- and second-order Chebyshev differentiation matrices. These matrices obtained by deleting the first and last rows and columns of the full differentiation matrices. In Figure 78 we plot two solution samples of the initial-boundary value problem (235) we obtained by integrating the discretized system (237) in time. The cost functional (236) can be discretized as

$$J = \frac{1}{2M}\sum_{i=1}^{M} \boldsymbol{w}^T \left[\boldsymbol{\psi}^{(i)}(t_f)^2 + 2\int_0^{t_f} \boldsymbol{\psi}^{(i)}(\tau)^2 d\tau\right] + \frac{1}{20}\int_0^{t_f} u(\tau)^2 d\tau, \tag{239}$$

where $\boldsymbol{w}$ are Clenshaw-Curtis quadrature weights (column vector), and M is the total number of samples.

To quantify the effectiveness of the ensemble optimal control algorithm we propose, we first determine the nominal control corresponding to the deterministic initial condition $\psi_0(x) = 2\sin(\pi x)$. Such control minimizes the functional (239) with $M = 1$ and $\psi_j^{(1)}(0) = 2\sin(\pi x_j)$, and sends $\psi(t,x)$ to zero after a small transient (see Figure 79).

Next, we introduce uncertainty in the initial condition. Specifically, we set

$$\boldsymbol{\psi}_{0j} = 2\sin(\pi x_j) + \epsilon_j \qquad \epsilon_j \sim \mathcal{N}(0, 001). \tag{240}$$

