Using Reinforcement Learning to Shrink Fractional Dimensions for Fun and Profit

Anonymous Author(s)
Affiliation
Address
email

Abstract:
A key limitation in using various modern methods of machine learning in developing feedback control policies is the lack of appropriate methodologies to analyze their long-term dynamics, in terms of making any sort of guarantees (even probabilistically) about robustness. The central reasons for this are, arguably, largely due to the so-called curse of dimensionality, combined with the black-box nature of the resulting control policies themselves. This paper aims at the first of these issues. Although the full state space of a system may be quite large in dimensionality, it is a common goal within model-based feedback control to design algorithms to drive the dynamics to lower-dimensional manifolds within the full state space. Whether this is done explicitly or implicitly, it is often a key feature of most model-based control methods that the resulting closed-loop systems demonstrate dominant dynamics that are rapidly driven to some lower-dimensional sub-space within the full state space. The aims of this paper are modest yet, to the best of the authors' knowledge, also quite novel: we explore approaches to encourage such dimensionality reduction during (model-free) machine learning. Of particular note is the fact that the long-term dynamics of a nonlinear system need not converge to some integer dimension space (e.g., 2D or 3D, etc.). Many dynamic systems, such as chaotic systems, are driven to a fractional dimensional sub-space. We observe such fractional dimensionality in the closed-loop dynamics of systems controlled via various reinforcement learning algorithms. In this work, we focus on methods to influence this dimensionality, using the Augmented Random Search (ARS) algorithm as an particular example.

Keywords: Locomotion, Reinforcement Learning

1 Introduction

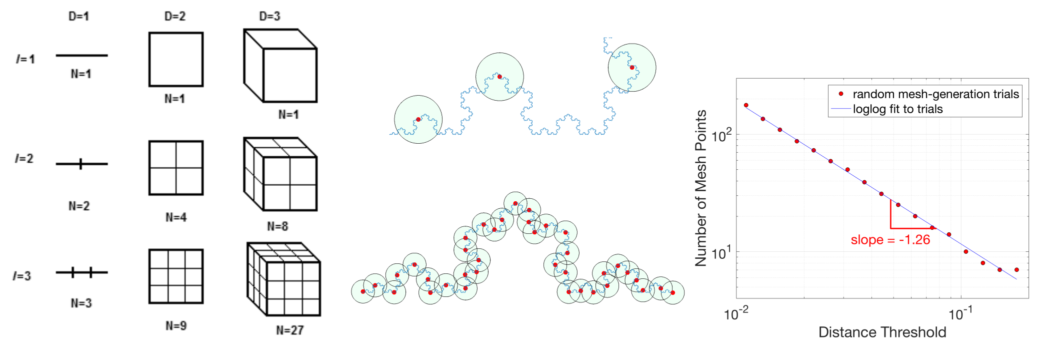
The availability of computation as a resource has been growing exponentially since at least the 1970s, and there is every indication that this resource will continue to become cheaper and more available well into the conceivable future. Researchers have been able to leverage the large amounts compute available to better control robotic systems, and advances in computational capacity and algorithmic development continue to open up new domains. One promising manifestation of this is model-free reinforcement learning, a branch of machine learning which allows an agent to interact with its environment and autonomously learn how to maximize some measure of reward. The promise here is to allow researchers to solve problems for systems that are hard to model, and/or that the user doesn't know how to solve themselves. Recent examples in the context of robotics include controlling a 47 DOF humanoid to navigate a variety of obstacles [1], dexterously manipulating objects with a 24 DOF robotic hand [2], and allowing a physical quadruped robot to run [3], and recover from falls [4].

In this paper we study legged locomotion. This class of problems is notoriously difficult, and as a result reinforcement learning is a popular tool to throw at it. We would argue that hand-designed, model based control still represents the state of the art (a la Boston Dynamics), but RL has been a fruitful approach. There are examples of learned policies outperforming hand designed ones [3], and there is good reason to believe these learning methods will continue their current trajectory of increasing performance gains and ease of use. But these algorithms have a serious draw back in that they are mostly black boxes. It is an open challenge to figure out what exactly it is that your RL agent has learned. If all you know is that one of your agents achieved very high reward, it is not clear how to verify that this system is safe and sensible in all the regions of state space it will visit during its life. Nor can we necessarily say anything about the stability or robustness properties of the system. Recent work [5] has used so-called mesh-based tools to examine precisely these questions.

