AMATH 563 Sparse Regression

Tyler Chen

ABSTRACT

We outline a sparse regression framework for finding a model given some data. This framework is implemented in Python and applied to two data sets. These examples demonstrate the difficulty of applying these methods to systems with limited data but yield results which encourage furture work.

I INTRODUCTION AND OVERVIEW

The classic example of data science and data driven modeling is Johannes Keppler extracting the laws of planetary motion from data collected by Tycho Brahe. In the present we are able to gather and save a huge amount of data. As a result, techniques must be developed to extract useful information and models from this data. One such approach, explain in more detail in [1, 2], is sparse regression. In this paper we outline the theoretical framework of sparse regression, implement such a framework in Python, and apply the framework to two data sets.

II THEORETICAL BACKGROUND

Sparse Regression

Suppose we have data $d \in \mathbb{R}^m$ for which we would like to find a model. Rather than picking a model ahead of time and "imposing" it on our data, we can construct a more general "library" of functions $f_i \in \mathbb{R}^m$, $i = 1, 2 \cdots, n$ (each possibly depending on d) from which to pick our model used to represent d. We take the model y to be a linear combination of of these functions with coefficients $c \in \mathbb{R}^n$. In some sense, a model which depends on fewer terms is desirable, and we can attempt to find such a model by promoting sparsity in the coefficients of the linear combination of our library functions.

Written in matrix form,

$$\begin{bmatrix} | & | & & | \\ f_1 & f_2 & \cdots & f_n \\ | & | & & | \end{bmatrix} \begin{bmatrix} c_1 \\ \vdots \\ c_n \end{bmatrix} \approx \begin{bmatrix} d_1 \\ \vdots \\ d_m \end{bmatrix}$$

In practice this will be an overdetermined system as we will have many samples in d but hope to find a "simple" model. The classic way to solve an overdetermined system is least squares regression. However, since least squares gives a relatively "balanced" weighting to all of the functions in the library, the coefficients chosen will almost certainty all be nonzero. As a result, we would like to find a method of solving the overdetermined system which will force some coefficients to be zero.

We can generalize the least squares problem as finding $c \in \mathbb{R}^n$ which, for a fixed A and d, solves the minimization problem,

$$\min_{c \in \mathbb{R}^n} g(A, d, c) \tag{1}$$

A common choice for g is,

$$g(A, d, c) = \frac{1}{2} \|d - Ac\|_2^2 + \lambda \|c\|_1 \tag{2}$$

When $\lambda = 0$ this is the standard least squares fit. For $\lambda > 0$, extra weight is placed on the sparsity of the solution. This is commonly referred to as least absolute shrinkage and selection operator (lasso).

KL Divergence, AIC, BIC

Once we have constructed a model, y, we would like to know how "good" of a model it is. We use three metrics, KL divergence, AIC, and BIC, to measure how accurate our models are. We pull much of our methodology in this section from Chapters 1 and 2 of [3].

Given two (discrete) probability distributions P and Q we define the KL divergence,

$$\operatorname{KL}(P, Q) = \sum_i P(i) \log\left(\frac{P(i)}{Q(i)}\right) \tag{3}$$

This is a measure of the information lost when Q is used to approximate P.

We make the assumption that the residuals $d_i - y_i$ are i.i.d. normal random variables with mean zero. Defining, RSS $= \|d_i - y_i\|_2^2$ and variance of these residuals, $\sigma^2 = \text{RSS}/n$, the maximum value of a model's log likelihood function is,

$$\ln \mathcal{L} = -\frac{n}{2}\ln 2\pi - \frac{n}{2}\ln \sigma^2 - \frac{n}{2} \tag{4}$$

Denote the number of parameters in a model y with k and the number of data points with n. We define the AIC and BIC scores,

$$\text{AIC}(d,y) = 2k - 2\ln\mathcal{L} \tag{5}$$
$$\text{BIC}(d,y) = k\ln n - 2\ln\mathcal{L} \tag{6}$$

These scores penalize the use of more parameters and the use of more observations. This makes intuitive sense, since a simpler model which performs as "well" as a more complex one is desirable.

We make the note that what we take as our data depends on the analysis. For instance, even if we are given some data set, if we are doing sparse regression to find a model for the time derivative, we would use the time derivatives as the d_i described in (4).

This makes sense as even a perfect model with a slight error in the initial condition may vary wildly in position after some time, even if the derivatives (which is what were fit) are very accurate.

Cross Validation

In order to make sure our models represent the system we cross validate. Loosely speaking, this means we train a model on part of the data, and compare the prediction to some part of the data which was not used in training. This helps get a sense of how well the model will work on new data.

Time Delay Embedding

If we have a set of measurement for a system, one question is how many hidden variables the system contains. Broadly, time delay embedding hopes to recover the system's dimensionality.

We provide a simple example as intuition for this approach. Consider the system of n coupled linear first order ODEs,

$$x' = Ax, \qquad x(0) = x_0 \tag{ode}$$

If we numerically solve the system using forward Euler, we will have a system of n coupled difference equations,

$$x^{(i+1)} = x^{(i)} + \mathrm{d}tAx^{(i)}, \qquad x^{(0)} = x_0 \tag{diff}$$

Let $x_j^{(k)}$ denote the j-th component of the k-th iterate. Consider the matrix,

$$H = \begin{bmatrix} x_j^{(0)} & x_j^{(1)} & \cdots & x_j^{(n)} \\ x_j^{(1)} & x_j^{(2)} & \cdots & x_j^{(n+1)} \\ \vdots & \vdots & & \vdots \\ x_j^{(k)} & x_j^{(1+k)} & \cdots & x_j^{(n+k)} \end{bmatrix}$$

If $k > n$ then the $n+1$-th row can be written as a linear combination of the previous rows. This means there will be n nonzero singular values in the singular value decomposition (SVD) of H.

