Analysis of Data-Reduction Techniques Used in Dynamical Systems Modeling

Jessica Lee

Under the direction of

Dr. Katie Byl
Associate Professor
University of California, Santa Barbara

July 2020 - November 2020

Abstract

In robotics, the state space of a dynamical system is a vector space where each vector corresponds to a possible configuration of the robot. Each component of the state vector is a position or a velocity of some part of the system. Adding more degrees of freedom increases the size of the state space exponentially—a concept known as the curse of dimensionality. For complicated systems, it becomes computationally impossible to analyze the state space, a step that is necessary for designing control policies that would, for example, enable a walking robot to maintain its stability. Thus arises the need for techniques to reduce the information contained in given data while maintaining accuracy. This research explores two of these methods: meshing and principal components analysis. A mesh discretizes a set of points in continuous space according to a specified box length and groups together nearby points. Principal components analysis makes use of the fact that attractors commonly have self-similar structures and exist in a lower number of dimensions than the original space. In this study, both techniques are implemented on the dataset outputted by a simulation of the 2D planar walker, and the runtimes are graphed as a function of accuracy to the original dataset. It is shown that the runtime decreases as expected. This concept can be applied to find the best tradeoff between time and accuracy for designing an optimal feedback policy for a robotic system.

1 Introduction

Dynamical systems are systems that evolve over time, whether approaching an equilibrium state, a repeating cycle, or something more complicated [1]. Simple examples of dynamical systems include a pendulum or a moving particle [2]. In robotics, a legged robot is classified as a dynamical system because its future state depends on its current state. The entire system's state can be represented as a vector of positions and velocities, either linear or angular, or a combination of both [3]. At every point in time, the system is in a state that depends on the previous one and determines the next one. Over a period of time, as the system changes, so does the vector that represents it. This vector resides in a vector space known as the state space, which, for a complicated robotic system, can be upwards of a hundred dimensions [2]. One version of the Multi-Joint dynamics with Contact (MuJoCo) simulation planar walker, for example, is a five-link humanoid model that moves in the yz-plane (as opposed to three-dimensional space). Despite its "simplicity," the state vector consists of 17 dimensions. Analyzing a robot's movement and its corresponding path in state space becomes computationally difficult in these high-dimensional spaces for a reason known as the curse of dimensionality, which states that the number of points increases exponentially on the number of dimensions [4].

Thus, arises a need to somehow simplify a set of points to find locations where the system migrates to over time. These are known as attractors, and they can be fixed points, limit cycles, or strange attractors [1]. Strange attractors in particular are difficult to find because they are complex geometric shapes and might not be as visually obvious as simple limit cycles. Techniques to reduce the dimension of the data generated by legged locomotion systems in this research include principal component analysis, as well as the quantization of sets of points in the same general area. From this, a set of finite states, the number of which is less than or equal to the original number of points, can be used to create a transition matrix.

The necessity of reducing both the dimension and number of points becomes apparent during this construction of a large matrix, as the size of the matrix is equal to the number of states. Performing this calculation for a complicated system would push the limits of computing power.

For this reason, the goal of this research is to analyze actual simulation data of legged locomotion systems in a way that is both efficient and accurate. When applied to robots in the real world, this would allow autonomous robots to make quick judgements about their state and form decisions based on these perceptions, which is one more step towards achieving artificial intelligence.

2 Methods

2.1 Data Collection

The data used in this experiment are the outputs of simulations run by Sean Gillen at the University of California, Santa Barbara. Three different models were run in the MuJoCo physics simulation engine.

Figure 1: From left to right: walker, hopper, half cheetah [5]

Each model was run with a random seed and from five different initial conditions. For each run, the output was three matrices: observations, actions, and rewards. The observations matrices are the only data considered in this research, as they represent the state vector over

discrete time intervals. All observations matrices consist of 1000 points, which are observed at these intervals. Before performing any further analysis, the raw data are first normalized to have mean 0 and variance 1.

