Model Discovery on Population and Video Datasets

Stephen Jonany[1]

***Abstract*— We apply model discovery techniques such as dynamic mode decomposition (DMD), time-delay embedding and Sparse Identification of Nonlinear Dynamics (SINDy) to summarize the dynamics encoded within the provided population and video datasets. We also compare the efficacy of these different approaches with various performance metrics such as KL divergence, AIC and BIC.**

I. Introduction and Overview

Many interesting real-world phenomena, such as population growth, can be represented as values that fluctuate over time. The problem of predicting future values given a limited historical sample is thus a very a salient problem to solve. Many model discovery approaches attempt to predict the summarized dynamics behind given historical samples, before using these simplified dynamics to extrapolate future time points.

Among the various model discovery approaches, we will consider dynamic mode decomposition (DMD)[1], time-delay embedding and Sparse Identification of Nonlinear Dynamics (SINDy)[2]. By using evaluation metrics such as KL divergence, AIC and BIC, these models will then be evaluated not only based on how well they fit the provided data, but also on their simplicity.

We will work with two datasets to test these approaches. The first dataset contains yearly hare and lynx populations. The second dataset is a video snippet of a Belousov-Zhabotinsky chemical oscillator movie.

II. Theoretical Background

A. Singular Value Decomposition

Singular value decomposition (SVD) is a dimensionality reduction technique that aims to decompose a matrix M into the product of three matrices $U\Sigma V^*$. The implementation that we are using is the economy SVD, where if M, the original matrix, is of dimension $m \times n$ and we have chosen a low rank $r \leq min(m, n)$, then both U and V are composed of orthonormal column vectors, with U of dimension $m \times r$, and V of dimension $n \times r$. Σ is a square diagonal matrix of dimension $r \times r$, and the diagonal values are often referred to as the singular values.

The singular values can be thought of as a measure of the importance of the corresponding column vectors of U and V in reconstructing the original matrix M. By selecting the top r values of Σ and the corresponding r columns of U and V, one can reconstruct an approximation of the matrix M. One useful metric for deciding on how many top singular values to keep is the total energy, as defined in Equation 1.

[1]Stephen is a Master's student in Applied Mathematics at the University of Washington. Email: sjonany@uw.edu.

$$Energy(r) = \frac{\sum_{i=1}^{r} \sigma_i^2}{\sum_{i=1}^{n} \sigma_i^2} \tag{1}$$

The energy metric measures how much of the original variance of the data is retained if we only keep the top U and V components corresponding to the top r singular values.

There are also some intuitive interpretations of U and V. The columns of U serve as the basis vectors of the column space of M. Likewise, the columns of V serve as the basis vectors of the row space of M. Thus, if M is composed of multiple time series, where each row is a single entity changing over time, say, the hare population across the years, then we can say that the hare population time series can be well approximated by a linear combination of the columns of V.

B. Dynamic Mode Decomposition (DMD)

DMD[1] is a model discovery technique for time-series prediction. There are two major assumptions of DMD, the first being that future states can be represented as a linear combination of historical states, the second being that the linear transformation that moves the states forward in time can be well-approximated in a lower dimension.

We first discuss the assumption that there is a linear relationship between the past and the future. Consider state vector $\boldsymbol{x_i}$, which represents an n-dimensional observation in a specific time t_i. We assume that we have m such observations from $\boldsymbol{x_1}$ to $\boldsymbol{x_m}$, where the time increment between each observation is kept constant. Let X be the collection of observations from times 1 to m stacked as column vectors, and X' be similarly defined, except that it corresponds to observations from times 2 to $m+1$. Then, DMD assumes there is a linear transformation A from X to X' as seen in Equation 2.

$$AX = X' \tag{2}$$

Note that the dimension of X and X' are $n \times m$, because there are m column vectors, each of length n. This means A is of the dimension $n \times n$.

We now discuss the second component of DMD: a low-rank approximation of the A matrix. Although Equation 2 can be solved as a normal regression problem, in practice, one tackles a lower-dimensional version for computational tractability. The steps are as follows.

1) **SVD on X**. We pick a low rank r following Section $II-A$, and obtain a low-rank approximation of X like so,

$$X \approx \tilde{U}\tilde{\Sigma}\tilde{V}^* \tag{3}$$

where $\tilde{U}$ is $n \times r$, $\tilde{\Sigma}$ is $r \times r$ and $\tilde{V}$ is $m \times r$.

2) **Reduce the dimension of** A. In this step, we reduce the dimension of A from $n \times n$ to $r \times r$. We achieve this by performing a similarity transform of A onto the columns of U. Since $AX = X'$, then $A = X'X^{\dagger} \approx X'(\tilde{U}\tilde{\Sigma}\tilde{V}^{*})^{\dagger} = X'\tilde{V}\tilde{\Sigma}^{-1}\tilde{U}^{*}$. We then use this to obtain the similarity transformed, lower-rank $\tilde{A}$ as follows.

$$\tilde{A} = \tilde{U}^{*}A\tilde{U} = \tilde{U}^{*}X'\tilde{V}\tilde{\Sigma}^{-1} \tag{4}$$

A useful property is that the similarity transformation preserves the nonzero eigenvalues of A. Thus, we can avoid working with the high-dimensional A and instead focus on the lower-dimensional $\tilde{A}$.

3) **Eigendecomposition of** $\tilde{A}$. Through eigendecomposition, we achieve the following:

$$\tilde{A}W = W\Lambda \tag{5}$$

where the columns of W are the eigenvectors of $\tilde{A}$, and the Λ is a diagonal matrix consisting of the eigenvalues of $\tilde{A}$, which are also the eigenvalues of the original A. Thus, Λ contains the DMD eigenvalues themselves.

4) **DMD mode** Φ. We have the eigenvalues, and now we will find the DMD modes. Tu et al. [3] calculates the high-dimensional DMD modes like so.

$$\Phi = X'\tilde{V}\tilde{\Sigma}^{-1}W \tag{6}$$

Now that we have the DMD mode Φ, of dimension $n \times r$, and the diagonal matrix containing the eigenvalues, Λ, of dimension $r \times r$, the we can obtain the low-rank projection of the initial condition $\boldsymbol{x_1}$, referred to as $\boldsymbol{b}$, like so.

$$\boldsymbol{b} = \Phi^{\dagger}\boldsymbol{x_1} \tag{7}$$

Finally, we are able to predict future states x_k like so.

$$\boldsymbol{x_k} = \Phi\Lambda^{k-1}\boldsymbol{b} \tag{8}$$

Equation 7 is reminiscent of exponentiating an eigendecomposed matrix, where one simply has to exponentiate the eigenvalue matrix, and leave the eigenvector matrix untouched. In this case, we are applying an approximation of A, the matrix that supposedly carries our state vectors forward in time, $k-1$ times, hence bringing $\boldsymbol{x_1}$ to a predicted value of $\boldsymbol{x_k}$. Furthermore, we have done so through low-rank approximations, allowing us not to deal with the high-dimensional A matrix.

In summary, DMD enables us to summarize the dynamics that move state variables forward in time in a given set of time series data, and allows us to predict future states.

C. Time-delay Embedding

A major assumption of DMD is that Equation 2 mostly holds true. That is, we assume that the future states can be expressed as a linear combination of the historical states. For cases where this assumption is majorly violated, a solution would be to project the state variables into a different functional space. One such projection approach is time-delay embedding.