However, utility of any mesh-based tool to accurately discretize a state-space is limited, due to the curse of dimensionality. In practice, these methods are only able to work on relatively high dimensional systems if the reachable space grows at a rate that is much smaller than the exponential growth of the full state space the system within which it is embedded. To expand these methods to higher dimensional systems. We will need to find ways to keep the volume of visited states from expanding commensurately. One way to quantify this rate of growth is by using one of the several notions of ”fractional dimensions” from fractal geometry.

In this work, we discuss an efficient meshing algorithm, which we call box meshing. We show that this approach makes calculating the so called mesh dimension feasible in the context of reinforcement learning. We also propose using other notions of fractional dimension from the literature as a proxy for the property we care about. We then show that reinforcement learning agents can be trained to shrink these measures by post processing their reward function. We present the results of this training, and finally present some brief analysis of the resulting structure for select policies.

2 Meshing & Fractional Dimensions



(a) Scaling in different dimensions (b) A non uniform mesh (c) Calculating the mesh dimension

Figure 1: credit(a): [6], credit(b),(c): [7]

Let’s say we have a continuous set S that we want to approximate by selecting a discrete set M composed of regions in S. We will call this set M a mesh of our space. Figure 1(a) shows some examples of this: a line is broken into segments, a square into grid spaces, and so on. The question is: as we increase the resolution of these regions, how many more regions N do we need? Again, Figure 1(a) shows us some very simple examples. For a D dimensional system, if we go from regions of size d to d/k, then we would expect the number of mesh points to scale as $N \propto k^D$. But not all systems will scale like this, as 1(b) and 1(c) illustrate. Figure 1(b) is an example of a (rather famous) curve embedded in a two dimensional space. The question of how many mesh points are required must be answered empirically. Going backwards, we can use this relationship to assign a notion of ”dimension” to the curve.

$$D_f = -\lim_{k \to 0} \frac{\log N(k)}{\log k} \tag{1}$$

What we are talking about is called the Minkowski–Bouligand dimension, also known as the box counting dimension. This dimension need not be an integer, hence the name ”fractional dimension”. As a practical matter, we use the slope of the log-log plot of mesh sizes vs d to calculate this, rather than taking a limit. This is one of many measures of ”fractional dimension” that that emerged from the study of fractal geometry. Although these measures were invented to study fractals, they can still be usefully applied to non-fractal sets.

In [8], Saglam and Byl introduced a technique that is able to simultaneously build a non-uniform mesh of a reachable state space while developing robust policies for a bipedal walker on rough terrain. Having a discrete mesh allows for value iteration over several candidate controllers, which found a robust control policy. In addition this mesh allows for the construction of a state transition matrix, which was used to calculate the mean first passage time [9], a metric that quantifies the expected number of steps a meta-stable system can take before falling.

Since its introduction, meshing in this fashion has been used for designing walking controllers robust to push disturbances [7], to design agile gaits for a quadruped [10], and to analyze hybrid zero dynamics (HZD) controllers [11]. There has also been recent work to use these tools to analyze policies trained by deep reinforcement learning [5]. A long term goal and motivation for this work is to take a high performance controller obtained via reinforcement learning, and extract from it a mesh-based policy that is both explainable and amenable to analysis.

2.1 Box Meshing

Our primary improvement to the prior work on meshing is to introduce something we call box meshing. Prior, a new mesh point could take any value in the state space. To determine if a new state is already in the mesh, we would compute a distance metric to every point in the mesh, and check if the minimum was below our threshold. Thus, building the mesh was an $O(n)^2$ algorithm. By contrast, in box meshing we a priori divide the space uniformly into boxes with side length d. We identify any state s with a key obtained by: key $=$ round$(\frac{s}{d})d$, where round performs an element-wise rounding to the nearest integer. We can then use these keys to store mesh points in a hash table. Using this data structure, we can still store the mesh compactly, only keeping the points we come across. However, insertion and search are now $O(1)$, and so building the mesh is $O(n)$. This is very similar to non-hierarchical bucket methods, which are well studied spatial data structure [12], although we are using them for data compression here. In the prior meshing work, this sort of speedup would be minor, the run-time is dominated by the simulator or robot. However, this speedup does open some new possibilities: most poignantly, it makes calculating the mesh dimension during reinforcement learning plausible.