Since this only requires a single component of the solutions in time, we make the observation that the dimensionality of the system can be recovered from just the solution $x_j^{(k)}$, $k = 1, 2, \ldots$.

Obviously this is a very simple system. Such an approach can be applied to more complex systems with varying degrees of success. In practice we found that even using other solving methods such as Runge-Kutta 4,5 to discretized the original system gave a Hankel matrix which had no clear cutoff in singular values even when very small time steps were used. That said, since we do not solve for the data used in the Hankel matrix, perhaps this does not matter.

III ALGORITHM IMPLEMENTATION AND DEVELOPMENT

The code we have written is roughly split into three parts: larger libraries such as SciPy, NumPy, and Scikit-Learn, a custom Python module of functions to perform low level tasks not included in these libraries libraries, and the higher level data analysis itself.

Custom Module Functions

All functions written for this library are contained in Appendix A along with a description of their input and outputs. These functions are written to generalize the tasks repeatedly performed in the main code, usually to work with arbitrary dimensional data.

The first group of functions are a variety of differentiation functions. These methods allow for an arbitrary dimension array to be numerically differentiated along any axis. All functions work by slicing the input array along the appropriate axes. This is more efficient computationally than multiplication by a differentiation matrix. Moreover, it allows the arrays to be differentiated without being reshaped.

The next group of functions help with the construction of such libraries of polynomials. Broadly speaking, these functions help construct the terms of the expansion of

$$(x_1 + x_2 + \cdots x_n)^k, \qquad k \in \mathbb{Z}_{\geq 0} \tag{7}$$

Notationally, let $\mathcal{P} : \mathbb{R}^{m\times n} \times \mathbb{Z}_{\geq 0} \to \mathbb{R}^{C(k+n-1,n-1)}$ be defined as,

$$\mathcal{P}(x,k) = \left[\prod_{i=1}^{n} x_i^{\alpha_i}\right]_{\alpha_1\in\mathbb{Z}_{\geq 0}, \sum_i \alpha_i \leq k} \tag{8}$$

That is, the entries of P along the first axis are the $C(k_n - 1, n-1)$ monomials in the expansion of (7).

The last group of functions are to compute KL divergence, AIC, and BIC. These are implemented as described in (3), (5), and (6).

We also write a few wrappers for functions already implemented in Python libraries. This is generally done to simplify the syntax and align it to in class examples.

Implementation of Sparse Regression

We now describe the general framework within which we implement sparse regression.

Data is imported into a multi-dimensional array. In general, one or more axes may be a dummy axes. This means they store measurements of different variables which are not spatially related. In such cases, we never differentiate along this axis as that would have no physical meaning. We choose to store these different variables in the same array rather than different arrays to allow for programmatic manipulation of these variables.

Once the data is collected into a single matrix, we now construct our "library" of functions, as well as the right hand side of our system. In most cases the right hand side it some time derivative of our system, and the library is polynomials in variables and their spacial derivatives.

All differentiation is done using our implemented functions which act multi-dimensional arrays without the need for reshaping. We then construct the library, reshaping it to two dimensions by setting the width as number of parameters in the model.

As described above, we then try to approximately solve our system. This is done using either Numpy's least squares solver, or Scikit-Learn's Lasso solver (using our wrapper). Note that both methods allow for the system to be solved for multiple right hand sides with a single function call. This is often unitized for notational convenience.

The output of the solver calls gives us the coefficients for the linear combination of our library functions. This means that we are able to

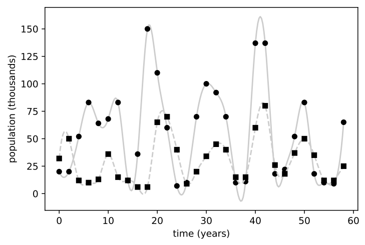
The models are then reconstructed and stepped out over time using Scipy's initial value problem solver from the integrate package.

IV COMPUTATIONAL RESULTS

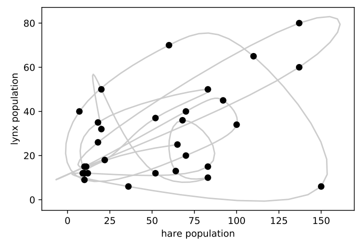
We roughly separate our results by the two data sets to which we apply sparse regression. A more thorough analysis has been done for the population data than the chemical reaction data.

Population Data

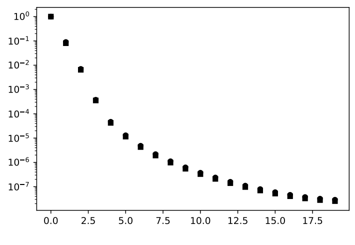
We were given historical data $P = [H, L]$ for the populations of hares and lynx in some region, sampled every two years from 1845 to 1903. Figure 1a shows the original data, along with a local cubic interpolation of each



(a) circle: population (circle), lynx population (square), interpolated populations (grey)

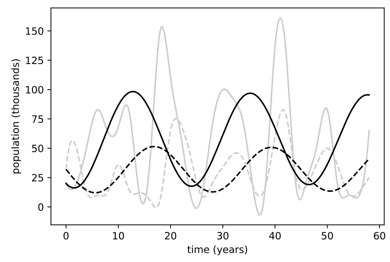


(b) phase diagram with interpolated populations (grey)