In time-delay embedding, we increase the state space dimension by k-fold, where k indicates how many future state variables one wants to include in every column of the transformed matrix. If one starts with an original matrix M of dimension $n \times m$, where each n-dimensional column vector is a measurement in time, then with k embedding, the higher dimensional matrix M' is of dimension $nk \times (m-k+1)$. The first column will then contain $x_1, ..., x_k$ stacked up row-by-row, and the second column will contain $x_2, ..., x_{k+1}$, going to the last column, which contains $x_{m-k+1}, ..., x_m$.

As we will see later on, time-delay embedding could help improve the efficacy of the DMD approach.

D. Sparse Identification of Nonlinear Dynamical Systems (SINDy)

SINDy [2] is a model discovery approach that one can use to model systems believed to be produced by ordinary different equations (ODEs). The problem setup that is relevant for our given datasets is such that we are provided with observations of state variables $\boldsymbol{x(t_1)}, ..., \boldsymbol{x(t_m)}$, corresponding to measurements taken at regular intervals in time, each state variable is of dimension n . Our goal is to find the first order ODE $\frac{d\boldsymbol{x}}{dt} = f(\boldsymbol{x}) = f(x_1, x_2, ..., x_n)$.

We make a simplifying assumption that the true ODE is well approximated by a linear combination of a library of functions of the state vector. That is, we assume that we have already chosen a list of L functions $f_1(\boldsymbol{x}), f_2(\boldsymbol{x}), ..., f_L(\boldsymbol{x})$. Let us focus on just the recovery of the ODE for the first out of the n state variables, x_1. The ODE can be expressed like so:

$$\frac{dx_1}{dt} = \sum_{i=1}^{L} c_i f_i(x_1, x_2, ..., x_n) \tag{9}$$

where c_i's, the weights applied to the library functions, are what we wish to discover.

Given this assumption, and given the data matrix X of dimension $n \times m$, composed of $\boldsymbol{x(t_1)}, ..., \boldsymbol{x(t_m)}$ arranged along the column vectors, and we can obtain the c_i's with the following steps.

1) **Obtain** $\frac{dx_1}{dt}$ **labels,** $\boldsymbol{b}$. From the provided matrix X, we would like to obtain the values of $\frac{dx_1(t)}{dt}$ for $t_2, t_3, ..., t_{m-1}$. We approximate these values using centered finite difference like so:

$$\frac{dx_1(t_i)}{dt} \approx \frac{x_1(t_{i+1}) - x_1(t_{i-1})}{2d} \tag{10}$$

, where d is the time increment between each time snapshot. Note that we throw out t_1 and t_m, and so are left with $m-2$ labels. Let this $(m-2) \times 1$ column vector be $\boldsymbol{b}$.

2) **Create the function library matrix** A. We next generate the matrix A, of dimension $(m-2) \times L$, where each column i of this matrix is taking the one out of the L functions, f_i, and applying it to each of

the $m-2$ column in X (without the first and last time point), a measurement in a single time. That is, $A_{ij} = f_j(\boldsymbol{x}(\boldsymbol{t_{i+1}}))$.

3) **Solve for function weights**, **c**. The formulation provided in Equation 9 means that to find the weight vector $\boldsymbol{c}$, of dimension $L \times 1$, we just have to solve the familiar problem $A\boldsymbol{c} = \boldsymbol{b}$. We will use a sparse regression method, Lasso[4], to find the best $\boldsymbol{c}$ that balances data fit and overfitting. This will allow us to subselect only a few of the library functions, keeping our final predicted ODEs more compact. For a more thorough discussion on Lasso, please refer to the previous writeup on the MNIST dataset.

We have discussed how to find the best set of function library weights to describe $\frac{dx_1}{dt}$. One would then repeat the above steps $n-1$ more times to obtain the ODEs for the other state variables. All in all, one would end up with $L \times n$ coefficients to describe all n ODEs.

With the ODEs in hand, one can employ various ODE solvers such Runge–Kutta methods in order to predict any future state $\boldsymbol{x(t)}$.

E. Lotka-Volterra

Lotka-Volterra equations model the populations of predator and prey with a set of ODEs like so.

$$\begin{aligned} \frac{dx}{dt} &= (b - py)x \\ \frac{dy}{dt} &= (rx - d)y \end{aligned} \tag{11}$$

x is the prey population, y the predator population, and b, p, r, d are the parameters. The intuition is that when all the parameters are positive, the prey will reproduce to grow, but is inhibited by the predator population, whereas an isolated predator population would decrease over time due to competition, but can grow given more prey to eat.

We will obtain the best parameters that fit the data by following the steps outlined in Section II-D. The differences are as follows:

1) Instead of Lasso, we will use the ordinary least squares solution. For interested readers, please refer to the previous writeup on the MNIST dataset for more details.
2) For $\frac{dx}{dt}$, we will use the library functions $f_1(x, y) = x$ and $f_2(x, y) = -xy$. This means that the coefficient for f_1 gives you b, and the coefficient for f_2 gives you p.
3) For $\frac{dy}{dt}$, we will use the library functions $g_1(x, y) = -y$ and $g_2(x, y) = xy$. This means that the coefficient for g_1 gives you d, and the coefficient for g_2 gives you r.

F. Time Series to Probability Distributions

In the following sections, we will be applying metrics that are well-defined for probability distributions, to pairs of time series, the hare and lynx populations across the years. We now highlight some assumptions to be made before we can arrive at a meaningful conversion from pairs of time series to probability distributions.

We assume that each hare and lynx population in a single year is independent of those of the other years, and is modelled by a joint distribution $P(x, y)$, where x and y are the hare and lynx populations respectively. We also further assume that the sample space consists of pairs of integers, each from 0 to 200. We find that 200 is an upper bound that all the model predictions are respecting.

Given a pair of time series, to obtain an estimate of $P(x, y)$, we go through the x, y observation from each year, round them down to integers, and tally up the observations. With the counts of all the possible x, y observations, the joint distribution $P(x, y)$ is simply the normalized tally. That is,

$$P(x, y) = \frac{Count(x, y)}{\sum_{x'=0}^{200} \sum_{y'=0}^{200} Count(x', y')} \tag{12}$$

To deal with the problem of the sparse dataset, we perform this tallying process on the interpolated data, which will give us more data points that the original data. Furthermore, we also apply Laplace smoothing with a pseudocount of 0.001, so that unobserved pairs do not have 0 probabilities.

G. Kullback–Leibler (KL) divergence

The KL divergence is a metric for quantifying the distance between two different probability distributions. We consider the discrete case, where the sample space is S. Then, the KL divergence for two probability distributions $P(s)$ and $Q(s)$ is defined like so.

$$D(P, Q) = \sum_{s \in S} P(s) \log(\frac{P(s)}{Q(s)}) \tag{13}$$

For the hare and lynx population prediction, we convert the gold and model time series pairs to probability distributions as per described in II-F. Then, we let P correspond to the gold distribution, and Q the model distribution. Finally, we will be able to use this measure to compare how well the different models perform in fitting the gold dataset.

H. AIC and BIC

Akaike information criterion (AIC) and Bayesian information criterion (BIC) allow us to quantify not only how well a statistical model fits the data, but also takes the model complexity into account. The intuition is that the simpler model that can explain the data well enough should have low AIC and BIC scores.