2.2 Algorithmic Box Mesh Dimension

The ”mesh dimension” is the quantity extracted from the slope of the log log plot of mesh sizes vs d values. For this paper, it is assumed that the mesh algorithm being used for this calculation is the box mesh. Automatically computing the mesh dimension of a data set generated from learning agent with speed, accuracy, and robustness is very challenging. A single trajectory provides only a small amount of data, which adds significant noise to the mesh sizes. Agents might do things like fall over and generate extremely short trajectories, or learn a trajectory that ”stands in place”, which can lead to numerical errors. Finally, every decision is a trade-off between accuracy and speed. Model-free RL is predicated on having a huge number of rollouts to learn from, and we would like for any mesh-dimension quantification algorithm to be fast enough so as to not dominate the total learning time. With these factors in mind, we introduce two box mesh dimensions. The ”normal” mesh dimension does the linear fit, but intentionally errs on the side of including flat parts of the graph, and therefore tends to underestimate the true mesh dimension. We then have the conservative mesh dimension, which takes the largest slope in the log log relationship, thus tending to overestimate the true mesh dimension. Neither of these measures are correct, but taken together they can bound the mesh dimension, and as we will see they can be useful on their own.

2.3 Variation Estimators

As discussed, computing the mesh dimension automatically is fraught with peril, in many practical scenarios. But there are also many other, different metrics one might consider, to give various approximations to the fractional dimension we seek to estimate. Gneiting et al. [13] compare a number of these estimators, and submit that the variation estimator [14] offers a very good trade off

Algorithm 1: Create Box Mesh 2.1

```
Input: State set $S$, box size d.
Output: Mesh size m.
Initialize: Empty hash table M.
for $s \in S$ do
    $\bar{s}$ = Normalize(s)
    key = round($\bar{s}$ / d)d
    if $s \in M$ then
        M[key]++
    else
        M[key]=1
    end
end
Return: M
```

Algorithm 2: Compute Box Mesh Dimension 2.2

```
Input: State set $S$.
Output: Mesh M.
Hyperparameters: scaling factor f, initial box size $d_0$.
Initialize: Empty list of mesh sizes H, empty list of d values D.
m = Size(CreateBoxMesh(S, $d_0$))
d = $d_0$
Append m to H, append d to D.
while $m < size(S)$ do
    d = d/f
    m = Size(CreateBoxMesh(S, d))
    Prepend m to H, prepend d to D.
end
while $m \neq 1$ do
    d = d*f
    m = Size(CreateBoxMesh(S,d))
    Append m to H, append d to D.
end
X = log d
Y = -log m
Mesh Dim: fit Y = gX + b, Return: g
Conservative Mesh Dim: w = greatest slope in Y over X Return: w
```

between speed and robustness. To obtain this estimator, first define the power variation of order p as:

$$P_p(X, l) = \frac{1}{(2n - l)} \sum_{i=l}^{n} |X_i - X_{i-l}|^p \tag{2}$$

Then, we define the variation estimator of order p as:

$$Dv_p(X) = 2 - \frac{\log P_p(X, 2) - \log P_p(X, 1)}{p \log 2} \tag{3}$$

The **madogram** estimator is the special case of (3) where p = 1, and the **variogram** is where p = 2.

3 Reinforcement Learning

The goal of reinforcement learning is to train an agent acting in an environment to maximize some reward function. At every timestep $t \in \mathbb{Z}$, the agent receives the current state $s_t \in R^n$, uses that to compute an action $a_t \in \mathbb{R}^b$, and the receives the next state s_{t+1}, which is used to calculate a reward $r : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}$. The objective is to find a policy $\pi_\theta : \mathbb{R}^n \to \mathbb{R}^m$

$$\arg\max_{\theta} \mathbb{E}_{\eta} \left[\sum_{t=0}^{T} r(s_t, a_t, s_{t+1}) \right] \tag{4}$$

Where $\theta \in \mathbb{R}^d$ is a set that parameterizes the policy, and η is a parameter representing the randomness in the environment. This includes the random initial conditions for episodes.