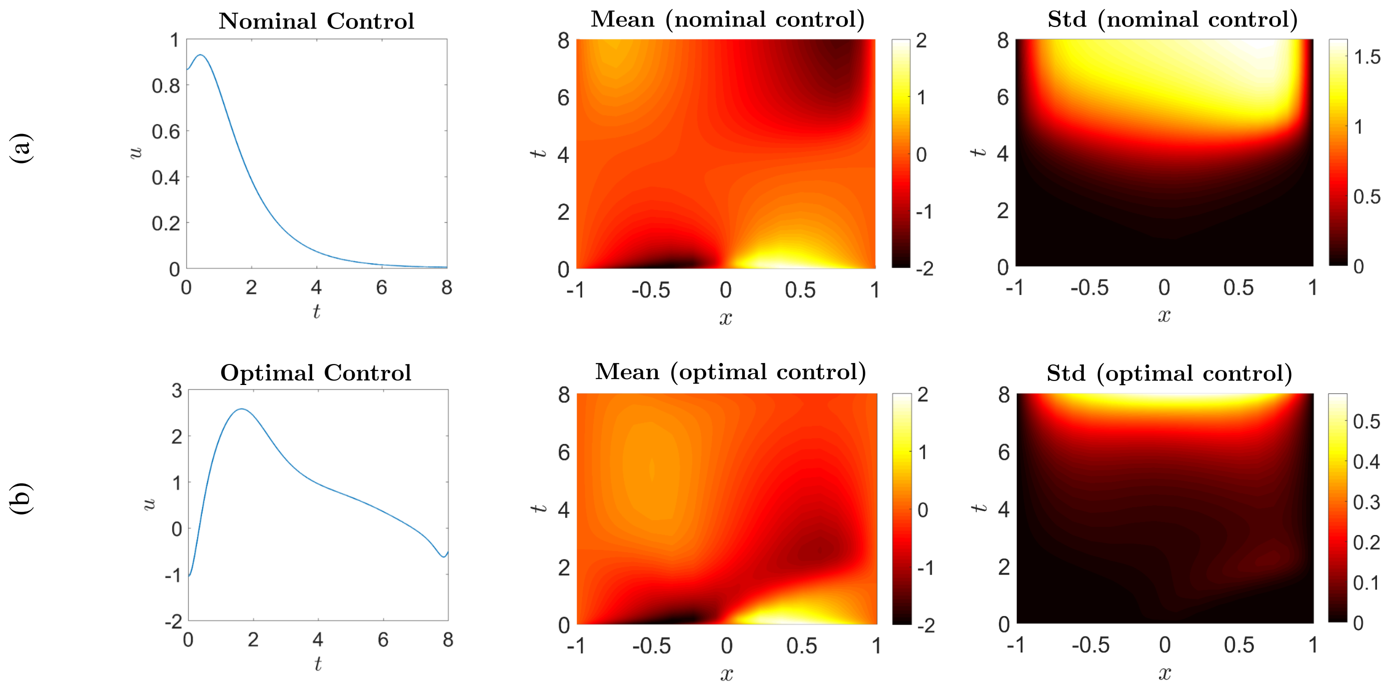


Figure 80: Comparison between the mean and the standard deviation of the stochastic solution to the PDE (235) under (a) nominal control and (b) optimal control.

This makes the solution to (235) and (237) stochastic. As seen in Figure 80 the introduction of this small uncertainty, causes large perturbations to the system solution under nominal control especially at t=8. To compute the optimal control, we minimized the functional (239) subject to $M = 1000$ replicas of the dynamical system (237). Specifically, we considered the single-shooting setting described in section 2.4.1 with $\Delta t = 0.0005$, Euler time integrator, and CSE gradient algorithm. In Figure 80 we compare the mean and the standard deviation of the solution we obtain by using nominal and optimal controls. It is that the optimal control is more effective is driving the solution ensemble to zero. In fact, both the mean and standard deviation of the optimally controlled dynamics are closer to zero. Note also that the optimal control differs substantially from the nominal control.

2.9.8 Closed loop control

We next consider a Deep Neural Net feedback scenario for the closed loop control problem (237)-(239), where the control $u(t)$ depends also on the state $\boldsymbol{\psi}$.

2.9.8.1 Learning high dimensional value functions

Using the proposed adaptive deep learning framework, we approximate solutions to (237)-(239) in $n = 10$, 20, and 30 dimensions. We focus on demonstrating what is possible using our approach, rather than carrying out a detailed study of its effectiveness under different parameter tunings. Indeed, in [35] the infinite-horizon version of the problem is solved only up to twelve dimensions, and the accuracy of the solution is not readily verifiable. The ability to conveniently measure model accuracy for general high dimensional problems with *no known analytical solution* is a key advantage of our framework. For each discretized optimal control problem, $n = 10$, 20, and 30, we ap-

Table 37: Validation accuracy of NNs trained to approximate solutions to the HJB equation associated with the collocated Burgers'-type optimal control problem (237)-(239), depending on the state dimension n.

n	num. trajectories	training time	value accuracy	co-state accuracy
10	132	10.1 min	2.4×10^{-3}	1.8×10^{-2}
20	60	9.2 min	8.9×10^{-4}	2.1×10^{-2}
30	59	13.3 min	5.0×10^{-4}	2.0×10^{-2}

Table 38: Convergence of BVP solutions for (237)-(239) when using the time-marching trick, depending on the problem dimension, n, and the number of time steps k.

n	k	% BVP convergence	mean integration time
10	6	68%	1.0 s
	8	90%	1.1 s
20	8	3%	5.9 s
	10	99%	8.5 s
30	10	49%	33.4 s
	12	100%	52.4 s