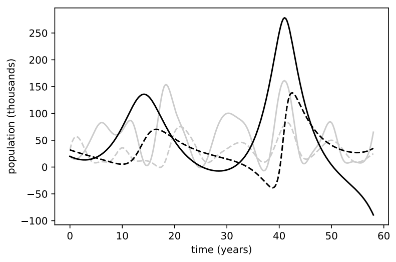


(c) singular values of time delay matrices (using data interpolated at 1451 points); hare (circle), lynx (square)

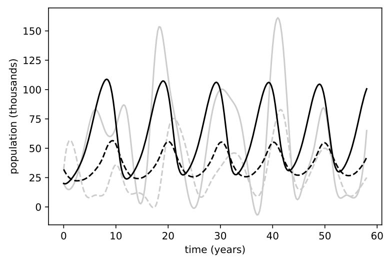
Figure 1



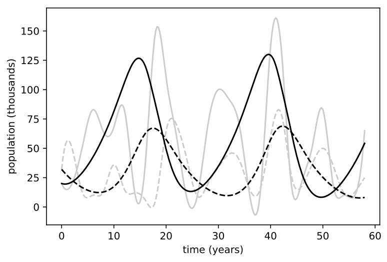
(a) `m100`



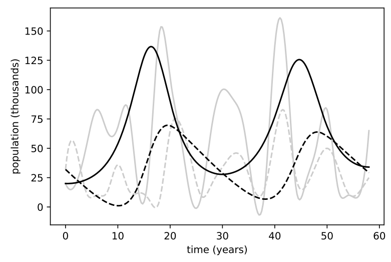
(b) `m200`



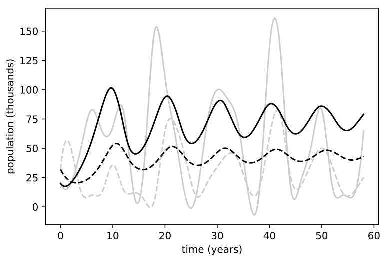
(c) `m500`



(d) `m211`



(e) `m222`



(f) `m511`

Figure 2: hare model (solid), lynx model (dashed)

data set. Figure 1b shows the phase portraits of the original and interpolated data. The data was interpolated (using local cubic interpolation) because of the limited number of samples. While some models do converge with the original data, we generally have better success with the interpolated data.

All of our models are found by solving an overdetermined system of the form

$$\partial_t P = L(P)c \tag{9}$$

where $L(P) \in \mathbb{R}^{m,n}$ is some library function and $c = [c_H, c_L] \in \mathbb{R}^{n,2}$ are the coefficients of the model for the Hare and Lynx populations respectively. Note that $\partial_t P \in \mathbb{R}^{m,2}$ so this system is actually two seperate $m \times n$ overdetermined systems to be solved independently. However, for notational convenience we write them in this form.

We first try using a simple library $\mathcal{P}(P,1) = [1, H, L]$ in (9) and solve the corresponding systems.

Since we used a library with maximum degree 1, and did not drop terms from the library for either the hares or the lynx (discussed below), we will refer to this model as `m100`. As pictured in Figure 2, this model gives a solution which does not decay or explode over time. Moreover, it is oscillatory. However, based on a visual check, the frequency of the oscillations do not match with the data.

Expanding on this idea, we tried libraries of the same form but higher degree. In particular, we tested models with libraries of degree 2 and 5. These models are respectively refereed to as `m200` and `m500`. We note that `m200` has some points which are negative, indicating an unphysical solution. On the other hand, `m500` displays relatively frequent oscillations which seem to roughly match those of the data.

model	KL div.	AIC (hare)	BIC (hare)	AIC (lynx)	BIC (lynx)
`m100`	0.0035	393.842	398.045	311.082	315.285
`m200`	0.0480	470.660	479.068	399.749	408.156
`m500`	0.0387	476.720	506.145	382.221	411.646
`m211`	0.0207	399.384	406.390	326.423	333.429
`m222`	0.0286	414.228	419.833	326.793	332.398
`m511`	0.6857	449.901	477.925	355.687	383.710

Table 1: Population Model Accuracy Metrics

With these models as a baseline, we explored different methods of solving our system and different library functions. In particular, we attempted to use lasso. However, for every $\alpha > 0$ which we tried, the model either decayed (often to negative populations), or diverged. We interpret this to mean that our least squares models are the sparsest model given the libraries we use.

While lasso was generally unsuccessful, we did have some encouraging results with culling our coefficient list and re-regressing using least squares. Our approach is as follows: compute the "least important" coefficient by picking the coefficient which would have smallest weight if the columns of the library were normalized, delete this component, re-regress. Using this approach, we can delete a variable number of parameters from each of our coefficient vectors, reducing the number of parameters in our models. Deleting one library function from the libraries for the hares and the lynx in models `m200` and `m500` give rise to `m211` and `m511`.

We attempt to drop a more terms, with varying success. It seems like many models end up decaying, even if there are oscillations present. One interesting model is `m222`, where two terms are dropped from each library, starting with a library of second degree monomials. This model seems to capture some of the larger spikes in population.

Table 1 shows the KL divergence, AIC, and BIC scores of the various models described above.

We compute the KL divergence in phase space using 4 bins centered at the average of the original data points. The distribution is done in phase space, since we are interested in how often the hare and lynx population combinations predicted by our model match actual data.