3.1 Post Processing Rewards

In order to influence the dimensionality of the resulting policies, we introduce various postprocessors, which act on the reward signals before passing them to the agent. These obviously modify the problem: in some sense the postprocessed environment is a completely different problem from the original. However our meta-goal is to train agents that achieve reasonable rewards in the base environment, while simultaneously exhibiting reduced dimensionality we are looking for. These postprocessors take the form:

$$R_*(\mathbf{s}, \mathbf{a}) = \frac{1}{D_*(\mathbf{s})} \sum_{t=0}^{T} r(s_t, a_t, s_{t+1}) \tag{5}$$

Where $\mathbf{s}, \mathbf{a}$ are understood to be an entire trajectory of state action pairs, and D_* is some measure of fractional dimension. The variational estimators can be used here directly, we creatively call these the **madogram postprocessor** and **variogram postprocessor**, respectively. The mesh dimensions require a little more care. We must first define a clipped dimension:

$$D_*^c = \text{clip}[D_*(\mathbf{s}_{t>Tr}), 1, D_t/2)] \tag{6}$$

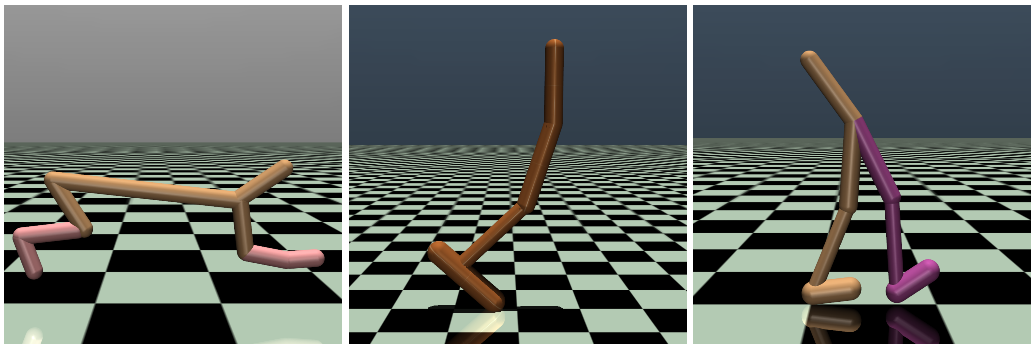
where D_t is the topological dimension, equal to the number of states in the system. Tr is a fixed timestep chosen to exclude the initial transients resulting from a system moving from rest to into a quasi-cyclical “gait”. In this paper we set Tr = 200 for all experiments. For comparison, the nominal episode length is 1000. The clipping is to ensure that the pathological trajectories that and RL agent sometimes generates don’t interfere with the training. It will also clip trajectories that terminate early, to prevent agents learning to fall over immediately to “game the system”. Half of the topological dimension proved to be a decent upper bound for the worst case dimensionality of each system. The **mesh and conservative mesh dimension postprocessors** use the clipped dimension. Finally, when $D_* = 1$ is used, we call the result is the **identity post processor**, since in this case the total reward is completely unchanged.

3.2 Environments

We examine a subset of the popular OpenAI Mujoco locomotion environments introduced in [15]. In particular, we evaluate our work on HalfCheetah-v2, Hopper-v2, and Walker2d-v2. These environments were chosen because they have a relatively high dimensionality (11-17 DOF), yet we believe can be made feasible for meshing based approaches. The state space consists of all joint / base positions and velocities, with the x (the ”forward”) position being held out, because we want a policy that is invariant along that dimension.

3.3 Augmented Random Search

In [16] Mania et al introduce Augmented Random Search (ARS) which proved to be efficient and effective on the locomotion tasks. Rather than a neural network, ARS used static linear policies, and compared to most modern reinforcement learning, the algorithm is very straightforward. The algorithm operates directly on the policy weights, each epoch the agent perturbs it’s current policy N times, and collects 2N rollouts using the modified policies. The rewards from these rollouts are used to update the current policy weights, repeat until completion. The algorithm is known to have high variance; not all seeds obtain high rewards, but to our knowledge their work in many ways represents



(a) HalfCheetah-v2 (b) Hopper-v2 (c) Walker2d-v2

Figure 2: The Mujoco locomotion environments

the state of the art on these benchmarks. Mania et al introduce several small modifications of the algorithm in their paper, our implementation corresponds to the version they call ARS-V2t.