ply the time-marching strategy to build an initial training data set $\bar{\mathcal{D}}^1_{\text{train}}$ from 30 uniformly sampled initial conditions, $\boldsymbol{\psi}_0^{(i)}$ for $i = 1, \ldots, 30$. For each initial condition $\boldsymbol{\psi}_0^{(i)}$, the BVP solver outputs an optimal trajectory $\boldsymbol{\psi}^{(i)}\left(t^{(k)}\right)$ and associated co-state $\boldsymbol{\lambda}^{(i)}\left(t^{(k)}\right)$, where $t^{(k)} \in [0, t_f]$ are collocation points chosen by the solver. Typically this can be a few hundred or even thousand points per initial condition, depending on the state dimension n and the BVP solver tolerances. As such, we train the NN on a subset of the available data, randomly selected before each round r. We manually tune the hyper-parameter schedules, e.g. $\mu = \mu(r)$, but we omit these minor implementation details to focus on the outcomes. In Table 37, we present validation accuracy results for the trained NNs. We include the MAE in predicting the value function and the mean relative L^2 error in predicting the costate, $\boldsymbol{\lambda}(t;\boldsymbol{\psi}) \approx V^{NN}_{\boldsymbol{\psi}}(t, \boldsymbol{\psi})$. Accuracy is measured empirically on independently generated validation data sets comprised of trajectories from 50 randomly selected initial conditions. We find that the trained NNs have good accuracy in both value and co-state prediction, even in 30 dimensions. Table 37 also shows the total number of sample trajectories seen by the NN, including the initial data $\bar{\mathcal{D}}^1_{\text{train}}$. It may seem surprising that the number of sample trajectories decreases with the dimension n. This happens because the BVP solver usually needs more collocation points for higher dimensional problems, thus producing more data per trajectory. Consequently, fewer trajectories need to be integrated to fulfill the data set size recommendation (145). Similarly, in Section 2.9.5 we use data only for $t = 0$, so we need thousands of trajectories to fill in the state space and train the NN. This suggests that learning the time dependent value function can be more efficient than a MPC implementation. Lastly, Table 37 shows the training time for each NN, including time spent generating additional trajectories on the fly but not time spent generating the initial data. Generating the initial data set quickly becomes the most expensive computation as n increases, but once some data is available, we find that computational effort scales reasonably with the problem dimension. This demonstrates the viability of the proposed method for solving high dimensional optimal control problems.

2.9.8.2 Adaptive sampling and fast BVP solutions In our experience, generating the initial training data set can be the most computationally expensive part of the process, especially as the problem dimension

Table 39: Convergence of BVP solutions for (237)-(239) when using initial guesses generated by NNs with varying costate prediction accuracy (measured as mean relative L^2 error on validation data).

n	μ	co-state accuracy	% BVP convergence	mean integration time
10	10^{-2}	1.4×10^{-1}	89%	0.6 s
	10^{2}	4.3×10^{-2}	92%	0.5 s
20	10^{-2}	3.1×10^{-1}	98%	2.4 s
	10^{2}	4.0×10^{-2}	100%	2.3 s
30	10^{-2}	4.2×10^{-1}	99%	6.6 s
	10^{2}	3.5×10^{-2}	100%	6.7 s

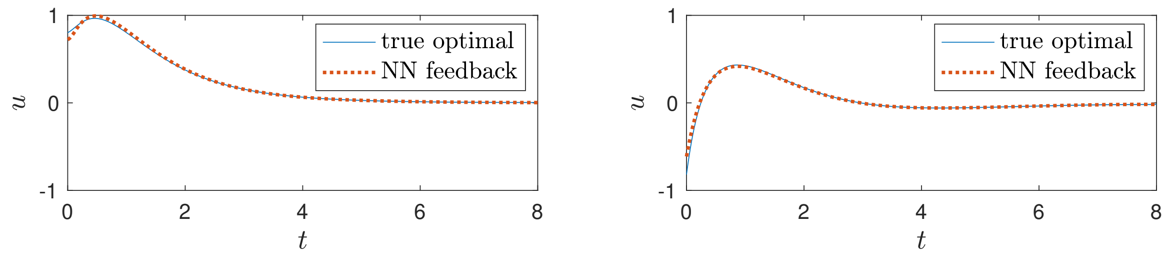


Figure 81: Comparison of true optimal control (open-loop BVP solution) and NN control profiles for two different initial conditions: $\psi(0,x) = 2\sin(\pi\xi)$ (left) and $\psi(0,x) = -2\sin(\pi\xi)$ (right).