The definitions are as follows.

$$AIC = 2k - 2\ln(\tilde{L}) \tag{14}$$

$$BIC = k\ln(n) - 2\ln(\tilde{L}) \tag{15}$$

k is the number of model parameters that are optimized against the data, and can be thought of as the measure of model complexity. n is the number of data points the model is being fit to. $\tilde{L}$ is the likelihood of the data given the model.

We now discuss what the log likelihood term, $\ln(\tilde{L})$, means in the context of the hare and lynx population dataset. We start with the time-independent joint probability distribution model, $P(x, y)$ as defined in Section II-F. Given

the gold population snapshots $(x_1, y_1), (x_2, y_2), ..., (x_m, y_m)$ over m time points, then the log likelihood term can be computed by exploiting the time-independence assumption.

$$\ln(\tilde{L}) = \ln(\prod_{i=1}^{m} P(x_i, y_i)) = \sum_{i=1}^{m} \ln(P(x_i, y_i)) \tag{16}$$

III. Algorithm Implementation and Development

All the codes are written in Python. The following Python libraries are used.

- SciPy is used for implementations of linear regression and Lasso, cubic spline interpolation, as well as the Explicit Runge-Kutta method of order 5[5] for ODE simulation.
- NumPy is used for efficient 2D matrix manipulations.
- Matplotlib is used to generate all the plots.

The key steps of algorithms such as DMD have been discussed extensively in the previous section, and will not be repeated. We invite interested readers to refer to the code appendices for more detail.

IV. Computational Results

A. Population Dataset Overview

The population dataset consists of 30 observations of hare and lynx populations from year 1845 to year 1903, in increments of 2 years. The hare population ranges from 7 to 150, and the lynx population ranges from 6 to 80.

Because of the minimal number of observations, we applied cubic spline interpolation with time increments of 0.1 year to obtain 600 observations. The interpolation result can be seen in Figure 1.

We chose not to perform cross validation (as described in the MNIST writeup) due to the small number of samples. Instead, for the model discovery approaches described in the following sections, we always train on the first half (300 observations), and test on the last half.

Finally, the data is arranged in a matrix such that there are two rows, corresponding to the hare and lynx population, and there are m columns, corresponding to the m time snapshots.

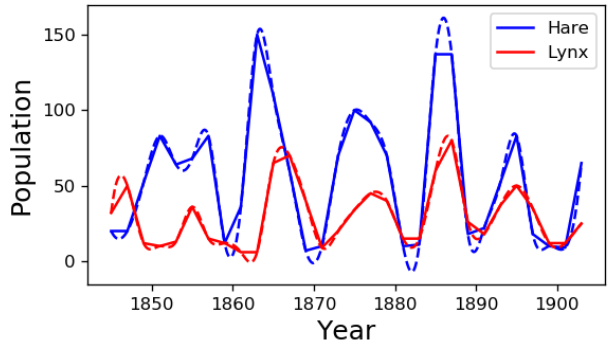

Fig. 1: The original hare and lynx population dataset (in bold), along with the cubic spline interpolation (dotted).

B. DMD Results

We train a DMD model with rank 2 on the first 300 observations, and use the model to predict the population across all the years, following the steps outlined in Section II-B. As seen in Figure 2, the model performed poorly. Not only do we have a poor fit because of the grossly mismatched period, but we also have negative population predictions, which are biologically implausible.

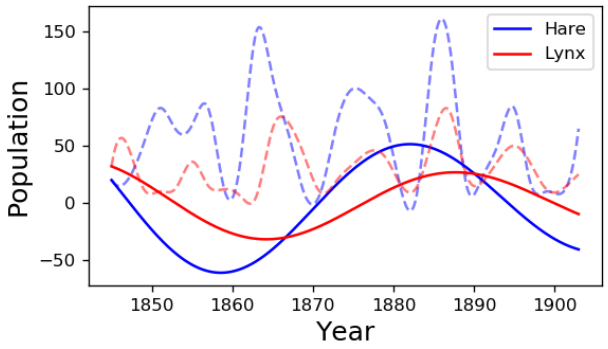

Fig. 2: The interpolated dataset (in dots), along with the DMD prediction (in bold).

However, there are some redeeming qualities of this prediction. First, it manages to produce an oscillatory prediction. When we inspect the DMD eigenvalues, we see that both eigenvalues have real parts that are very close to 1, allowing the predictions to not explode or decay too quickly, with small complex components that allow the predictions to oscillate in time. Second, even with just half the dataset used for training, the model captures the fact that the prey population should rise before the predator population begins to rise.

C. Time-delay Embedding SVD Analysis

We then apply time-delay embedding to the data, as described in Section II-C, to see if the data can be reprojected into a new space where DMD would work better.

We found that using 6 as the embedding degree produced decent results. On applying SVD to the 6-degree time-delay embedded dataset, we found that we need at least 2 modes to capture 99% of the energy, and 3 modes to capture 99.9% of the energy, as shown in Figure 3. Thus, we believe that there are likely at least 2 latent variables behind the dynamics.

The population modes can be seen in Figure 4. As discussed in Section II-A, these are obtained from the columns of V, which serve as good basis vectors of the rows of the original data. As a reminder, the original rows of the data correspond to the interpolated hare and lynx population time series. The first two modes seem to be phase-offset in nature, with the peaks of the first mode lagging behind the peaks of the second mode. This perhaps captures the phase offset between predator and prey that is also present in the original dataset.

The time modes can be seen in Figure 5. This is harder to interpret because due to time-delay embedding, the hare

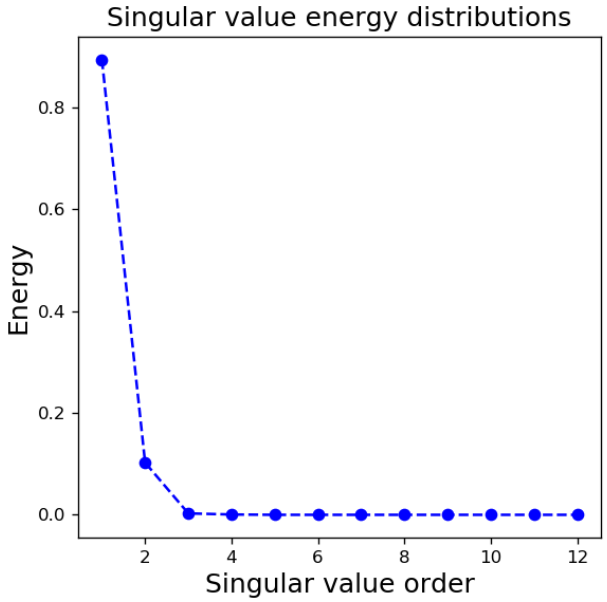

Fig. 3: Energy distribution of singular values of time-delay embedded data.

