FACULTY OF SCIENCE AND TECHNOLOGY

MASTER THESIS

Study program/specialisation:	Spring 2023
Masters of Science and Engineering /	Open
Robot Technology and Signal Processing	

Author: Sander André Søndeland

Course responsible: Aksel Hiort

Supervisor: Aksel Hiort

Title: Data Driven Model Discovery - Petroleum application

Credits: 30

Keywords:	Pages: 63
SINDy, DISKOS, Data Driven Model	+ appendix, bibliography: 63
	Stavanger 15. juni 2023

Abstract

The SINDy algorithm is a data driven algorithm that discovers dynamical system in data that evolves over time. The method can be utilized for every dataset that evolves over time. In this study we have looked the Lorenz system, covid-19 data and production data from two different oil fields on the Norwegian shelf.

The aim of the study was to investigate if SINDy can be used on the well data to extract sparse and suitable well models. The complexity of the models are decided by the user when using prior knowledge to choose the candidate function. If you have limited knowledge about the system a handful of different models are tested and parameters are optimized to fit the data.

Noisy and spiky data are an issue for the SINDy method due to its use of the differentiated data. Therefor filtering is needed on production data to minimize the large spikes and smooth out the data.

The SINDy algorithm gives good results to the production data using polynomials to describe the data. The results are good for data from Draugen and Statfjord Øst. And the results from the covid-19 data are promising.

List of Figures

2.1 Lorenz system 3
2.2 Origianl Lorenz system and SINDy approximation. 9
2.3 Schematic of SINDy algorithm [17]. 9
2.4 Overview of the PySINDy package [11]. 10
2.5 Flow chart [11]. 11

3.1 Covid data from Afghanistan. 13
3.2 Diskos organization map [1]. 14
3.3 Statfjord C-platform [9]. 14
3.4 Statfjord Øst field. Map taken from Norsk Petroleum. https://www.norskpetroleum.no/interaktivt-kart-og-arkiv/interaktivt-kart/. (accessed: 14.05.23). 15
3.5 Model of water injection of a well [6]. 16
3.6 Production data from Statfjord Øst. 16
3.7 Draugen platform [12]. 17
3.8 Draugen field. Map taken from Norsk Petroleum. https://www.norskpetroleum.no/interaktivt-kart-og-arkiv/interaktivt-kart/. (accessed: 14.05.23) 17
3.9 Production data from Draugen. 18

4.1 Flowchart for the Lorenz system. 25
4.2 Derivaitve with smoothing using the kalman method on data from Draugen. 29
4.3 Derivaitve with smoothing using the finite difference method on data from Draugen. 29

5.1 Covid data from Afghanistan. 34
5.2 Covid results using data from Afghanistan. 35
5.3 Distance between the three wells during the tests for Statfjord Øst. Map taken from Norwegian Petroleum Directorade. https://factmaps.npd.no/factmaps/3_0/. (accessed: 22.05.23). 37
5.4 The result for x-data 39
5.5 The result for y-data 39
5.6 The result for z-data 40
5.7 The result for x-data 42
5.8 The result for y-data 42
5.9 The result for z-data 43
5.10 The result for x-data 45
5.11 The result for y-data 46
5.12 The result for z-data 46
5.13 Distance between the three wells during the tests on Draugen. Map taken from Norwegian Petroleum Directorade. https://factmaps.npd.no/factmaps/3_0/. (accessed: 22.05.23). 47
5.14 The result for x-data 49
5.15 The result for y-data 49
5.16 The result for z-data 50
5.17 The result for x-data 52
5.18 The result for y-data 52
5.19 The result for z-data 53
5.20 Distance between the wells during this test on Draugen. Map taken from Norwegian Petroleum Directorade. https://factmaps.npd.no/factmaps/3_0/. (accessed: 22.05.23). 54
5.21 The result for x-data 56
5.22 The result for y-data 56
5.23 The result for z-data 57
5.24 The data used in this test. 57
5.25 The result for x-data 59
5.26 The result for y-data 59