3.4 Training

To keep things simple, we wanted to find one set of hyper parameters for all environments and post processors. These parameters were chosen by hand, with the parameters reported in Table 9 from [16] as a starting point. We tuned until our unprocessed learning achieved satisfactory results across all tasks. Again, ARS is known to have high variance between random seeds, and some seeds never learn to gather a large reward. The parameters we found are able to consistently solve the cheetah and walker; for the hopper, the algorithm learns a policy with high reward around half the time. This seems consistent with the performance reported in [16]. We train each postprocessor on 10 random seeds, the evaluation metrics are averages over 5 rollouts from each seed, and for the dimension metrics we use extended episodes of length 10,000 to get a more accurate measurement. The reported returns, and the training, both use the normal 1,000 step episodes. The variational postprocessors are trained for 750 epochs each. We found that the mesh postprocessors were getting very poor performance when trained from a blank policy. However, we found that we saw good results when these trials were initialized with a working policy. To that end, we used the results from the variational trials as the initial policies for the mesh trials. The mesh policies were then trained for an additional 250 epochs and the results reported.

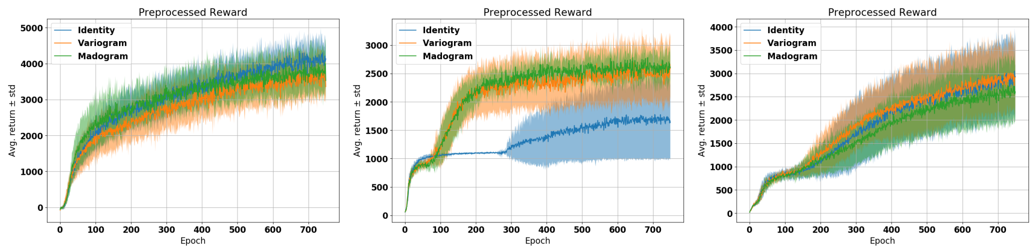
4 Results

4.1 Variational Postprocessors

It seems that the variational postprocessors had a modest effect the variational dimension, but that does not seem to correlate to a smaller mesh dimension, despite what our preliminary tests had led us to believe. The hopper and walker did have remarkable consistency in the variation dimensions they found; possibly this could be used to lower the variance in ARS. The fact that the variogram and madogram also got higher performance on the hopper task could support this claim. However without running many more trials and hyper parameter sweeps, that's not a claim that can be substantiated. These experiments show that 1) measures for fractional dimension can be influenced without adversely effecting the reward, and 2) that it is possible for an agent to shrink it's variogram and madogram dimensions without a large impact on its mesh dimension.

4.2 Mesh Dimension Postprocessors

The effects from these trials are far more pronounced, for all environments the mesh post processor had a significant impact on the resulting mesh dimension, and the conservative post processor had a significant effect on the conservative mesh dimensions. It's important to remember here, the dimensions reported represent lower and upper bounds for the actual mesh dimensions. There was also a corresponding and significant decrease in the unprocessed rewards. However with our meta

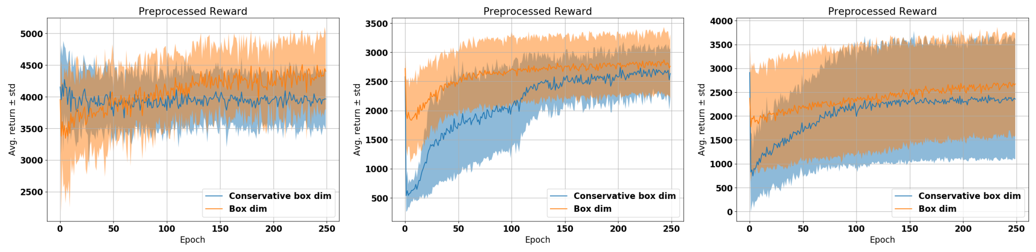


(a) HalfCheetah (b) Hopper (c) Walker

Figure 3: Reward curves for the variation postprocessors

Environment	Postprocessor	Variogram	Madogram	Mesh Dim.	Return
HalfCheetah-v2	Identity	1.71 ± .03	1.42 ± .05	2.36 ± .61	5545 ± 593
	Variogram	1.68 ± .01	1.36 ± .02	2.06 ± .60	5136 ± 851
	Madogram	1.65 ± .02	1.31 ± .04	2.09 ± .64	5234 ± 950
Hopper-v2	Identity*	1.61 ± .14	1.22 ± .28	1.03* ± .71	2063 ± 1052
	Variogram	**1.51 ± .02**	**1.03 ± .04**	1.58 ± .54	3299 ± 711
	Madogram	**1.51 ± .002**	**1.02 ± .004**	1.57 ± .36	3449 ± 146
Walker2d-v2	Identity	1.68 ± .35	1.36 ± .71	2.14 ± .29	3742 ± 1038
	Variogram	**1.54 ± .07**	**1.07 ± .01**	1.85 ± .54	3779 ± 894
	Madogram	**1.53 ± .01**	**1.06 ± .02**	1.99 ± .53	3414 ± 1025