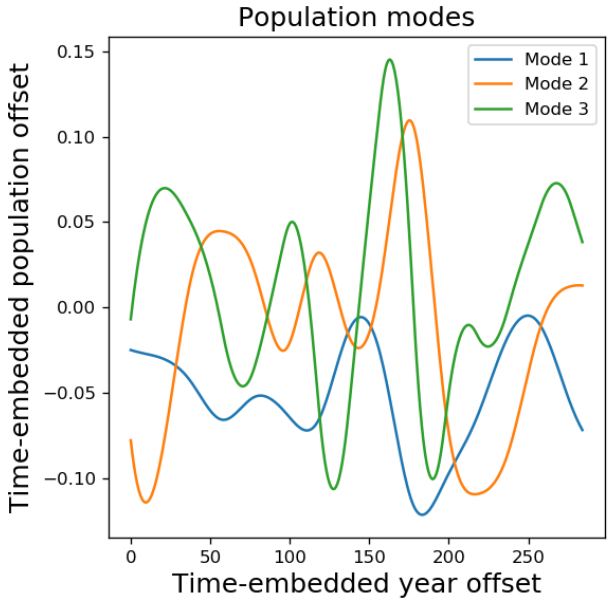

Fig. 4: Population modes of the time-delay embedded data.

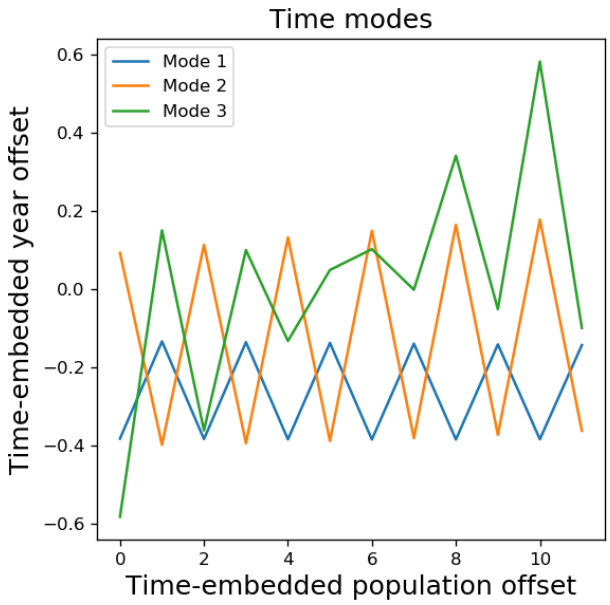

Fig. 5: Time modes of the time-delay embedded data.

and lynx populations of each year are interleaved along the columns. The first two modes seem to again be phase-offset. The third mode that seems to increase over time perhaps captures a global increasing or decreasing trend of the interpolated data.

D. DMD with Time-delay Embedding Results

From the previous analysis, we decided to stick to 6-degree embedding, and chose 3 for the DMD rank, since the 3 top singular values captures more than 99.9% of the energy. We then apply DMD on this time-delay embedded dataset. Note that to obtain the predicted hare and lynx population, we extract just the first two rows from the time-delayed embedded matrix.

As seen in Figure 6, the result is much better than the regular DMD, with the prediction obtaining a frequency that is a much better fit to the interpolated data, with the phase offset quality still being retained. Furthermore, this prediction also captures the fact that the hare population has a larger amplitude than that of the lynx. This perhaps indicates that the time-delay embedded space allows the linearity assumptions of DMD to hold better than in the original interpolated data.

E. Lotka-Volterra Results

We obtain the best-fit parameter values for the Lotka-Volterra model, as described in Section II-E. As seen in Figure 7, the result seems reasonable. The amplitudes of the predicted hare and lynx populations match those of the interpolated data. Furthermore, the phase offset characteristic where the peaks of hare population precede the peaks of lynx is also captured. The main downside is that the Lotka-Volterra model seems to capture a single peak amplitude, and fails to capture the lower peaks which we see in the original data.

Letting x be the hare population, and y be the lynx population, the recovered equations, up to three decimal places, are as follows.

$$\begin{aligned} \frac{dx}{dt} &= 0.217x - 0.009xy \\ \frac{dy}{dt} &= -0.381y + 0.006xy \end{aligned} \tag{17}$$

It is interesting to see that the non-linear coefficients are as small as 0.006. This informs us that for our SINDy

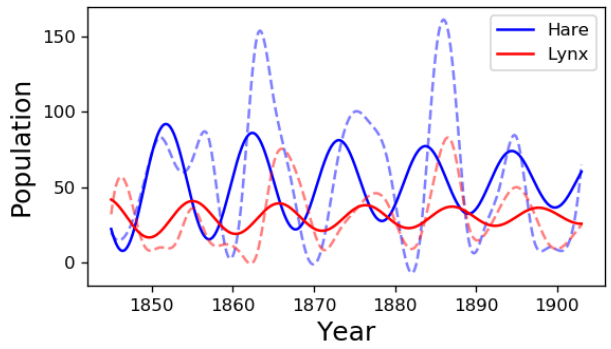

Fig. 6: The interpolated dataset (in dots), along with the 6-degree time-delay embedded DMD prediction (in bold).

approaches, we should not carelessly discard small values that capture important non-linear components. In this case, if we had discarded the small non-linear components, we would be left with two exponential solutions without oscillations.

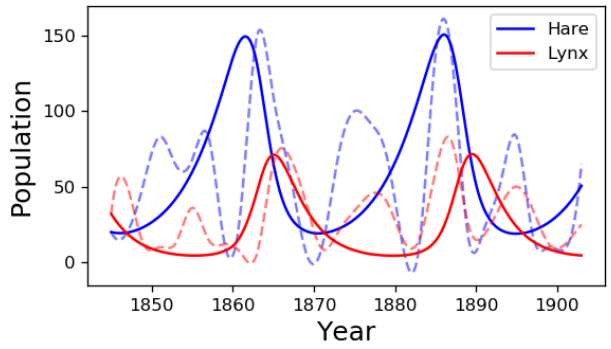

Fig. 7: The interpolated dataset (in dots), along with the Lotka-Volterra prediction (in bold).

F. SINDy Results

We perform SINDy on the first half of the interpolated dataset to obtain different ODE models, as per Section II-D. We have tried three different function libraries, corresponding to polynomials of degrees one to three. That is, letting x and y be the hare and lynx population respectively, then the first function library consists of $[x, y]$ and the second function library consists of $[x^2, xy, y^2, x, y]$. For each SINDy run, we obtain the coefficients by running multiple Lasso iterations, with regularization weight α ranging from 0 to 10 in increments of 0.1. We only report the results that give the lowest Frobenius norm of the deviation between the predicted pairs of time series and the interpolated time series for the first half of the dataset. The results can be seen in Table I.

We discuss the results from function libraries of polynomials of degree 1, 2, and 3, which we will refer to as Poly-1, Poly-2, and Poly-3. The predictions of Poly-3 seems to just be a regression of the mean, with the prediction piercing through the center of the original dataset. The predictions of Poly-2 are very poor, with the hare population going to negative values. Poly-1 seems to be the only model that captures the oscillation in the data, but unfortunately also predicts negative population values. We also note that across all the models, heavier weights are placed onto the lower degree terms, and much smaller weights are placed on the higher degree terms, perhaps due to Lasso discouraging the model from recruiting the complex terms.

Given this discouraging result, we create a final model Poly-3-trimmed by selecting the 4 highest weighted terms from Poly-3: $[x, x^2, xy, y^2]$, and running SINDy on this smaller function library. We were able to see a marked improvement, where the prediction has a weak oscillation, and also stays within the centers of the true timeseries values.