2.2 Principal Component Analysis

Principal Component Analysis (PCA) is a common way of processing high-dimensional data into a smaller number of dimensions. It finds the most important, or "principal," components of a large data set and expresses the set of points using only those coordinates, thus getting rid of some of the less important dimensions and making the data easier to process [6]. For this analysis, the `pca` function in MATLAB is implemented in a helper function that also takes into account a desired minimum percent of retained variance. The data is projected onto the lower-dimensional space and saved as the new set of points.

2.3 Meshing

Creating a mesh of the data allows for the discretization of points in the state space and groups together "nearby" states. This is implemented in MATLAB using the floor function, mapping each point in each dimension to the nearest integer interval of r, the mesh size. $\frac{r}{2}$ is added back to all components to center it to the middle of the interval. Once the points have been quantized according to the mesh, they can be mapped to the integers 1 to k, where k is the number of unique points after the mesh. This is implemented using a hash function whose coefficients are randomly generated between 0 and 1. Once each point is mapped to a real number, a for-loop is used to assign these to the integers from 1 to k.

2.4 Transition Matrix

Transition matrices are used to represent finite state systems and how they evolve over time. They are defined as

$$T_{ij} = \Pr(\mathrm{X}[n+1] = x_j \mid \mathrm{X}[n] = x_i) \qquad [7] \tag{1}$$

where $\mathrm{X}[n]$ is the system's state at time n and x_i is the ith state. The construction of the transition matrix for the given data is straightforward once each point is assigned to a state. The ordered list is paired with itself, but offset by one index placement, and these are used as the input arguments for the `sparse` matrix command. A large matrix is constructed, with all entries equal to zero. Memory is allotted only for nonzero entries, thus saving a significant amount of space. If the maximum number of points in any given state is n, then the maximum number of nonzero entries in any row is n, so the majority of the entries in a large transition matrix are 0.

Once the transition matrix is constructed, an eigenanalysis is performed to extract long term behavior patterns from the data. This is done using `eigs`, a MATLAB function for finding the first six eigenvalues and eigenvectors of a sparse matrix. The largest eigenvalue, which is paired with the principal eigenvector, represents the eventual equilibrium of the system over a long period of time. The vector is a distribution of probabilities, with each index corresponding to the state with that number as its ID. This information reveals which states are most visited and therefore more important to the data.

3 Results

3.1 Principal Component Analysis

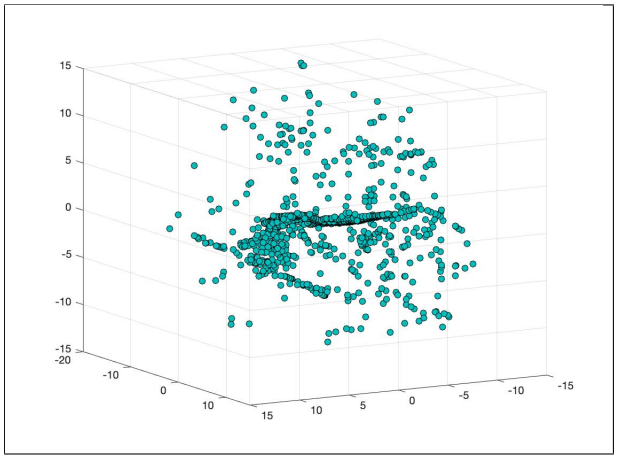

Figure 2: Walker data projected onto three principal components

When PCA is implemented with the minimum percent of variance retained set to 80%, the data can be projected from 17 to 6 dimensions.

3.2 Transition Matrix

The transition matrix generated for the walker model with specific $r = 1$ is a 509-by-509 sparse matrix whose rows each sum to 1.

$$\lambda_{\text{critical}} = 0.9988 \tag{2}$$

The greatest eigenvalue of this matrix is 0.9988, which should theoretically be 1 by the Perron Frobenius theorem [8], but the calculation is likely affected by roundoff error in MATLAB. The corresponding eigenvector for this largest eigenvalue is a column vector that, when re-scaled, consists of 509 nonnegative entries; the greatest is approximately $2.48 \cdot 10^{-3}$ and the least is 0.