Figure 4: Mesh dimensions and returns for trajectories after training. See 3.4 for details
* This includes policies which learned to ”stand still”, which lowers the average mesh dimension considerably see discussion



(a) HalfCheetah (b) Hopper (c) Walker

Figure 5: Reward curves for mesh dimension postproccesor runs.

Environment	Postprocessor	Mesh Dim.	Cons. Mesh Dim.	Return
HalfCheetah-v2	Identity	2.31 ± 0.71	7.34 ± 1.56	5469 ± 823
	Mesh Dim	**0.66 ± 0.51**	**2.55 ± 1.52**	4962 ± 598
	Cons. Mesh Dim.	**1.06 ± 1.13**	**2.83 ± 1.27**	4432 ± 539
Hopper-v2	Madogram*	1.62 ± .27	4.68 ± 0.82	3461 ± 119
	Mesh Dim.	1.13 ± .02	3.54 ± 0.96	2941 ± 538
	Cons. Mesh Dim.	1.27 ± .50	2.98 ± 1.48	3020 ± 337
Walker2d-v2 (walking seeds)**	Identity	2.13 ± 0.31	4.62 ± 1.03	3758 ± 1037
	Mesh Dim.	1.21 ± 0.06	4.09 ± 1.03	3339 ± 887
	Cons. Mesh Dim.	1.89 ± 0.42	3.10 ± 0.93	3359 ± 903
Walker2d-v2 (all seeds)**	Identity	2.13 ± 0.31	4.62 ± 1.03	3758 ± 1037
	Mesh Dim.	1.04 ± 0.53	4.45 ± 1.19	3034 ± 1086
	Cons. Mesh Dim.	1.48 ± 0.67	2.27 ± 0.95	2556 ± 1378

Figure 6: Mesh dimensions and returns for trajectories after training. See 3.4 for details
* Because our instance of ARS does not consistently perform well on the hopper, we instead use madodiv for the seed policies.
** See 4.2

goal of training agents that have acceptable reward but which are more amenable to meshing, this is a more than acceptable trade. In the case of walker, several seeds (4 for the conservative dim., 3 for mesh dim.), ”forget how to walk”, and learn a policy that stands in place. This certainly has a low dimensionality, but is not very useful, to be complete we include statistics from the seeds that learned a gait, and for all 10 seeds, including the standing policies.

5 Analysis

We now examine the learned behavior for one of the more notable policies. By far the most dramatic effect from the tables above was the mesh dimension postprocessors on the cheetah. Both measures of dimension shrunk by 2-4 times. Figure 7 presents data for this case.

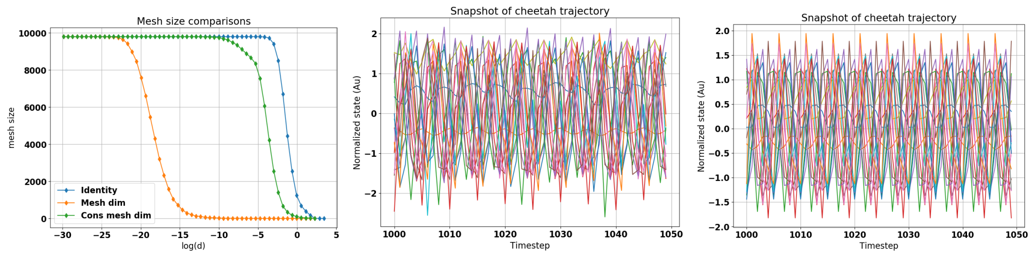


Figure 7: Left: mesh size comparisons across the post processors. middle: Trajectory after training normal ARS for 750 epochs. Right: Trajectory after training 250 additional epochs with the mesh dimension post processor

Toward more intuitively understanding this data, a few comments are worth making, first. We have discussed the mesh dimension rather abstractly so far. In visualizing what this really means, imagine two different gait cycles. In one case, there is a general pattern to the motion, but it wanders in a noisy-looking way, like a “signature” that does not quite match up, cycle after cycle. As motions become closer to being exact limit cycles, there is a more clear pattern of repetition, exactly analogous to re-tracing the same path, again and again, within the state space. Such a more tightly-structured limit cycle nature in turn results in a significantly lower-dimensional set of states being visited.