G. KL Divergence Results

All the previous models have been trained only on the first half of the interpolated data. We now compare their performances on the unseen second half, by using KL divergence as discussed in Section II-G. We will discard DMD, Poly-1 and Poly-2, since they produce negative population predictions, which are biologically implausible.

As seen in Table III, Time-delay embedded DMD seems to perform the best, with the smallest KL divergence of 6.04. Looking at Figure 6, we see that this is the only model that captures the high frequency of the original dataset.

H. AIC and BIC Results

The drawback from the KL divergence metric is that we are not taking into account model complexity. We also report the AIC and BIC results as described in Section II-H. Note that AIC and BIC is meant to measure the tradeoff between model complexity and data fit on the same dataset, and so we calculate these values solely based on the first half of the dataset. We will also only consider the top 3 models that performed best as measured by KL divergence: Time-delay embedded DMD, Lotka-Volterra and Poly-3-trimmed.

The number of parameters, k, for each model, are as follows.

1) Time-delay embedded DMD has 39 parameters in total. This is because the DMD modes Φ is shaped $n \times r$, which is 12×3. Note that the 12 comes from the 6-embedding of the hare and lynx population, and 3 is the number of modes we decided to keep. While the DMD modes give us 36 numbers, the 3 DMD eigenvalues bring us to a total of 39.
2) Lotka-Volterra has 4 parameters in total, corresponding to the 2 coefficients for $\frac{dx}{dt}$, and another 2 coefficients for $\frac{dy}{dt}$.
3) Poly-3-trimmed has 8 parameters in total, since we only have 4 library functions, and two state variables.

Here we see that the Lotka-Volterra model has the lowest AIC and BIC scores. We now see that even though according to the KL divergence metric, the time-delay embedded DMD was deemed the best model, AIC and BIC metrics successfully take into account the fact that the Lotka-Volterra

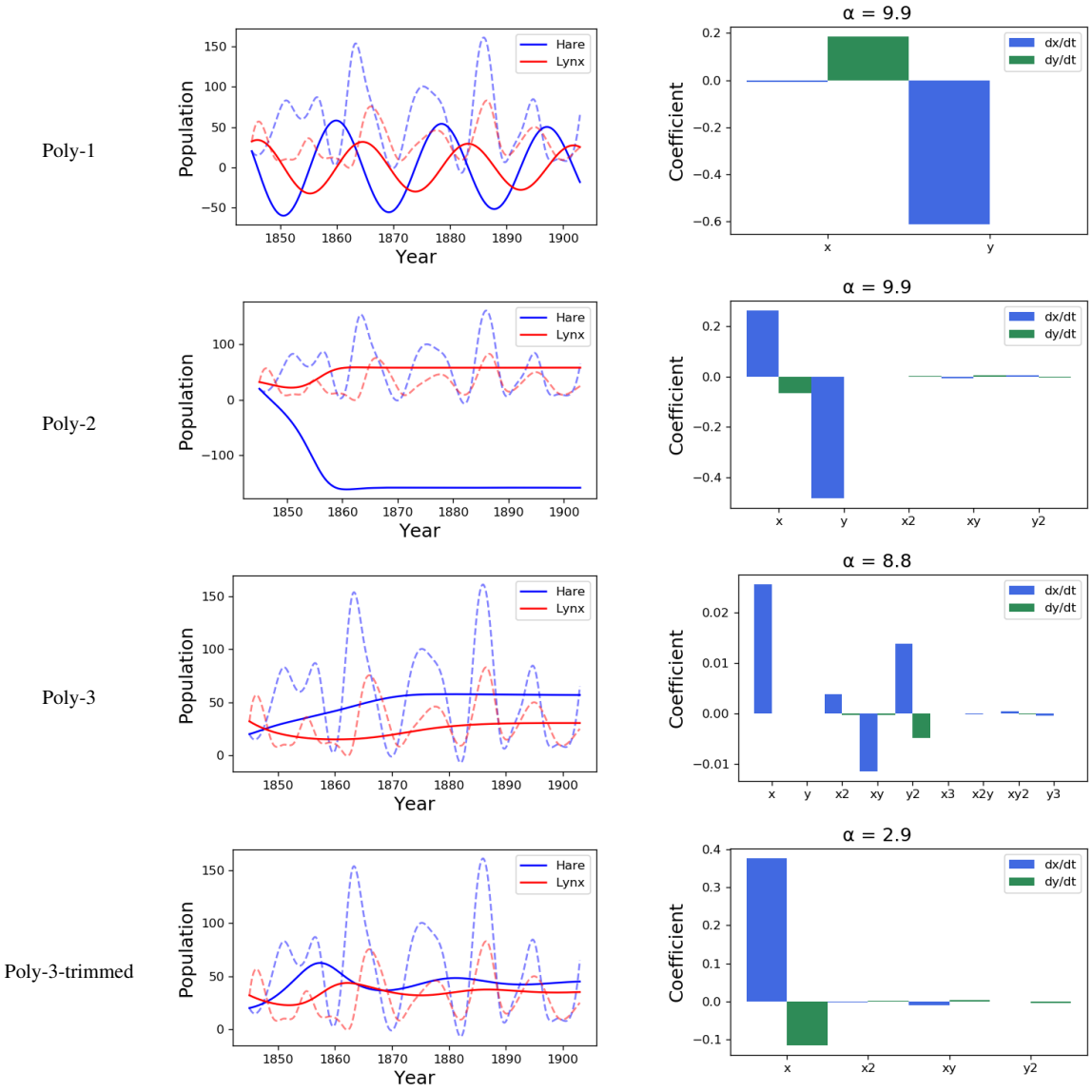

TABLE I: SINDy results on library functions with polynomials of degree 1,2 and 3. The last row is the trimmed degree 3 library as described in Section IV-F. The left column contains the interpolated dataset (in dots), along with the SINDy predictions (in bold). The right column are the weights assigned to the library functions, as obtained by Lasso with the regularization parameter α stated on the title of each figure.

TABLE II: KL Divergence on Test Set (rounded to 2 decimal places)

Model	KL Divergence
Time-delay embedded DMD	**6.04**
Lotka-Volterra	6.21
Poly-3	6.28
Poly-3-trimmed	6.18

TABLE III: AIC and BIC of Top 3 models (rounded to the closest integer)

Model	AIC	BIC
Time-delay embedded DMD	7282	7425
Lotka-Volterra	**7266**	**7280**
Poly-3-trimmed	7324	7354

model has far fewer free parameters to work with, and is hence the better model at balancing data fit and model complexity.