Note that while the models came from interpolated data, the residuals used for AIC and BIC are computed only at data points from the original data. In particular, we take the time derivative for each the hare and lynx populations at each point, and compare them to the our model at these points.

We do not cross validate this model because of the limited data. We did run some tests where the models were trained on the first bit of the data, and compared to the interpolated model on the later part of the data to varying degrees of success. However, since there is not much data to train over, we did not think this was worth including in the report.

We compute the Hankel matrices of each the hare and lynx populations, using the interpolated data (as this is what we used to find our models). Figure 1c shows the singular values of the time delay matrices. While there is some change in behavior after the 5th singular values, it is not immediately apparent the dimensionality of the system. Without further exploration into the use of time delay embedding, it is hard to say whether this in-clarity is due to to our methodology, or due to the method not being applicable to this type of system.

BZ reaction data

We are also given a video with 1200 frames of a chemical reaction. Figure 3 shows this data at two different frames. Since the data is so large, and since there are clearly many different regimes, we try to find models for three subsamples of the data. We take two one dimensional cuts along the dotted and dashed diagonals, as well as one two dimensional cut inside the square.

Let u denote the data from the one dimensional cut. We solve,

$$\partial_t u = [u, u^2, u^3, u_x, u_x x, u_x u, u_x u_x, u_x u_x x]c$$

We integrate our models over time using Scipy's integrate package. Unfortunately our models decay to zero rather quickly, so we do not do any analysis of them.

Let I denote the data from the two dimensional slice. We solve,

$$\partial_t I = [I, I_x, I_y, I_{xx}, I_{xy}, I_{yy}]c \tag{10}$$

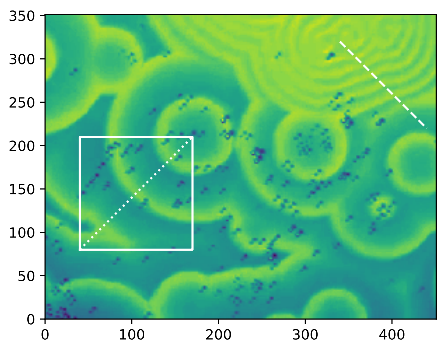
$$\partial_t^2 I = [I, I_x, I_y, I_{xx}, I_{xy}, I_{yy}]c \tag{11}$$

Interestingly, the coefficients for the ∂_t model are much (one/two orders of magnitude) larger on the I_x and I_y terms than other terms. Similarly, the coefficients for the ∂_t^2 model are much larger on the I_{xx} and I_{yy} terms than other terms. These suggest we have found a wave equation, which is expected based on what the image looks like over time.

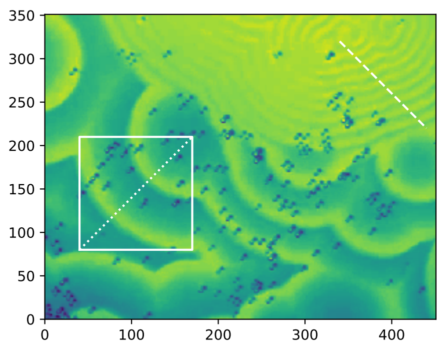
Despite the coefficients looking encouraging, when we integrate the model over time (using a difference scheme in the double time derivative case), and again the solution decays.

If we wanted to compute the KL divergence of these models over phase space we would have to integrate over a space with dimension equal to the number of variables. Since we have only 1200 data points for each variable it is not reasonable to try to compute the KL divergence in thsi way.

Instead we compute the AIC and BIC scores (as a proof of concept). The models from (10) and (11) each have AIC, BIC scores, 247869.27, 247915.68 and 254525.35, 254571.76 respectively. That said, without any other models to compare, these are not particularly meaningful. However we at least demonstrate how we can compute these scores.



(a) $t = 650$



(b) $t = 1100$

Figure 3: various frames of the given data. Square and lines indicated subsets of the data on which sparse regression was applied.

V SUMMARY AND CONCLUSIONS

In this project we described, implemented, and applied the sparse regression framework. While the implementation is relatively straightforward, the results are mixed.

Despite the small population data set we were able to find oscillatory models. That said, these models are generally not very good. In fact, the simples `m100` performs best under all metrics, despite the fact that it is clearly a bad representation of the dynamics at play in the data. It was not clear that our Hankel matrix provided any results.

With the chemical reaction data we were able to apply sparse regression; however we did not manage to find a non-decaying solution. Given more time it might be possible to find non-trivial solutions by picking a different/larger library.

Cross validation was not done due to the lack of data. In order to cross validate our models, enough data would be needed to train on some subset of the data. While this may have been possible for the chemical reaction data, we did not have good enough results to justify trying to train on even less data.

Through this project we became aware of some of the difficulties of applying sparse regression to small data sets. While we did manage to get some results, it is not clear that they are meaningful. Even so, a lot of intuition and experience was gained during the process.

REFERENCES

[1] Steven L. Brunton, Joshua L. Proctor, and J. Nathan Kutz. Discovering governing equations from data by sparse identification of nonlinear dynamical systems. *Proceedings of the National Academy of Sciences*, 113(15):3932–3937, 2016.

[2] J. Nathan Kutz. Course lecture notes. 2018.

[3] Kenneth P. Burnham and David R. Anderson. *Model Selection and Multimodel Inference: A Practical Information-Theoretic Approach*, volume 67. 01 2002.