5.27 The result for z-data 60

Listings

4.1 Imported packages. 20
4.2 Derivative-function 21
4.3 Data generator for Lorenz System 21
4.4 The library generator. 22
4.5 The least-square algorithm to solve equation 2.14. 22
4.6 This algorithm is adding sparsity to the Ξ-matrix. 23
4.7 This is the which computes the approximated u from the SINDy method. 23
4.8 This function returns the right hand side of equation 2.14. 24
4.9 Code for plotting the approximation in 3D. 24
4.10 Packages that were imported. 26
4.11 Code for downloading the dataset from Github. 26
4.12 Getting data for a specific well and adding oil and gas production together. 27
4.13 Using datetime to get the timeinterval of the production. 27
4.14 Median_filter that was used for all production data to smooth out the data. 27
4.15 Defining t and t_np to use in the derivation function afterwards. 27
4.16 Derivation with smoothing using the kalman method. 28
4.17 Derivation with smoothing using the finite difference method. 28
4.18 Creating a matrix for the filtered data. 30
4.19 New library function with sine cosine and exponential functions. 30
4.20 These functions finds the Ξ-matrix and then add sparsity. 31

Contents

Abstract **i**

List of Figures **ii**

Acknowledgements **iv**

1 Introduction **1**

2 Theory **2**

2.1 SINDy 2

2.2 PySINDy - A robust Python package for SINDy 10

3 Construction **12**

3.1 SINDy 12

3.2 Lorenz equation 12

3.3 Covid 19 example 12

3.4 Petroleum data 13

3.4.1 Statfjord Øst 13

3.4.2 Draugen 16

4 Implementation **19**

4.1 Implementation used for the Lorenz system 19

4.2 Implementation for Draugen and Statfjord Øst 25

5 Results **33**

5.1 Covid example 33

5.2 Statfjord Øst 36
5.2.1 Test 1 37
5.2.2 Test 2 40
5.2.3 Test 3 43
5.3 Draugen 47
5.3.1 Test 1 48
5.3.2 Test 2 50
5.3.3 Test 3 53
5.3.4 Test 4 58

6 Conclusion **61**
6.1 Future work 61

Bibliography **63**

A Appendix A **65**
A.1 Code for Lorenz system 65
A.2 Code for Covid-19 72
A.3 Code for SINDy on Statfjord Øst 80
A.4 Code for SINDy on Draugen 97
A.5 Code for PySINDy on Draugen 114
A.6 Master poster 126

Acknowledgements

This master thesis will conclude my master's degree in Robot Technology and Signal Processing at the Univeristy of Stavanger. The thesis is proposed by Aksel Hiorth, and the thesis has been a working process from January 2023 to June 2023.

I want to express gratitude towards my thesis supervisor, Aksel Hiorth, for the excelent guidance and support offered during the work of this master thesis from the very start until the end of the thesis study.

Thanks to my family and girlfriend for the continued encouragement and support throughout my Master's degree.

Stavanger, 15th June 2023

Sander André Søndeland

Chapter 1

Introduction

Using dynamical system as a mathematical framework to describe how the world around us evolves in time is a big task. By utilizing how data evolves over time we can discover and approximate a model of its underlying dynamical system. Data-driven dynamical systems is a rapidly evolving field, and with the influence of big data and machine learning has had its renaissance these last years [14, p. 253].

The paper "Discovering governing equations from data by sparse identification of nonlinear dynamical systems" was first published in 2016 which was the first paper about the SINDy method [17]. Prior knowledge of the system or a handful of candidate functions are used to form a candidate model which describes how the data evolves in time [14, p. 275]. The SINDy algorithm has been utilized to identify system of high dimensional dynamical systems, for example fluid flows [14, p. 278].

The data used in this study was downloaded from Diskos which is a national data repository for exploration and production related data shared by the Authorities and oil companies [3]. In 1995 the Norwegian Petroleum Directorate (NPD) and oil companies represented on the Norwegian shelf designed the Diskos National Data Repository [3]. This contain three different data types which is seismic and navigational data, well data and production data [3].

For this study the production data has been utilized with the SINDy algorithm to discover its underlying dynamical system.

Chapter 2

Theory

Data-driven model discovery is a developing field, and one of these techniques are sparse identification of nonlinear dynamics (SINDy). In recent time there has been a push towrads parsimonious modelling such as the SINDy method [10]. It which was introduced in 2016 by Steve L. Brunton, Joshua L. Proctor and J. Nathan Kutz [17]. A parsimonius model are trying to accomplish high level or prediction with as few predictor variable as possible [10]. These types of models want to make an approach as easy as possible while accomplishing the precision that is needed for it's purpose.

2.1 SINDy

The SINDy algorithm are handed a set of data, and the algorithm gives back a set of dynamcial equations that describes how the data evolves in time. But how does this actually work?