I. Time-delayed Embedded DMD on Chemical Oscillator Movie

We now move away from the population dataset, and instead focus on the chemical oscillator movie dataset. We focus only on the 40 by 40 square patch as shown in Figure 10. We will also focus only on the last 50 frames of the video. This setup allows us to focus on the prediction of

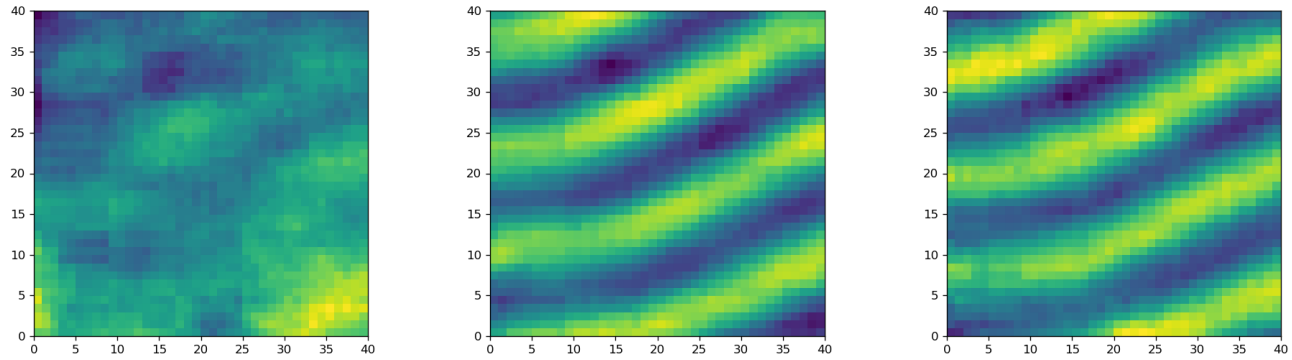

Fig. 8: The top 3 spatial SVD modes of the training video. On the left is the first mode, on the center is the second mode, and on the right is the third mode.

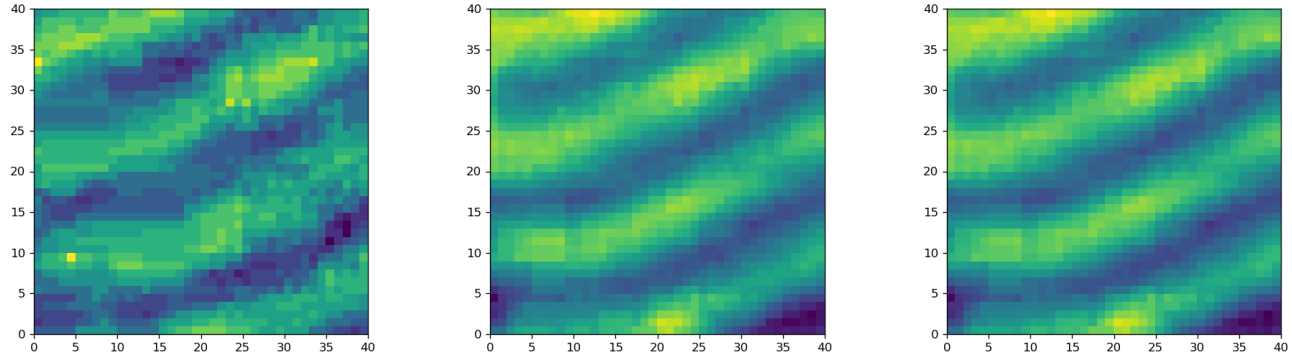

Fig. 9: On the left is the true 50th frame, on the center is the prediction by DMD, and on the right is the prediction by time-delayed embedded DMD.

waves that are propagating diagonally to the bottom right corner.

Our prediction task is to use DMD and time-delay embedded DMD on the first 25 frames, and predict the 50th frame. This is to test if DMD can successfully capture wave propagation in a video data.

We first preprocess the video data into a 2D matrix of size $n \times m$, where n corresponds to the number of pixels and m corresponds to the frame number. We flatten each 40×40 2D image into 1600×1 vectors by taking each transposed image row and concatenating them to form a long column. This gives us the training data matrix of dimension 1600×25.

We apply SVD to this training data matrix. As seen in Figure 11, even the first mode already captures 99.9% of the energy.

We inspect the first 3 spatial modes (the first 3 column vectors of U from SVD), and de-flatten these vectors into a 2D image. As seen from Figure 8, the first mode seems to just capture the static background of the video, whereas the other two modes, which are waves with a phase offset, seems to capture the phenomena of the travelling wave. This was further validated by performing DMD with varying number of modes. With just one or two modes, the predicted video will just be a static image. It is not until we include the third mode that the predicted video starts to result in a propagating wave.

This intuitively makes sense, since DMD tries to reconstruct the predicted frame through a linear combination of these spatial modes. To construct a wave of any phase offset, one would need a linear combination of at least two waves with a non-zero phase offset.

With the above analysis, we apply DMD with rank 3, and time-delayed embedded DMD with rank 3 and embedding degree of 2, onto the first 25 training video frames, and use it to predict the 50th frame. The results are show on Figure 9. We can see that both DMD and time-delayed embedded DMD were able to produce pretty good predictions, although a lot smoother than the actual image. Time-delayed embedded DMD has a slight edge over regular DMD because for the darker patches, it is able to produce darker pixels similar to the original image.

Overall, this was a very pleasant surprise. DMD is able to reproduce the travelling wave video, even with as little as 3 modes.

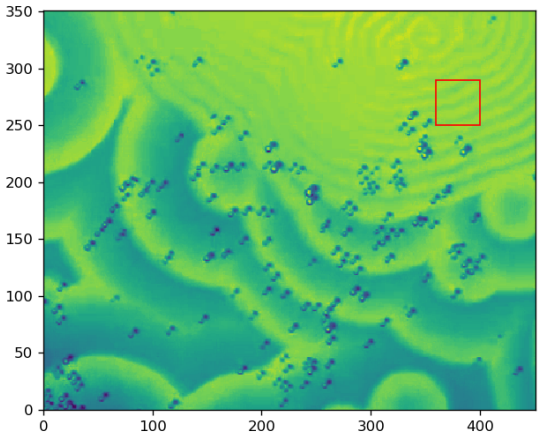

Fig. 10: A frame from the chemical oscillator movie, with the portion to be analyzed outlined by the red square patch.

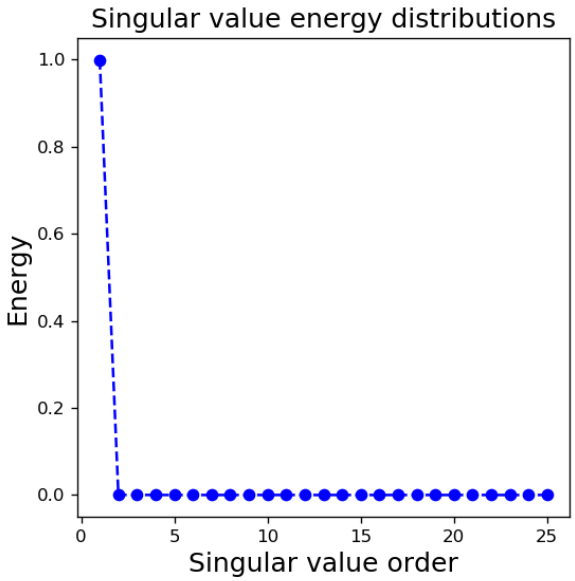

Fig. 11: Energy distribution of singular values of the training video data.

V. Summary and Conclusions

We have applied various model discovery techniques such as DMD, time-delayed embedding and SINDy on population and video datasets, and compared their performances using evaluation metrics such as KL divergence, AIC and BIC. We showed that these models were able to capture the core dynamics behind the noisy datasets, and were able to make decent predictions of future states that they have not been trained on.

References

[1] P. J. SCHMID, "Dynamic mode decomposition of numerical and experimental data," *Journal of Fluid Mechanics*, vol. 656, p. 5–28, 2010.

[2] S. L. Brunton, J. L. Proctor, and J. N. Kutz, "Discovering governing equations from data by sparse identification of nonlinear dynamical systems," *Proceedings of the national academy of sciences*, vol. 113, no. 15, pp. 3932–3937, 2016.