VI APPENDIX A

```
'''
Parameters
----------
l: integer
n: integer
start: integer

Returns
-------
generator of all tuples of length l whose minimum entry is start and whose entries sum to at
    most n
'''

def get_poly_exponents(l, n, start=0):
    if l == 0:
        yield ()
    else:
        for x in range(start, n+1):
            for t in get_poly_exponents(l - 1, n):
                if sum(t) + x <= n:
                    yield t + (x,)
```

```
'''
Parameters
----------
x: array like
max_deg: integer

Returns
-------
ndarray whose entries are sums of the expansion of (x_0 + ... + x_n)^max_deg, where n = len(x
    ). If entries of x are themselves arrays, then multiplication of terms in the expansion
    is done componentwise
'''

def bin_exp(x,max_deg):
    E_ = np.array(list(get_poly_exponents(len(x),max_deg)))
    E = E_[np.argsort(np.sum(E_,axis=1))]
    return np.array([np.prod([x[k]**e[k] for k in range(len(x))],axis=0) for e in E ]).T
```

```
'''
Parameters
----------
a: array like
dx: float
axis: integer (must be between 0 and dimension of A-1)
endpoints: boolean

Returns
-------
array numerically differentaited with mesh spacing dx a along given axis using centered
    second order finite difference method. If endpoints flag is True, then first order
    forward and backward differences are used at the endpoints.
'''

def diff(a,dx,axis=0,endpoints=False):
    a = np.asanyarray(a)
    nd = a.ndim

    slice1 = [slice(None)] * nd
    slice2 = [slice(None)] * nd
    slice1[axis] = slice(2, None)
    slice2[axis] = slice(None, -2)
    slice1 = tuple(slice1)
    slice2 = tuple(slice2)

    da = np.subtract(a[slice1], a[slice2])
    if endpoints:
        slicel1 = [slice(None)] * nd
        slicel2 = [slice(None)] * nd
        slicel1[axis] = slice(1,2)
        slicel2[axis] = slice(0,1)
        slicel1 = tuple(slicel1)
        slicel2 = tuple(slicel2)

        dl = np.subtract(a[slicel1], a[slicel2])

        slicer1 = [slice(None)] * nd
        slicer2 = [slice(None)] * nd
        slicer1[axis] = slice(-1,None)
        slicer2[axis] = slice(-2,-1)
        slicer1 = tuple(slicer1)
        slicer2 = tuple(slicer2)

        dr = np.subtract(a[slicer1], a[slicer2])

        return np.concatenate((dl,da/2,dr),axis=axis)/dx

    else:
        return da/(2*dx)
```

```
'''
Parameters
----------
a: array like
dx: array of float
axis: array of integer (lenth must equal dx, entries must be between 0 and dimension of A-1)
endpoints: boolean

Returns
-------
array numerically differentaited with mesh spacing dx[i] a along given axes using centered
    second order finite difference method in each direction. If endpoints flag is True, then
    first order forward and backward differences are used at the endpoints. This is a wrapper
     for calling diff repeatedly.
'''

def pdiff(a,dx,axis=[0],endpoints=False):
    if len(axis) == 1:
        return diff(a,dx[0],axis[0],endpoints=endpoints)
    else:
        a = diff(a,dx[0],axis[0],endpoints=endpoints)
        return pdiff(a,dx[1:],axis[1:],endpoints=endpoints)
        '''
Parameters
----------
a: array like
dx: float
axis: integer (must be between 0 and dimension of A-1)
endpoints: boolean

Returns
-------
array numerically differentaited with mesh spacing dx a along given axis using centered
    second order finite difference method for the second derivative. If endpoints flag is
    True, then first order forward and backward differences are used at the endpoints. Due to
     floating point errors it is better to use this function than diff twice, or equivalently
     pdiff.
'''

def diff2(a,dx,axis=0,endpoints=False):
    a = np.asanyarray(a)
    nd = a.ndim

    slice1 = [slice(None)] * nd
    slice2 = [slice(None)] * nd
    slice3 = [slice(None)] * nd
    slice1[axis] = slice(2, None)
    slice2[axis] = slice(1,-1)
    slice3[axis] = slice(None, -2)
    slice1 = tuple(slice1)
    slice2 = tuple(slice2)
    slice3 = tuple(slice3)

    da = np.subtract(np.add(a[slice1], a[slice3]),2*a[slice2])

    if endpoints:
        slicel1 = [slice(None)] * nd
        slicel2 = [slice(None)] * nd
        slicel3 = [slice(None)] * nd
        slicel1[axis] = slice(2,3)
        slicel2[axis] = slice(1,2)
        slicel3[axis] = slice(0,1)
        slicel1 = tuple(slicel1)
        slicel2 = tuple(slicel2)
        slicel3 = tuple(slicel3)

        dl = np.subtract(np.add(a[slicel1], a[slicel3]),2*a[slicel2])

        slicer1 = [slice(None)] * nd
        slicer2 = [slice(None)] * nd
        slicer3 = [slice(None)] * nd
        slicer1[axis] = slice(-1,None)
        slicer2[axis] = slice(-2,-1)
        slicer3[axis] = slice(-3,-2)
        slicer1 = tuple(slicer1)
        slicer2 = tuple(slicer2)
        slicer3 = tuple(slicer3)

        dr = np.subtract(np.add(a[slicer1], a[slicer3]),2*a[slicer2])
        return np.concatenate((dl,da,dr),axis=axis)/dx**2

    else:
        return da/dx**2
```

```
'''
Parameters
----------
A: array like
b: array like (must be compatiable with A)
alpha: float

Returns
-------
array solving minimization problem: min_x: ||b-Ax||_2 + alpha*||x||_1
'''

def lasso(A,b,alpha=1):
    reg = linear_model.Lasso(alpha=alpha/len(A))
    reg.fit(A,b)
    return reg.coef_
```

```
'''
Parameters
----------
p: array like (normalized probability distribution)
p_m: array like (normalized probability distribution)

Returns
-------
KL divergence of distributions
'''

def kl_divergence(p,p_m):
    return np.sum(p * np.log(p / p_m))

#<start:kl_divergence>

#<start:AIC>
'''
Parameters
----------
model: array like (model points)
data: array like (actual data points)
k: integer (number of parameters)

Returns
-------
AIC score of models
'''

def AIC(model,data,k):
    n = len(data)
    RSS = np.linalg.norm(model-data)**2
    sigma2 = RSS / n
    logL = - n* np.log(2*np.pi) / 2 - n * np.log(sigma2) - n / 2
    return 2*k - 2*logL
#<end:AIC>
```

```
#<start:BIC>
'''
Parameters
----------
model: array like (model points)
data: array like (actual data points)
k: integer (number of parameters)

Returns
-------
BIC score of models
'''

def BIC(model,data,k):
    n = len(data)
    RSS = np.linalg.norm(model-data)**2
    sigma2 = RSS / n
    logL = - n* np.log(2*np.pi) / 2 - n * np.log(sigma2) - n / 2
    return np.log(n) * k - 2 * logL
#<end:BIC>
```

```
'''
Parameters
----------
model: array like (model points)
data: array like (actual data points)
k: integer (number of parameters)

Returns
-------
AIC score of models
'''

def AIC(model,data,k):
    n = len(data)
    RSS = np.linalg.norm(model-data)**2
    sigma2 = RSS / n
    logL = - n* np.log(2*np.pi) / 2 - n * np.log(sigma2) - n / 2
    return 2*k - 2*logL
```

```
'''
Parameters
----------
model: array like (model points)
data: array like (actual data points)
k: integer (number of parameters)

Returns
-------
BIC score of models
'''

def BIC(model,data,k):
    n = len(data)
    RSS = np.linalg.norm(model-data)**2
    sigma2 = RSS / n
    logL = - n* np.log(2*np.pi) / 2 - n * np.log(sigma2) - n / 2
    return np.log(n) * k - 2 * logL
```