n increases. Consequently, for difficult high dimensional problems it may be infeasible to generate a large-enough data set from scratch. This obstacle can be largely overcome by using partially-trained/low-fidelity NNs to aid in further data generation. In this section, we briefly compare the consistency and speed of BVP convergence between our two strategies: time-marching and NN warm start. These experiments demonstrate the importance of NN guesses for high dimensional data generation. For each of n = 10, 20, and 30, we randomly sample a set of 1,000 candidate points from the domain $[-2, 2]^n$, from which we pick 100 points with the largest predicted value gradient. The set of initial conditions is fixed for each n. Next we proceed as in Section 2.9.5, solving each BVP by time-marching with various k. Results are summarized in Table 38. We then solve the same BVPs directly over the whole time interval $t \in [0, 8]$ with NN warm start. These NNs are trained only for a single round on fixed data sets, but with different gradient weights, μ, and thus have varying costate prediction accuracy. Results are given in Table 39. As before, we find that even NNs with relatively large costate prediction error enable consistently convergent BVP solutions. Time-marching also works once the sequence of time steps t_k is properly tuned, but the speed of this method scales poorly with n. Now the advantage of utilizing NNs to aid in data generation becomes clear: the average time needed for convergence when using the NN approach is drastically lower than that of the time-marching trick. Because low-fidelity NNs are quick to train, training such a NN and then using it to aid in data generation is the most efficient strategy for building larger data sets.

2.9.8.3 Closed-loop performance Now we show that the feedback control output by the trained NN not only stabilizes the high dimensional system, but that it is close to the true optimal control. The optimal feedback control law can again be calculated with (205), from which we obtain

$$u(t, \boldsymbol{\psi}) = -\frac{1}{2}[\boldsymbol{I}_\omega]^T V_{\boldsymbol{\psi}}(t, \boldsymbol{\psi}). \tag{241}$$

In Figure 78 we plot the uncontrolled dynamics and closed-loop controlled dynamics 79 starting from the conditions, $\psi(0,x) = 2\sin(\pi\xi)$ where the dimension of the discretized system is $n = 30$. For both of these initial conditions (and almost all others tested), the NN controller successfully stabilizes the open-loop unstable origin. Figure 81 compares the NN-generated controls with the true optimal controls, calculated by solving the associated BVP problems. From this we can see that the NN controls are very close to optimal. Finally, the speed of online control computation is not sensitive to the problem dimension: each evaluation still takes just $O(10^{-3})$ seconds on both an NVIDIA RTX 2080Ti GPU and a 2012 MacBook Pro.

2.9.8.4 Discussion Solving HJB equations in six dimensional domains is a very difficult problem. Our method makes no compromises in the simplicity of the model (in particular we use the full nonlinear dynamics), and is able to efficiently generate solutions to a good degree of accuracy. Thus, we believe that it has the potential to be useful in tackling high-dimensional problems with more complicated value functions. We expect that adaptive sampling along with strategies for adjusting hyperparameters will be key tools in addressing such problems, and we hope to address this in future work. Dealing with bound-constrained controls and free final time problems will also require further developments of the framework, leaving a wide range of research topics to explore. This approach to solving HJB equations can also easily be combined with an unscented Kalman filter (UKF, see [34, 68]) for control of systems with parameter uncertainty, process noise, and measurement noise. In particular, in combination with a reduced order UKF [49] or other reduced-order estimators, our method may be especially effective in the case where noise is negligible and there is only parameter uncertainty. This provides solutions which converge to the optimal solution for any initial distribution. Thus we can solve all the kinds of problems solved previously with open-loop control but with feedback control. It is important to note that the NN is trained offline, but this feedback control is evaluated online in real-time and is valid for any initial distribution which is similar enough to that which the NN was trained on, and *any* initial condition in the training domain.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] P-A Absil, Robert Mahony, and Rodolphe Sepulchre. *Optimization algorithms on matrix manifolds*. Princeton University Press, 2009.

[3] E.G. Al'brekht. On the optimal stabilization of nonlinear systems. *Journal of Applied Mathematics and Mechanics*, 25(5):1254–1266, 1961.

[4] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind. Automatic differentiation in machine learning: a survey. *Journal of Marchine Learning Research*, 18:1–43, 2018.

[5] Albert S. Berahas, Jorge Nocedal, and Martin Takac. A multi-batch L-BFGS method for machine learning. In *Advances in Neural Information Processing Systems 29*, pages 1055–1063, 2016.

[6] Lorenz T Biegler and Victor M Zavala. Large-scale nonlinear programming using ipopt: An integrating framework for enterprise-wide dynamic optimization. *Computers & Chemical Engineering*, 33(3):575–582, 2009.