The definition of a dynamical system is "a system whose state evolves with time over a state space according to a fixed rule." [8]. We can consider the form of a dynamical system as equation (2.1) below [17].

$$\frac{d}{dt}x = f(x). \tag{2.1}$$

The Lorenz system can be used to explain how the SINDy algorithm works, and utilize

it to gather data and discover the underlying dynamical equations of the system [18]. The Lorenz equations can be described by these three equations below:

$$\dot{x} = \sigma(y - x), \tag{2.2}$$

$$\dot{y} = x(\rho - z) - y, \tag{2.3}$$

$$\dot{z} = xy - \beta z. \tag{2.4}$$

Here we will put $\sigma = 10$, $\rho = 28$ and $\beta = 8/3$. The Lorenz system with these parameters are shown in figure 2.1. This system is a well-studied dynamical system, and are known to be one of the simplest systems that exhibit chaos [14, p. 254]. In figure 2.1 there is a trajectory of the Lorenz system with initial conditions of x, y and z value has been set to -8, 8 and 27.

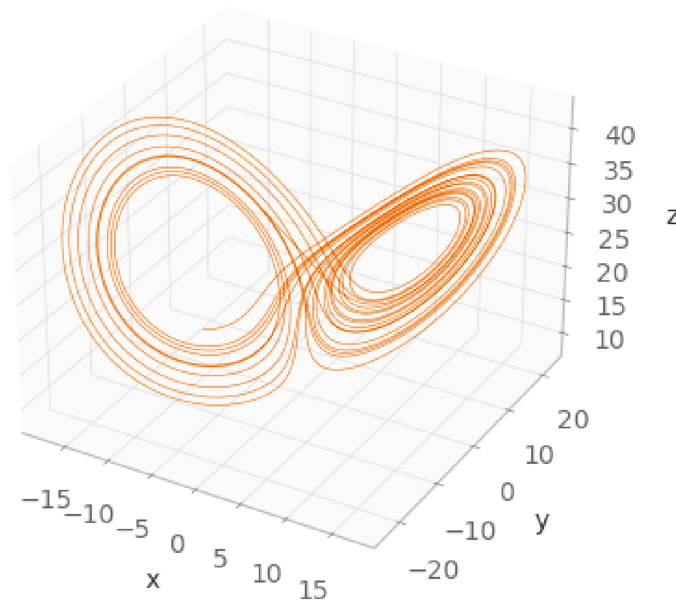


Figure 2.1: Lorenz system

Data for the Lorenz system is created in python, and are generated with no noise in this practical example for the SINDy algorithm. The data that are collected, or as in

this example generated, are arranged into a matrix **X**. Which is an mxn-sized matrix for a n dimensional system, and m is the number of time evolving data collected and are decided by frequency and the time interval the data is collected. The general matrix is shown in matrix (2.5).

$$\mathbf{X} = \begin{bmatrix} \mathbf{X}^T(t_1) \\ \mathbf{X}^T(t_2) \\ \vdots \\ \mathbf{X}^T(t_m) \end{bmatrix} = \begin{bmatrix} \mathbf{X}_1(t_1) & \mathbf{X}_2(t_1) & \dots & \mathbf{X}_n(t_1) \\ \mathbf{X}_1(t_2) & \mathbf{X}_2(t_2) & \dots & \mathbf{X}_n(t_2) \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{X}_1(t_m) & \mathbf{X}_2(t_m) & \dots & \mathbf{X}_n(t_m) \end{bmatrix}. \tag{2.5}$$

$\mathbf{X}_1(t_1)$ is the value for the first timestep for n=1. $\mathbf{X}_2(t_1)$ is then the value for the first timestep but for n=2. The first contains all the values for the first timestep, and the next contains values for the second timestep and this goes up to timestep m, which is the last timestep. The Lorenz system is a three dimensional system which means that n equals to 3. The data has been called x, y and z instead of x_1, x_2 and x_3. Matrix (2.6) gives an example of how the Lorenz data are arranged into a matrix with x, y and z values.

$$\mathbf{X} = \begin{bmatrix} \mathbf{X}(t_1) & \mathbf{Y}(t_1) & \mathbf{Z}(t_1) \\ \mathbf{X}(t_2) & \mathbf{Y}(t_2) & \mathbf{Z}(t_2) \\ \vdots & \vdots & \vdots \\ \mathbf{X}(t_m) & \mathbf{Y}(t_m) & \mathbf{Z}(t_m) \end{bmatrix}. \tag{2.6}$$

There is no limit in the dimensional size of the matrix, but for the Lorenz example the matrix is a three dimensional matrix.