We can see from the mesh size curves in Figure 7 that there is an overwhelming difference in the mesh sizes between the mesh dimension post processor and the other two. To put this in perspective, before the extra 250 epochs of training, if given a mesh size of .01, the agent would need a unique mesh point for every single state in the 10,000 state trajectory. After the additional training, the agent cant represent all 10,000 points with just 5 mesh points! Examining the trajectory in Figure 7 gives us clues, it seems that the agent learned an extremely tight limit cycle. It’s a bit of a weird limit cycle, being only 5 cycles long, and this is clearly the sort of thing that can only happen in a noiseless simulation, but nonetheless we think this is interesting and surprising behavior.

6 Conclusion

In this work, we introduced a technique to influence the fractional dimension of the closed-loop dynamics of a system through the use of novel, dimensionality-based modifications to the cost functions for reinforcement learning policies. We demonstrate this technique on several benchmark tasks, and we briefly analyze a resulting policy to verify the outcome, demonstrating a much smaller mesh dimension without a large loss in reward or function.

7 APPENDIX

Hyper Parameters

ARS: For all environments $\alpha = .02$, $\sigma = .025$, $N = 50$, $b = 20$.
MeshDim: f = 1.5, d_0 = 1e-2

References

[1] N. Heess, D. TB, S. Sriram, J. Lemmon, J. Merel, G. Wayne, Y. Tassa, T. Erez, Z. Wang, S. M. A. Eslami, M. Riedmiller, and D. Silver. Emergence of Locomotion Behaviours in Rich Environments. *arXiv:1707.02286 [cs]*, July 2017. URL `http://arxiv.org/abs/1707.02286`. arXiv: 1707.02286.

[2] OpenAI, M. Andrychowicz, B. Baker, M. Chociej, R. Jozefowicz, B. McGrew, J. Pachocki, A. Petron, M. Plappert, G. Powell, A. Ray, J. Schneider, S. Sidor, J. Tobin, P. Welinder, L. Weng, and W. Zaremba. Learning Dexterous In-Hand Manipulation. *arXiv:1808.00177 [cs, stat]*, Aug. 2018f. URL `http://arxiv.org/abs/1808.00177`. arXiv: 1808.00177.

[3] J. Hwangbo, J. Lee, A. Dosovitskiy, D. Bellicoso, V. Tsounis, V. Koltun, and M. Hutter. Learning agile and dynamic motor skills for legged robots. *Science Robotics*, 4(26): eaau5872, Jan. 2019. ISSN 2470-9476. doi:10.1126/scirobotics.aau5872. URL `http://robotics.sciencemag.org/lookup/doi/10.1126/scirobotics.aau5872`.

[4] J. Lee, J. Hwangbo, and M. Hutter. Robust Recovery Controller for a Quadrupedal Robot using Deep Reinforcement Learning. *arXiv:1901.07517 [cs]*, Jan. 2019. URL `http://arxiv.org/abs/1901.07517`. arXiv: 1901.07517.

[5] N. Talele and K. Byl. Mesh-based Tools to Analyze Deep Reinforcement Learning Policies for Underactuated Biped Locomotion. 2019. URL `http://arxiv.org/abs/1903.12311`.

[6] Brendan Ryan / Public domain. Fractal Dimension Example, 2020.

[7] N. Talele and K. Byl. Mesh-based methods for quantifying and improving robustness of a planar biped model to random push disturbances. *Proceedings of the American Control Conference*, 2019-July:1860–1866, 2019. ISSN 07431619. doi:10.23919/acc.2019.8815226.

[8] C. Oguz Saglam and K. Byl. Robust Policies via Meshing for Metastable Rough Terrain Walking. In *Robotics Science and Systems*, 2015. doi:10.15607/rss.2014.x.049.

[9] K. Byl and R. Tedrake. Metastable walking machines. *International Journal of Robotics Research*, 28(8):1040–1064, 2009. ISSN 02783649. doi:10.1177/0278364909340446.

[10] K. Byl, T. Strizic, and J. Pusey. Mesh-based switching control for robust and agile dynamic gaits. *Proceedings of the American Control Conference*, pages 5449–5455, 2017. ISSN 07431619. doi:10.23919/ACC.2017.7963802.