[7] Raghu Bollapragada, Dheevatsa Mudigere, Jorge Nocedal, Hao-Jun Michael Shi, and Ping Tak Peter Tang. A progressive batching L-BFGS method for machine learning. *arXiv preprint: arXiv:1802.05374*, 2018.

[8] Léon Bottou, Frank E. Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. *SIAM Review*, 60(2):223–311, 2018.

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

[10] Richard H. Byrd, Peihang Lu, Jorge Nocedal, and Ciyou Zhu. A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific Computing*, 16:1190–1208, September 1995.

[11] Simone Cacace, Emiliano Cristiani, Maurizio Falcone, and Athena Picarelli. A patchy dynamic programming scheme for a class of Hamilton–Jacobi–Bellman equations. *SIAM Journal on Scientific Computing*, 34(5):A2625–A2649, 2012.

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

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

[14] Yat Tin Chow, Jérôme Darbon, Stanley Osher, and Wotao Yin. Algorithm for overcoming the curse of dimensionality for state-dependent Hamilton-Jacobi equations. *J. Comput. Phys.*, 387:376–409, 2019.

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

[16] Curt Da Silva and Felix J Herrmann. Optimization on the hierarchical tucker manifold–applications to tensor completion. *Linear Algebra and its Applications*, 481:131–173, 2015.

[17] Jérôme Darbon and Stanley Osher. Algorithms for overcoming the curse of dimensionality for certain Hamilton-Jacobi equations arising in control theory and elsewhere. *arXiv preprint: arXiv:1605.01799*, 2016.

[18] D.R. Durran. The third-order Adams-Bashforth method: An attractive alternative to leapfrog time differencing. *Monthly Weather Review*, 119:702–720, 1990.

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

[20] M. Falcone and R. Ferretti. *Semi-Lagrangian Approximation Schemes for Linear and Hamilton-Jacobi Equations*. Society for Industrial and Applied Mathematics, 2013.

[21] C. E. Garcia, D. M. Prett, and M. Morari. Model predictive control: theory and practice – a survey. *Automatica*, 25(3):335–348, 1989.

[22] Philip E Gill, Walter Murray, and Michael A Saunders. Snopt: An sqp algorithm for large-scale constrained optimization. *SIAM review*, 47(1):99–131, 2005.

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

[24] L. Grasedyck and C. Löbbert. Distributed hierarchical svd in the hierarchical tucker format. *Numerical Linear Algebra with Applications*, page e2174, 2018.

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

[26] Richard Haberman. *Applied partial differential equations with Fourier series and boundary value problems*. Pearson Higher Ed., 2012.

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

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

[29] J. Han, A. Jentzen, and W. E. Solving high-dimensional partial differential equations using deep learning. *arXiv.org, arXiv:1707.02568*, July 2017.

[30] Gennadij Heidel and Volker Schulz. A riemannian trust-region method for low-rank tensor completion. *Numerical Linear Algebra with Applications*, 25(6):e2175, 2018.

[31] Yegorov I. and P.M. Dower. Perspectives on characteristics based curse-of-dimensionality-free numerical approaches for solving Hamilton-Jacobi equations. *Appl. Math. Optim.*, 2018.

[32] Frank Jiang, Glen Chou, Mo Chen, and Claire J. Tomlin. Using neural networks to compute approximate and guaranteed feasible Hamilton-Jacobi-Bellman PDE solutions. *arXiv preprint: arXiv:1611.03158*, 2016.

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

[34] Simon J. Julier and Jeffrey Uhlmann. A new extension of the Kalman filter to nonlinear systems. In *Defense, Security, and Sensing*, 1997.

[35] Dante Kalise and Karl Kunisch. Polynomial approximation of high-dimensional Hamilton-Jacobi-Bellman equations and applications to feedback control of semilinear parabolic PDEs. *SIAM J. Sci. Comput.*, 40(2):A629–A652, 2018.

[36] Wei Kang, P.K. De, and A. Isidori. Flight control in a windshear via nonlinear h_∞ methods. In *Proceedings of the 31st IEEE Conference on Decision and Control*, pages 1135–1142, 1992.