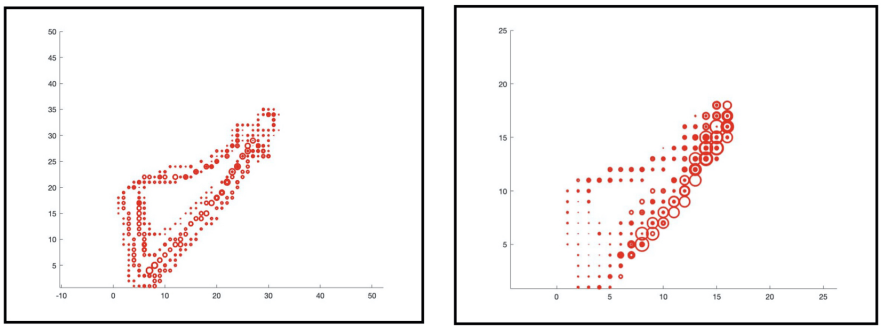

Figure 3: States projected onto dimensions 1 and 2 (Left: $r = 0.10$, Right: $r = 0.20$)

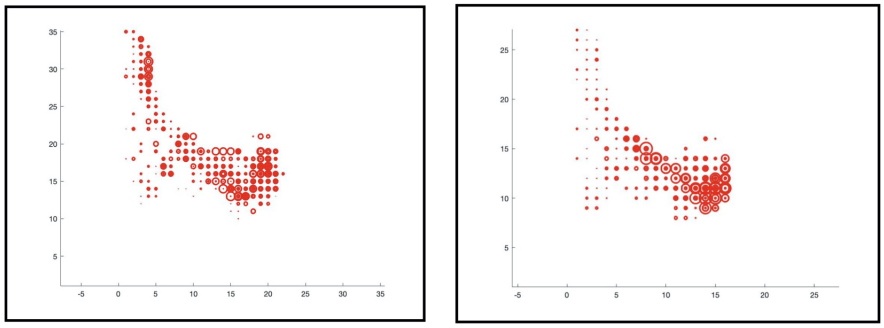

Figure 4: States projected onto dimensions 1 and 9 (Left: $r = 0.15$, Right: $r = 0.20$)

Other values of r were useful for plotting the states most visited by the system, as calculated by the transition matrix. Larger markings represent a higher probability in the corresponding index of the principal eigenvector.