[11] C. O. Saglam and K. Byl. Meshing hybrid zero dynamics for rough terrain walking. *Proceedings - IEEE International Conference on Robotics and Automation*, 2015-June(June):5718–5725, 2015. ISSN 10504729. doi:10.1109/ICRA.2015.7140000.

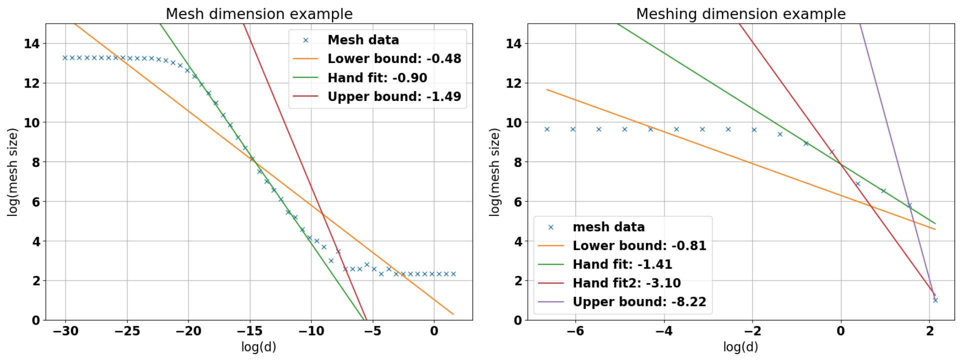
[12] H. Samet. *The design and analysis of spatial data structures.pdf*. Addison-Wesley, 1990.

[13] T. Gneiting, H. Ševčíková, and D. B. Percival. Estimators of fractal dimension: Assessing the roughness of time series and spatial data. *Statistical Science*, 27(2):247–277, 2012. ISSN 08834237. doi:10.1214/11-STS370.

[14] X. Emery. Variograms of order ω: A tool to validate a bivariate distribution model. *Mathematical Geology*, 37(2):163–181, 2005. ISSN 08828121. doi:10.1007/s11004-005-1307-4.

[15] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym, 2016.

[16] H. Mania, A. Guy, and B. Recht. Simple random search of static linear policies is competitive for reinforcement learning. *Advances in Neural Information Processing Systems*, 2018-December(NeurIPS):1800–1809, 2018. ISSN 10495258.



(a) High resolution curve of a well behaved policy (b) Run time resolution curve of a typical policy

Figure 8: Mesh curve and mesh dimension examples

8 Supplemental Material

8.1 Mesh Dimension Examples

Figure 8 illustrates two examples of the curves used to compute the mesh dimension. Recall that to compute the mesh dimension, we choose several values for d, the box length, and for each d construct a mesh using that box size. The x axis of these plots represents the log of the box length used, the y axis represents the log of size of the mesh created. For each curve, we display the lower bound and upper bound for the dimension as computed by algorithm 2, as well as several hand fits of the data. We hope that figure 8a makes clear what a close to ideal situation looks like, and provide intuition as to why the mesh dimension and conservative mesh dimension bound the quantity we are trying to measure. Figure 8b serves to illustrate some of the problems with making an algorithmic measure of the dimension. There is much less data to work with due to performance constraints, which causes a large amount of noise on the estimate of the mesh dimension. Indeed even fitting this data by hand becomes a challenge and we provide two fits which can both be argued to be ”correct”. Which of these two represents the quantity we care about depends on the exact system being used and the purpose of the meshes we want to build with the resulting policy.

8.2 Implementation Details

For performance reasons, the mesh dimension algorithm does not actually create meshes until the mesh size equals the total data size, but rather until the mesh size is 4/5 the total data size. Figure 8a shows a typical mesh curve, and we can see the long tail of values with mesh sizes close to the maximum value. Not much useful information is gained from this and it is wasting time, so we stop early. We do not place the same limitation on the lower size of the mesh, since typically the mesh size hits one much more rapidly, again figure 8a illustrates this. In addition implement a minimum size for d, set to 1e-9 in this work to avoid numerical errors.

The normalization done during box creation uses a running mean and standard deviation of all states seen so far during training. These stats are saved and used for evaluation as well, we found that the conservative mesh dimension is very sensitive to the normalization used, but that the other metrics where not.