[3] J. H. Tu, C. W. Rowley, D. M. Luchtenburg, S. L. Brunton, and J. N. Kutz, "On dynamic mode decomposition: Theory and applications," *arXiv preprint arXiv:1312.0041*, 2013.

[4] R. Tibshirani, "Regression shrinkage and selection via the lasso," *JOURNAL OF THE ROYAL STATISTICAL SOCIETY, SERIES B*, vol. 58, pp. 267–288, 1994.

[5] J. R. Dormand and P. J. Prince, "A family of embedded runge-kutta formulae," *Journal of computational and applied mathematics*, vol. 6, no. 1, pp. 19–26, 1980.

APPENDIX A: KEY PYTHON FUNCTIONS

Dynamic Mode Decomposition

```
def dmd(data, r):
  """
  Perform dynamic mode decomposition on the data matrix.
  Parameters:
  - data (n x m+1 matrix) Rows are spatial locations and columns are time points.
    That is, there are n spatial locations and m time points.
  - r (int) The low-rank to perform DMD in.
  Returns:
  - Phi (n x r matrix) The DMD (spatial) modes.
  - Lambda (r x r matrix) The diagonal eigenvalue matrix.
  - b (r x 1 matrix). The low-rank initial condition
  """
  X0 = data[:, :-1]
  X1 = data[:, 1:]

  # X0 is n x m
  # U is n x n
  # S is n x n, a diagonal matrix of the singular values.
  # Vh is m x m
  U, singular_vals, Vh = np.linalg.svd(X0)
  S = np.diag(singular_vals)
  V = Vh.conj().T
  # Ur is n x r
  Ur = U[:,:r]
  # Sr is r x r
  Sr = S[:r, :r]
  # Vr is m x r
  Vr = V[:,:r]
  Sr_inv = np.linalg.inv(Sr)
  # Atilde is r x r
  Atilde = Ur.conj().T @ X1 @ Vr @ Sr_inv

  # Both W and Lambda are r x r
  eigenvalues, W = np.linalg.eig(Atilde)
  Lambda = np.diag(eigenvalues)

  # Phi is n x r, the DMD spatial models
  Phi = X1 @ (Vr @ Sr_inv) @ W
  initial_condition = X0[:, 0]
  b = np.linalg.pinv(Phi) @ initial_condition

  return Phi, Lambda
```

Future Prediction from DMD Modes and Eigenvalues

```
def get_xt_from_dmd(num_time, Phi, Lambda, b):
  """
  Predict x_t given the DMD results.
  Parameters:
  - num_time. Number of time points to predict states for.
    We will predict x1, ..., x_num_time, where x1 is the predicted initial condition.
  - Phi, Lambda, b - See dmd()
  Return:
  - A (N x num_time) matrix, where N is the number of spatial coordinates.
    mat[:1] will give you the approximation to x1, the initial condition.
```

```
  """
  n = Phi.shape[0]
  r = Phi.shape[1]
  result = np.zeros((n, num_time))

  # x(t) = Phi Lambda^{t-1} b
  phi_times_lambda = Phi
  for t in range(num_time):
   # Sometimes the eigenvalues are complex. We only care about the real part.
    result[:, t] = np.real(phi_times_lambda @ b)
    phi_times_lambda = phi_times_lambda @ Lambda
  return result
```

Time-delayed Embedding

```
def time_delay_embed(data, k):
  """
  Create a time delay embedding of the data matrix, with k offsets.
  Parameters:
  - data (n x m matrix) Rows are spatial locations and columns are time points.
  - k (int) The number of embeddings to make. k = 1 means no embedding.
  Return:
  - matrix with (k*n rows, m-k+1 columns)
    The first n rows are the first k columns of data ~ data[all_rows, t = 1 -> k]
    The following n rows are the second k columns of data ~ data[all_rows, t = 2 -> k + 1]
    And so on
  """
  n = data.shape[0]
  m = data.shape[1]
  result = np.zeros((n*k, m-k+1))
  for delay_counter in range(k):
    start_row = n * delay_counter
    end_row_exclusive = start_row + n
    start_col = delay_counter
    end_col_exclusive = delay_counter + m-k+1
    result[start_row:end_row_exclusive, :] = data[:, start_col:end_col_exclusive]
  return result
```

SVD Energy and Mode Plots

```
def plot_energy_from_singular_values(data):
  U, singular_vals, Vh = np.linalg.svd(data)
  squared_singular_vals = np.power(singular_vals, 2)
  energy_values = squared_singular_vals / np.sum(squared_singular_vals)
  fig, ax = plt.subplots(figsize = (5, 5))
  ax.plot(range(1 , len(energy_values) + 1), energy_values, linestyle='--', marker='o', color=
  ax.set_xlabel("Singular value order")
  ax.set_ylabel("Energy")
  ax.xaxis.label.set_fontsize(15)
  ax.yaxis.label.set_fontsize(15)
  ax.set_title("Singular value energy distributions", fontdict={'fontsize': 15})
  print("Percent energy covered: %s" % (np.cumsum(energy_values)))
  return fig

def plot_population_modes(data, top_k):
  U, singular_vals, Vh = np.linalg.svd(data)
  fig1, ax = plt.subplots(figsize = (5, 5))
  for k in range(top_k):
    mode = U[:,k]
```

```
    ax.plot(range(len(mode)), mode, label = f"Mode {k + 1}")
    ax.set_xlabel("Time-embedded population offset")
    ax.set_ylabel("Time-embedded year offset")
  ax.xaxis.label.set_fontsize(15)
  ax.yaxis.label.set_fontsize(15)
  ax.set_title("Time modes", fontdict={'fontsize': 15})
  ax.legend()

  fig2, ax = plt.subplots(figsize = (5, 5))
  for k in range(top_k):
    mode = Vh[k, :]
    ax.plot(range(len(mode)), mode, label = f"Mode {k + 1}")
    ax.set_xlabel("Time-embedded year offset")
    ax.set_ylabel("Time-embedded population offset")
  ax.xaxis.label.set_fontsize(15)
  ax.yaxis.label.set_fontsize(15)
  ax.set_title("Population modes", fontdict={'fontsize': 15})
  ax.legend()
  return fig1, fig2
```

Centered Finite Difference

```
def get_dx_dts(xts):
  """
  Use centered finite difference to approximate dx/dt, given x(t)'s

  Parameters:
  - xts (n x 1 column vector). The x(t) over time.
  Return:
  - dx_dts ((n-2) x 1 column vector). The time in front and end are removed.
  """
  dt = util.NEW_DT
  dx_dts = np.zeros(len(xts) - 2)
  for i in range(len(dx_dts)):
    dx_dts[i] = (xts[i+2] - xts[i]) / (2.0 * dt)
  return dx_dts
```

Function Library Matrix for SINDy

```
def gen_library_matrix(measurements, f_lib):
  """
  Generate a matrix of library functions evaluated against the measurements.
  Parameters:
  - measurements (n x m matrix). n is the dimension of each state variable,
    m is the number of timepoints the measurements were taken.
  - f_lib: list((name, fun(x_vec)). A list of 'L' library functions,
    where each function accepts a column from 'measurements' matrix, that is,
    a single state variable vector, snapshotted in time.
  Return:
    m x L matrix, where each column is a single library function evaluated across
    the m measurements.
  """
  n = measurements.shape[0]
  m = measurements.shape[1]
  L = len(f_lib)
  library_mat = np.zeros((m, L))

  for i, name_fun in enumerate(f_lib):
    fun = name_fun[1]
```

```
    # Apply fun to each column, which represents x(t)
    library_mat[:,i] = np.apply_along_axis(fun, 0, measurements)
  return library_mat
```