3.3 Runtime

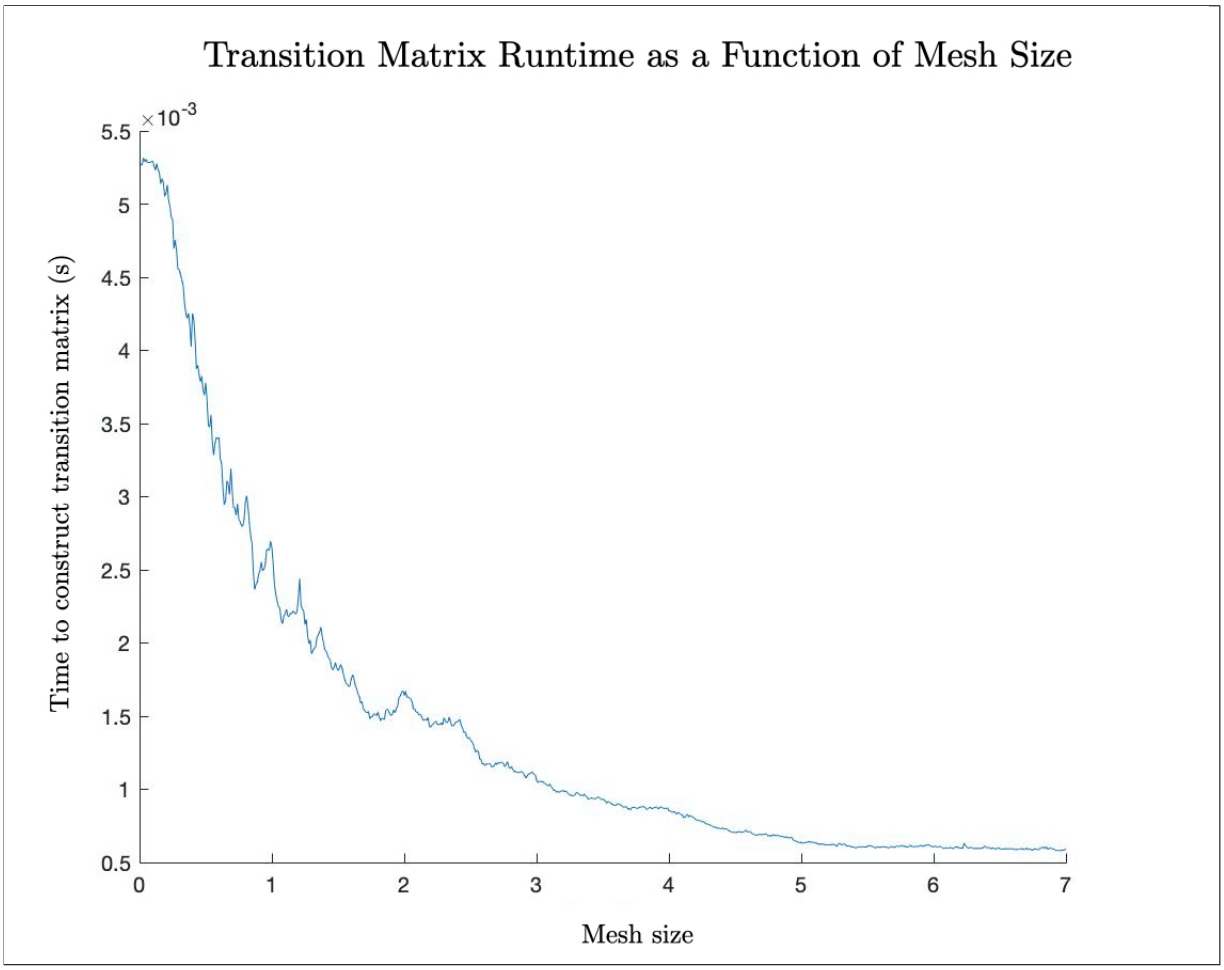

Figure 5: Runtime as a function of mesh length over 100 trials

The mesh size was set to a variable and iterated over a for-loop. `tic` and `toc` measured the runtime for the construction of the transition matrix. This was performed for mesh size from 0.01 to 7 at intervals of 0.01, over a total of 100 trials.

4 Discussion

This research aims to find a lower-dimensional representation of the simulation data, given that strange attractors are usually self-similar and thus of lower-than-expected dimensionality. PCA gives a representation of the data in only 6 dimensions while retaining 80% of the information. This indicates that attractors reside in a much lower dimensional space than they first appear. A possible implication of this would be that the computing time for processing similar structures for robots could be reduced if some of the information is redundant [9].

The relevant runtimes are for the following:

1. Creating the mesh
2. Constructing the transition matrix
3. Analyzing the transition matrix

Each step's time depends on the number of total points n, the dimension of each point d, and the specified mesh size r.

4.1 Creating the mesh

See Appendix A. The points are scaled to have mean 0 and variance 1 in each dimension, then shifted to begin at 0, a process whose runtime is bounded above by $\mathrm{O}(n \times d)$. The meshing takes an additional $\mathrm{O}(4(n \times d))$, as each component of each point is divided by r, the floor function taken, multiplied by r, then added to $\frac{r}{2}$. In total, this step is $\mathrm{O}(n \times d)$.