VII APPENDIX B

```
import numpy as np
np.set_printoptions(threshold=np.nan)
np.set_printoptions(precision=5)

import matplotlib.pyplot as plt

import scipy as sp
from scipy.interpolate import interp1d
from scipy import integrate

from data_driven_modeling import bin_exp,diff,diff2,lasso, kl_divergence, AIC, BIC

#%% setup data

# manually input data
hare_population = np.array([20,20,52,83,64,68,83,12,36,150,110,60,7,10,70,100,92,
    70,10,11,137,137,18,22,52,83,18,10,9,65])
lynx_population = np.array([32,50,12,10,13,36,15,12,6,6,65,70,40,9,20,34,45,40,15,
    15,60,80,26,18,37,50,35,12,12,25])

# make population vector from data
P = np.array([hare_population,lynx_population])

# set up times data was taken (in years since first data point)
t = np.linspace(0,60,30,endpoint=False)


#%% interpolate data
P_interp = interp1d(t, P, kind='cubic',axis=1)

# evaluate interpolation on a finer mesh
mesh_scale = 8
t_fine = np.linspace(0,58,30*mesh_scale-(mesh_scale-1))
P_ = P_interp(t_fine)


# plot interpolation vs. data
plt.plot(t_fine,P_[0],'.8');
plt.plot(t,P[0],color='0',marker='o',ms=5,linestyle='None')
plt.plot(t_fine,P_[1],'.8',linestyle='--');
plt.plot(t,P[1],color='0',marker='s',ms=5,linestyle='None')
plt.xlabel('time (years)')
plt.ylabel('population (thousands)')
plt.savefig('img/P/interp.pdf')

# plot interpolation phase plot
plt.figure()
plt.plot(P_[0],P_[1],color='.8');
plt.plot(P[0],P[1],color='0',marker='o',linestyle='None');
plt.xlabel('hare population')
plt.ylabel('lynx population')
plt.savefig('img/P/phase.pdf')


#%% dP/dt = kth degree polynomial in entries of P
```

```
def diff_model(t_fine, P_, max_deg, num_refinements):
    # compute time derivative of data
    dt = t_fine[1]-t_fine[0]
    dP = diff(P_,dt,endpoints=True,axis=1)


    # make dictionary of polynomials in both population1
    A = bin_exp(P_,max_deg)
    weights  = np.sum(A,axis = 0)

    # compute coefficeints
    c = np.linalg.lstsq(A,dP.T)[0]
    # c[:,0] = lasso(A,dP[0].T,alpha=100)
    # c[:,1] = lasso(A,dP[1].T,alpha=100)

    params = np.ones(np.shape(c))

    # iteritively remove some functions from library
    for i in range(2):
        for j in range(num_refinements[i]):

            # find nonzero entries
            nz = np.nonzero(params[:,i])[0]

            # find index of lowest weight of these entires
            x = np.argmin( weights[nz] * c[nz,i] )

            # zero this index
            params[nz[x],i] = 0

            # compute new coefficeints
            c[:,i] = np.linalg.lstsq(A*params[:,i],dP[i].T)[0]

    # solve solution from given coefficeints
    def rhs(t,xy):
        return bin_exp(xy,max_deg)@c
    ivp = integrate.solve_ivp(rhs,[0,60],P_[:,0],dense_output = True)

    return {"lib": lambda xy: bin_exp(xy,max_deg)@c, "model": ivp.sol, "num_params":np.sum(
       params,axis=0), "num_data": len(t_fine)}

#%% test various models

for max_deg, num_refinements in
    [[1,[0,0]],[2,[0,0]],[5,[0,0]],[1,[1,1]],[2,[1,1]],[2,[2,2]],[5,[1,1]]]:

    d_m = diff_model(t_fine,P_,max_deg,num_refinements)

    lib = d_m['lib']
    model = d_m['model']
    num_data = d_m['num_data']
    num_params = d_m['num_params']

    bins = [np.arange(-25,200+1,12.5),np.arange(-25,200+1,12.5)]
    eps = 1

    center= np.average(P,axis=1)

    # compute historgram of data and model in phase space
    p_ = np.histogram2d(P[0],P[1],[[-np.inf,center[0],np.inf],[-np.inf,center[1],np.inf]])[0]
    m_ = np.histogram2d(model(t)[0],model(t)[1],[[-np.inf,center[0],np.inf],[-np.inf,center