[37] Wei Kang and Lucas C. Wilcox. A causality free computational method for HJB equations with application to rigid body satellites. In *AIAA Guidance, Navigations, and Control Conference*. American Institute of Aeronautics and Astronautics, 2015.

[38] Wei Kang and Lucas C. Wilcox. Mitigating the curse of dimensionality: Sparse grid characteristics method for optimal feedback control and HJB equations. *Computational Optimization and Applications*, 68(2):289–315, Nov 2017.

[39] Wei Kang and Lucas C. Wilcox. Solving 1D conservation laws using Pontryagin's minimum principle. *Journal of Scientific Computing*, 71(1):144–165, 2017.

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

[41] J Kierzenka and Lawrence Shampine. A BVP solver that controls residual and error. *European Society of Computational Methods in Sciences and Engineering (ESCMSE) Journal of Numerical Analysis, Industrial and Applied Mathematics*, 3:27–41, 01 2008.

[42] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint: arXiv:1412.6980*, 2014.

[43] Dieter Kraft. A software package for sequential quadratic programming. Technical report, Deutsche Forschungs und Versuchsanstalt für Luft und Raumfahrt – Institut für Dynamik der Flugsysteme, 1988.

[44] 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.

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

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

[47] D.L. Lukes. Optimal regulation of nonlinear dynamical systems. *SIAM Journal on Control*, 7(1):75–100, 1969.

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

[49] Philippe Moireau and Dominique Chappelle. Reduced-order unscented Kalman filtering with application to parameter identification in large-dimensional systems. *ESAIM: COCV*, 17(2):380–405, 2011.

[50] Carmeliza Navasca and Arthur. J. Krener. Patchy solutions of Hamilton-Jacobi-Bellman partial differential equations. In A. Chiuso, S. Pinzoni, and A. Ferrante, editors, *Modeling, Estimation and Control*, volume 364 of *Lecture Notes in Control and Information Sciences*, pages 251–270. Springer Berlin Heidelberg, 2007.

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

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

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

[54] Chris Phelps, Qi Gong, Johannes O. Royset, Claire Walton, and Isaac Kaminer. Consistent approximation of a nonlinear optimal control problem with uncertain parameters. *Automatica*, 50(12):2987 – 2997, 2014.

[55] Chris Phelps, Johannes O. Royset, and Qi Gong. Optimal control of uncertain systems using sample average approximations. *SIAM Journal on Control and Optimization*, 54(1):1–29, 2016.

[56] 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.

[57] 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*, 2017.

[58] I. Michael Ross, Qi Gong, and Pooya Sekhavat. Low-thrust, high-accuracy trajectory optimization. *Journal of Guidance, Control, and Dynamics*, 30(4):921–933, 2007.

[59] Marta Savage. Design and hardware-in-the-loop implementation of optimal canonical maneuvers for an autonomous planetary aerial vehicle. Master's thesis, Naval Postgraduate School, Monterey, California, 2012.

[60] Richard Shaffer, Mark Karpenko, and Qi Gong. Unscented guidance for waypoint navigation of a fixed-wing uav. In *2016 American Control Conference (ACC)*, pages 473–478, July 2016.

[61] Yuval Tassa and Tom Erez. Least squares solutions of the HJB equation with neural network value-function approximators. *IEEE Transactions on Neural Networks*, 18(4):1031–1041, July 2007.

[62] Francesco Topputo and Chen Zhang. Survey of direct transcription for low-thrust space trajectory optimization with applications. *Abstract and Applied Analysis*, 2014.

[63] Lloyd N. Trefethen. *Spectral Methods in MATLAB*. Society for Industrial and Applied Mathematics, 2000.

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

[65] 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.

[66] 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.

[67] 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.

[68] E. A. Wan and R. Van Der Merwe. The unscented Kalman filter for nonlinear estimation. In *Proceedings of the IEEE 2000 Adaptive Systems for Signal Processing, Communications, and Control Symposium (Cat. No.00EX373)*, pages 153–158, 2000.

[69] Eric Xu. Pyipopt. *GitHub repository. URL: https://github. com/xuy/pyipopt*, 2014.

[70] 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.