4.2 Constructing the transition matrix

See Appendix B. Some of the most time-consuming steps in this process include the conversion to hashed points ($\mathrm{O}(n \times d)$), sorting rows ($\mathrm{O}(n \log n)$, used twice), and scaling

rows to add to 1 ($O(k^2)$, where k is the number of distinct states). k can be expressed as a function of r, the mesh size, using the formula Strogatz provides:

$$x = \lim_{r \to 0} \frac{\ln N(r)}{\ln(1/r)} \qquad [1]$$

This equation states that the dimension x of a set of points can be estimated using the mesh technique where $N(r)$, which we have called k, is the number of points after implementing a mesh of size r. With some algebraic manipulation, it can be shown that $k = (1/r)^x$, so $k^2 = r^{-2x}$ where x is almost constant and can be treated as such. In total, the transition matrix construction step is $O(n \times d + n \log n + r^{-\alpha})$.

4.3 Analyzing the transition matrix

See Appendix C The most costly function in this step is `eigs`, "which uses several tricks that make the algorithm faster in the general case, but also make its convergence behavior and computational complexity much more difficult to analyze" [10]. However, one known upper bound is $O(n^3)$.

4.4 Further discussion

Both techniques explored in this research aim to address this issue - PCA reduces the dimensionality of the points, and quantizing the data reduces the number of points to store in memory. By clustering nearby points with a given mesh size to eventually label them as the same state, the number of discrete states dropped from 1000 to 509. For longer intervals of time, with many more than 1000 intervals, a 49% reduction in the number of states could drastically affect computing time [11]. From these states, the experimental data of where each point transitions in the next time interval provides the information necessary to construct a transition matrix. The principal eigenvector of the matrix represents the stable equilibrium state to which the system evolves over time [7]. The result of this data analysis suggests the

presence of multiple states of higher probability in the equilibrium state, the highest one being the most prominent state. If translated back to a vector in the original state space, this would represent the most common position of the walker over time.

Similar work has been done in this field to effectively reduce the dimension of the state space of a dynamical system [9, 11]. However, applying such techniques to a concrete and interactive model such as the five link planar walker is a way to quickly and efficiently test the practicality of these methods. The results indicate that while the state space of complicated dynamical systems remains difficult to analyze from a computing point of view, these techniques are promising for future research.

5 Future Work

Being able to find strange attractors for complicated systems is a problem that is not fully solved, but the techniques discussed in this research are a step in the right direction. Although only an analysis was performed, the most immediate successive research from here would be to apply these data-reducing techniques to other models. The ultimate goal is to be able to implement them in real world robots who must constantly take in measurements from their surroundings and react to the external forces pushing them off their intended path. Optimizing reaction time to these events is already an existing field of research, but perhaps the combined reduction of dimension and quantization of points would enable newfound progress in this field.

6 Conclusion

Complicated dynamical systems, especially real world humanoid robots, are represented by state spaces that can contain up to hundreds of dimensions. When the number of di-

mensions increases linearly, the number of points in a given area grows exponentially. For this reason, it becomes impossible to analyze the raw data from these complicated systems. This research studies the data from a relatively simple model - the five link planar walker - and applies two techniques with the intention of reducing the amount of information needed to accurately describe the model. Using principal component analysis, 17 dimensions are reduced to 3 with minimal loss of information, and the quantization of points condenses the data even further and allows for the construction of a transition matrix. An eigenanalysis of this matrix produces the equilibrium state of the system, which is a prediction of its long term behavior. These methods for analyzing the path of a model in state space can be applied where standard computational methods fail, and building upon this research may lead to future innovation in the area of artificial intelligence in robots.

7 Acknowledgments

I would like to thank my mentor Dr. Katie Byl for her dedication to my project amidst mentoring many other students, Sean Gillen and Nihar Talele for guiding me as well (and for their data), River Grace for his advice on writing and presentation skills, and RSI, CEE, and MIT for providing me with this opportunity to experience research. And thank you to my family - my parents, grandparents, and brother - for supporting me always.