        [1],np.inf]])[0]

    p_ /= np.sum(p_)
    m_ /= np.sum(m_)

    # compute derivative of model at data points
    h = (t_fine[1]-t_fine[0])
    derivs = (model(t+h) - model(t-h)) / (2*h)

    # compute library times data points
    libs = lib(P).T

    # compute KL divergence
    data = np.array([kl_divergence(p_,m_)])
    np.savetxt('img/P/'+str(max_deg)+str(num_refinements[0])+str(num_refinements[1])+'.txt',
        data, fmt='%.4f')

    # compute AIC/BIC scores for each population
    for i in range(2):
        scores = {"AIC" : AIC(libs[i],derivs[i],num_params[i]),
                  "BIC": BIC(libs[i],derivs[i],num_params[i])}

        data = np.array([[scores['AIC'], scores['BIC']]])
        np.savetxt('img/P/'+str(max_deg)+str(num_refinements[0])+str(num_refinements[1])+'_'+
            str(i)+'.txt',data, fmt='%.3f & %.3f')


    # plot model vs actual data
    plt.figure()
    plt.plot(t_fine,P_[0],'.8');
    plt.plot(t_fine,P_[1],'.8',linestyle='--');
    plt.plot(t_fine,model(t_fine)[0],color='0',linestyle='-')
    plt.plot(t_fine,model(t_fine)[1],color='0',linestyle='--')
    plt.xlabel('time (years)')
    plt.ylabel('population (thousands)')
    plt.savefig('img/P/'+str(max_deg)+str(num_refinements[0])+str(num_refinements[1])+'.pdf')


#%% construct time delay matrices

# evaluate interpolation on a finer mesh
mesh_scale = 50
t_fine = np.linspace(0,58,30*mesh_scale-(mesh_scale-1))
P_ = P_interp(t_fine)

depth = 20
width = np.shape(P_)[1] - depth
H = np.zeros((2,depth,width))
H[0] = np.array([P_[0,i:width + i] for i in np.arange(depth)])
H[1] = np.array([P_[1,i:width + i] for i in np.arange(depth)])

# take SVD
u,s,vh = np.linalg.svd(H)

# plot singular values
plt.figure()
ax = plt.gca()
```

```
ax.set_yscale('log')
plt.plot(np.arange(depth),s[0]/np.max(s[0]),color='0',marker='o',ms=5,linestyle='None')
plt.plot(np.arange(depth),s[1]/np.max(s[1]),color='0',marker='s',ms=5,linestyle='None')
plt.savefig('img/P/time_delay.pdf')
```

```
import numpy as np
from scipy import io,integrate
import matplotlib.pyplot as plt
np.set_printoptions(threshold=np.nan)
np.set_printoptions(precision=5)

from scipy.interpolate import interp1d
from scipy.ndimage.filters import maximum_filter
from scipy.ndimage.filters import gaussian_filter

from data_driven_modeling import get_poly_exponents,bin_exp,diff,pdiff,diff2,lasso,
    kl_divergence,AIC,BIC


BZ_tensor=np.load('BZ.npz')['BZ_tensor'].astype('int16')

#%% pick interesting smaller region
region = (slice(80,210),slice(40,170),slice(650,None))

I = BZ_tensor[region].astype('float64')

#%% save plots of image

plt.figure()
t = 650
m = plt.pcolormesh(BZ_tensor[:,:,t])
m.set_rasterized(True)
plt.plot([40,170],[80,210], color='white', linestyle=':')
plt.plot([340,440],[320,220], color='white', linestyle='--')
plt.plot([40,40,170,170,40],[80,210,210,80,80], color='white')
plt.axis('image')
plt.savefig('img/BZ/slice_'+str(t)+'.pdf')


#%% TRY 1D SLICES

u = I[np.arange(130),np.arange(130)]
#u = BZ_tensor[np.arange(320,220,-1),np.arange(340,440),400:]

[M,T] = np.shape(u)

plt.figure()
plt.scatter(np.arange(M),u[:,200])
plt.show()

dt = 1;
u_t = diff(u,dt,axis=1,endpoints=True);
u_tt = diff2(u,dt,axis=1,endpoints=True);

dx = 1;
u_x = diff(u,dx,endpoints=True)
u_xx = diff2(u,dx,endpoints=True)
u2_x = diff(u**2,dx,endpoints=True)
```

```
A = np.array([u,u**2,u**3, u_x, u_xx, u_x*u, u_x*u_x, u_x*u_xx]).reshape(-1,M*T).T

#%% regress

xi= np.linalg.lstsq(A,u_t.reshape(-1))[0];