Lotka-Volterra Parameter Fitting

```
def get_best_fit_lotka_volterra_params(data):
  """
  Use linear regression to obtain the parameters for the ODEs below that best fit the data.
  dx/dt = (b-py)x
  dy/dt = (rx-d)y

  Parameters:
  - data (n x m matrix) Rows are spatial locations and columns are time points.

  Return:
  - b, p, r, d, the ODE parameters.
  """
  # Generate the RHS of Ax = b
  dx_dts = get_dx_dts(data[0,:])
  dy_dts = get_dx_dts(data[1,:])

  # Generate the LHS, A of Ax = b
  # Remove the first and last time samples, because finite difference removes them too.
  data_without_ends = data[:, 1:-1]
  # For dx/dt, our library is [x, -xy]
  # The shape is 28 (time) x 2 (number of library functions)
  x_library = gen_library_matrix(data_without_ends, [
    ("x", lambda x: x[0]),
    ("-xy", lambda x: -x[0] * x[1])
  ])
  # For dy/dt, our library is [-y, xy]
  y_library = gen_library_matrix(data_without_ends, [
    ("-y", lambda x: -x[1]),
    ("xy", lambda x: x[0] * x[1])
  ])
  model = linear_model.LinearRegression()

  model.fit(x_library, dx_dts)
  x_library_coefs = model.coef_
  b = x_library_coefs[0]
  p = x_library_coefs[1]
  model.fit(y_library, dy_dts)
  y_library_coefs = model.coef_
  d = y_library_coefs[0]
  r = y_library_coefs[1]
  return b, p, r, d
```

SINDy Coefficient Finding

```
def get_sindy_params(data, regression_model, f_lib):
  """
  Use sparse regression to obtain the parameters for the ODEs below that best fit the data.
  dx/dt, and dy/dt are linear combinations of (x, y, x2, xy, y, x3, x2y, xy2, y3)
  Note that there are 9 terms.
  That is, we are doing SINDy, but with library functions of polynomials up to degree 3.

  Parameters:
  - data (n x m matrix) Rows are spatial locations and columns are time points.
```

```
  - regression_model. A scipy regression model to solve Ax = b
  Return:
  - an array of 9 numbers, which are the weights for (x, y, x2, xy, y2, x3, x2y, xy2, y3)
  """
  # Generate the RHS of Ax = b
  dx_dts = get_dx_dts(data[0,:])
  dy_dts = get_dx_dts(data[1,:])

  # Generate the LHS, A of Ax = b
  # Remove the first and last time samples, because finite difference removes them too.
  data_without_ends = data[:, 1:-1]
  # The shape is 28 (time) x 2 (number of library functions)
  x_library = gen_library_matrix(data_without_ends, f_lib)
  y_library = gen_library_matrix(data_without_ends, f_lib)

  regression_model.fit(x_library, dx_dts)
  dx_library_coefs = regression_model.coef_

  regression_model.fit(y_library, dy_dts)
  dy_library_coefs = regression_model.coef_
  return dx_library_coefs, dy_library_coefs
```

SINDy Prediction by Solving ODE

```
def generate_predictions_from_sindy(x_library_coefs, y_library_coefs, f_lib):
  """
  Generate the predicted population formatted as the original data
  by using ODE generated by SINDy. Please see get_sindy_params.
  That is, from 1845 to 1903 in increment of 2 years, a 2 x 30 matrix
  Parameters:
  - poly_weights. The return value of get_sindy_params.
  Return:
  - scipy's integrate sol obj. sol.y gives you the predictions.
    sol.success tells you if the integration was successful.
  """
  def dyn(t, states):
    dx = 0
    dy = 0
    for i, name_f in enumerate(f_lib):
      f = name_f[1]
      f_res = f(states)
      dx += x_library_coefs[i] * f_res
      dy += y_library_coefs[i] * f_res
    return np.array([dx, dy])

  # 1845 to 1903 spans 58 years. Each unit time is 1 year. Let t = 0 be year 1845.
  t_max = 58
  t_span = [0, t_max]
  t_eval = np.arange(0, t_max + util.NEW_DT, util.NEW_DT)
  # Hare and lynx population in 1845
  init_conds = [20, 32]
  sol = integrate.solve_ivp(dyn, t_span, init_conds, method = 'RK45', t_eval = t_eval)
  return sol
```

Joint Probability Distribution Extraction from Population Data

```
def get_population_joint_distribution(population_data, pseudocount = 0.001):
  """
  Parameters
```

```
  - data (2 x m matrix) Rows are population types and columns are time points.
  - pseudocount. Laplacian smoothing parameter for never-seen samples.
  Returns
  - 200 x 200 float matrix where mat[i, j] is P(hare = i, lynx = j) at any time point.
    The joint probability is guaranteed to add up to 1 when summed over the entire sample spac
  """
  counts = np.zeros((200, 200))
  # Additive smoothing. For unseen, assume a pseudocount.
  counts += pseudocount
  for t in range(population_data.shape[1]):
    hare = int(population_data[0, t])
    lynx = int(population_data[1, t])
    counts[hare, lynx] += 1
  counts /= np.sum(counts)
  return counts
```

KL Divergence

```
def get_kl_divergence(gold_distribution, pred_distribution):
  """
  Compute KL divergence of two probability distributions produced by
  get_population_joint_distribution()
  The sample space is all possible integer (hare, lynx) pairs from 0 to 200.
  """
  result = 0
  for hare in range(gold_distribution.shape[0]):
    for lynx in range(gold_distribution.shape[1]):
      p = gold_distribution[hare, lynx]
      q = pred_distribution[hare, lynx]
      result += p * np.log(p/q)
  return result
```

AIC and BIC

```
def get_aic_bic(k, population_data, predictions):
  """
  Compute AIC and BIC for the provided gold population data and the given
  set of predictions.
  For the likelihood, we assume that each sample in time (a hare and lynx population)
  is independently from a time-invariant joint probability distribution interpolated from
  predictions.
  Parameters
  - k. The number of optimizable free parameters of the model.
  - population_data (2 x m matrix) Rows are population types and columns are time points.
    This is the gold standard
  - predictions (2 x m matrix) The predictions
  """
  n = population_data.shape[1]
  pred_distribution = get_population_joint_distribution(predictions)
  ll_tot = 0
  # Log likelihood of entire datast is the sum of log likelihood of each time step,
  # assumed to be independent
  for t in range(n):
    hare = int(population_data[0,t])
    lynx = int(population_data[1,t])
    # This is the probability of observing this one time point.
    p = pred_distribution[hare, lynx]
    ll_tot += np.log(p)
```

```
    aic = 2 * k - 2 * ll_tot
    bic = np.log(n) * k - 2 * ll_tot
    return aic, bic
```

APPENDIX B: SUPPORTING PYTHON CODES

The rest of the supporting Python codes can be found on https://github.com/sjonany/AMATH563-public/tree/master/hw2/code