References

[1] S. H. Strogatz. *Nonlinear dynamics and chaos: with applications to physics, biology, chemistry, and engineering*. CRC Press, 2019.

[2] F. Heilmann. The state space of complex systems, Jul 2005.

[3] K. M. Hangos, G. Miklos, and L. Rozalia. *Intelligent control systems an introduction with examples*. Kluwer, 2004.

[4] M. Verleysen and D. François. The curse of dimensionality in data mining and time series prediction. In J. Cabestany, A. Prieto, and F. Sandoval, editors, *Computational Intelligence and Bioinspired Systems*, pages 758–770, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[5] Y. Tassa, Y. Doron, A. Muldal, T. Erez, Y. Li, D. de Las Casas, D. Budden, A. Abdolmaleki, J. Merel, A. Lefrancq, T. Lillicrap, and M. Riedmiller. Deepmind control suite, 2018.

[6] S. Wold, K. Esbensen, and P. Geladi. Principal component analysis. *Chemometrics and Intelligent Laboratory Systems*, 2(1):37 – 52, 1987. Proceedings of the Multivariate Statistical Workshop for Geologists and Geochemists.

[7] K. Byl. Metastable legged-robot locomotion.

[8] O. Knill. Lecture 34: Perron frobenius theorem.

[9] B. Moore. Principal component analysis in linear systems: Controllability, observability, and model reduction. *IEEE Transactions on Automatic Control*, 26(1):17–32, 1981.

[10] What is the space complexity of built-in eigs function in matlab.

[11] D. Hartman and L. K. Mestha. A deep learning framework for model reduction of dynamical systems. In *2017 IEEE Conference on Control Technology and Applications (CCTA)*, pages 1917–1922, 2017.

Appendix A Creating the mesh

```
function [new_points] = reduce_box(old_points, radius)
new_points = radius * floor(old_points / radius);
new_points = new_points + radius/2;
end
```

Appendix B Constructing the transition matrix

```
function [matrix] = transition_mat(points)

num_points = size(points, 1);
num_dim = size(points, 2);
hash_coeff = rand(1, num_dim);
hash_points = points * hash_coeff';
hash_points = sortrows( [hash_points (1:num_points)'] );
hash_points = [hash_points zeros(num_points, 1)];

index = 1;
hash_points(1, 3) = index;
for i = 2:num_points
    if (hash_points(i, 1) - hash_points(i-1, 1))
        index = index + 1;
    end
    hash_points(i, 3) = index;
end
```

```
18 states_ordered = sortrows( [hash_points(:, 2) hash_points(:, 3)] )
       ;
19 states_ordered = states_ordered(:, 2);
20 num_states = max(states_ordered);
21
22 % Make transition matrix
23 states_shifted = circshift(states_ordered, -1);
24 states_ordered = states_ordered(1:(end-1));
25 states_shifted = states_shifted(1:(end-1)); % Last state goes
       nowhere
26
27 matrix = sparse(states_ordered, states_shifted, 1, num_states,
       num_states);
28 sums = sum(matrix, 2);
29 matrix = matrix ./ sums;
30 matrix(isnan(matrix)) = 0;
31
32 end
```

Appendix C Analyzing the transition matrix

```
1 function [] = eigenanalysis(T)
2
3 [Vecs,LambdaMat] = eigs(T')
4 Lambda = diag(LambdaMat)
5
```

```
6  [Lsort,idLsort] = sort(abs(Lambda))
7
8  IDfailure = idLsort(end)
9  Lfailure = Lambda(IDfailure)
10 Vecfail = Vecs(:,IDfailure)
11 FailDist = Vecfail
12
13 IDmetastable = idLsort(end-1)
14 Lmetastable = Lambda(IDmetastable)
15 Vecmeta = Vecs(:,IDmetastable)
16 Vecmeta_Length = sqrt(sum(Vecmeta.^2))
17
18 MetaDist = Vecmeta * (-1 / Vecmeta(1))
19 end
```