#%%
plt.figure();
plt.bar(np.arange(1,len(xi)+1),xi);
plt.show();


#%% define RHS for differential equation
def f(t,y):
    dx = 1;
    y_x = diff(y,dx,endpoints=True)
    y_xx = diff2(y,dx,endpoints=True)
    y2_x = diff(y**2,dx,endpoints=True)

    return np.array([y,y**2,y**3, y_x, y_xx, y_x*y, y_x*y_x, y_x*y_xx]).reshape(-1,len(y)).T
        @ xi

#%% run forward euler to solve

ys = np.zeros((M,T))
dt = 1
ys[:,0] = u[:,0]

for i in range(T-1):
    ys[:,i+1] = ys[:,i] + dt * f(0,ys[:,i])

#%% solve using Runge-Kutta 45

sol = integrate.solve_ivp(f,[0,550],u[40],method='RK45')

plt.scatter(sol.t, sol.y[1])

#%% TRY 2d cut

# what is sensor frequency?
[M,N,T]=np.shape(I)

dt = 1
I_t = diff(I,dt,axis=2,endpoints=True)
I_tt = diff2(I,dt,axis=2,endpoints=True)


# build library
dx = 1
dy = 1

I_x = diff(I,dx,axis=0,endpoints=True)
I_y = diff(I,dy,axis=1,endpoints=True)
I_xx = diff2(I,dx,axis=0,endpoints=True)
I_xy = pdiff(I,[dx,dy],axis=[0,1],endpoints=True)
I_yy = diff2(I,dy,axis=1,endpoints=True)

lib = np.array([I, I_x, I_y, I_xx, I_xy, I_yy]).reshape(-1,M*N*T).T;
lib2 = np.array([I, I_x, I_y, I_xx, I_xy, I_yy ]).reshape(-1,M*N*T).T;
```

```
#%% compute coefficients
c = np.linalg.lstsq(lib,I_t.reshape(-1))[0]
c2 = np.linalg.lstsq(lib2,I_tt.reshape(-1))[0]

#%% helper function for AIC BIC
def lib_at_c(I,c):
    dx = 1
    dy = 1

    I_x = diff(I,dx,axis=0,endpoints=True)
    I_y = diff(I,dy,axis=1,endpoints=True)
    I_xx = diff2(I,dx,axis=0,endpoints=True)
    I_xy = pdiff(I,[dx,dy],axis=[0,1],endpoints=True)
    I_yy = diff2(I,dy,axis=1,endpoints=True)

    return (np.array([I, I_x, I_y, I_xx, I_xy, I_yy]).reshape(-1,M*N).T @ c);

#%% compute library at all times
libs = np.zeros((M*N,T))
libs2 = np.zeros((M*N,T))
for t in range(T):
    libs[:,t] = lib_at_c(I[:,:,t],c)
    libs2[:,t] = lib_at_c(I[:,:,t],c2)

#%% compute AIC BIC

data = np.array([[AIC(libs,I_t.reshape(M*N,T),len(c)), BIC(libs,I_t.reshape(M*N,T),len(c))]])
data2 = np.array([[AIC(libs2,I_t.reshape(M*N,T),len(c)), BIC(libs2,I_t.reshape(M*N,T),len(c))
    ]])

np.savetxt('img/BZ/2d.txt',data, fmt='%.2f, %.2f')
np.savetxt('img/BZ/2d2.txt',data2, fmt='%.2f, %.2f')


#%% plot coefficients

plt.figure();
plt.bar(np.arange(1,len(c)+1),c);
plt.show();

plt.figure();
plt.bar(np.arange(1,len(c2)+1),c2);
plt.show();

#%% define RHS for sovler

def f(t,y):
    dx = 1
    dy = 1

    I = np.reshape(y,(M,N))

    I_x = diff(I,dx,axis=0,endpoints=True)
    I_y = diff(I,dx,axis=1,endpoints=True)
    I_xx = diff2(I,dx,axis=0,endpoints=True)
    I_xy = pdiff(I,[dx,dx],axis=[0,1],endpoints=True)
    I_yy = diff2(I,dx,axis=1,endpoints=True)
    return np.array([I, I_x, I_y, I_xx, I_xy, I_yy]).reshape(-1,M*N).T @ c;

#%% solve
sol = integrate.solve_ivp(f,[0,T],np.reshape(I[:,:,0],-1),t_eval = np.arange(T))
ts = sol.t
ys = np.reshape(sol.y,(M,N,len(ts)))

#%% define rhs for forward euler and integrate vs time twice

def f2(I):
    dx = 1
    dy = 1

    I_x = diff(I,dx,axis=0,endpoints=True)
    I_y = diff(I,dx,axis=1,endpoints=True)
    I_xx = diff2(I,dx,axis=0,endpoints=True)
    I_xy = pdiff(I,[dx,dx],axis=[0,1],endpoints=True)
    I_yy = diff2(I,dx,axis=1,endpoints=True)

    return (np.array([I, I_x, I_y, I_xx, I_xy, I_yy]).reshape(-1,M*N).T @ c2).reshape(M,N);

ys = np.zeros((M,N,T))

ys[:,:,[0,1]] = I[:,:,[0,1]]

dt = .5

for i in range(1,500):
    ys[:,:,i+1] = 2*ys[:,:,i] - ys[:,:,i-1] + dt * f2(ys[:,:,i])


#%%

plt.figure()
plt.pcolormesh(ys[:,:,300])
plt.axis('image')
```