For the simplicity and to match the python code for the Lorenz system, the data are collected as x, y and z instead of x_1, x_2 and x_3 such matrix (2.6) above describes. This is a three-dimensional matrix which means the matrix has three columns for x, y and z. Then the general matrix will look like matrix (2.6), and the one for this example appear like matrix (2.7). The size of the matrix is 100 000 x 3. The timestep starts from 0.001 and the final timestep is 100, and the intervalsize are 0.001.

$$\mathbf{X} = \begin{bmatrix} -8.00 & 8.00 & 27.00 \\ -7.84 & 7.98 & 26.86 \\ \vdots & \vdots & \vdots \\ -7.19 & -11.76 & 16.60 \end{bmatrix}. \tag{2.7}$$

Then the data are differentiated to find $\dot{x}$, $\dot{y}$ and $\dot{z}$ and arrange into a matrix called $\dot{\mathbf{X}}$, just like matrix 2.7. There is a few different ways to find the derivative matrix, $\dot{\mathbf{X}}$, but total variation regularization is recommenden because it denoise the derivative, and to avoid differentiation error [17]. This will work quite well, but another options are to filter both $\mathbf{X}$ and $\dot{\mathbf{X}}$ [17].

For the Lorenz system the derivative package has been utilized to import the dxdt function. Then the finite difference method has been used to find the derivatives. Because the data has zero noise this method doesn't need any filtering, but for a practical example it could be benefical to filter the data during the derivation [17].

$$\dot{\mathbf{X}} = \begin{bmatrix} \dot{\mathbf{X}}^T(t_1) \\ \dot{\mathbf{X}}^T(t_2) \\ \vdots \\ \dot{\mathbf{X}}^T(t_m) \end{bmatrix} = \begin{bmatrix} \dot{\mathbf{X}}_1(t_1) & \dot{\mathbf{X}}_2(t_1) & \dots & \dot{\mathbf{X}}_n(t_1) \\ \dot{\mathbf{X}}_1(t_2) & \dot{\mathbf{X}}_2(t_2) & \dots & \dot{\mathbf{X}}_n(t_2) \\ \vdots & \vdots & \ddots & \vdots \\ \dot{\mathbf{X}}_1(t_m) & \dot{\mathbf{X}}_2(t_m) & \dots & \dot{\mathbf{X}}_n(t_m) \end{bmatrix} \tag{2.8}$$

In matrix (2.8) $\dot{\mathbf{X}}_1(t_1)$ is the first derivative where n equals to 1. Then the value below will be the second derivative for n equals to 1. The second column is the derivatives where n is equal to 2. For the Lorenz system the differentiated matrix $\dot{\mathbf{X}}$ will look like this:

$$\dot{\mathbf{X}} = \begin{bmatrix} \dot{\mathbf{X}}(t_1) & \dot{\mathbf{Y}}(t_1) & \dot{\mathbf{Z}}(t_1) \\ \dot{\mathbf{X}}(t_2) & \dot{\mathbf{Y}}(t_2) & \dot{\mathbf{Z}}(t_2) \\ \vdots & \vdots & \vdots \\ \dot{\mathbf{X}}(t_m) & \dot{\mathbf{Y}}(t_m) & \dot{\mathbf{Z}}(t_m) \end{bmatrix}. \tag{2.9}$$

And after the values have been found the differentiated matrix will look like this

$$\dot{\mathbf{X}} = \begin{bmatrix} 159.12 & -16.44 & -135.11 \\ 158.24 & -16.86 & -134.23 \\ \vdots & \vdots & \vdots \\ -45.50 & -70.16 & 39.84 \end{bmatrix}. \tag{2.10}$$

The next step in the SINDy algorithm is to create a library which contains a list of candidate nonlinear terms. These library are only limited by one's imagination, and it may consist of constant, polynomials of dth-degree and trigonometric terms [17]. A general library, $\Theta(\mathbf{X})$ can be written as:

$$\Theta(\mathbf{X}) = [1 \quad \mathbf{x} \quad \mathbf{x}^2 \quad ... \quad \mathbf{x}^d \quad ... \quad sin(\mathbf{X}) \quad ...]. \tag{2.11}$$

For the Lorenz example a library with polynomials up to 5th order has been chosen, and the size of this matrix will depend on the what the library contains and the size of the data set. A library with polynomials of second degree will have less columns than a library with fifth degree polynomials. The amount of rows will stay the same as long as there is no change in the data set. A library with second degree polynomials will look like this:

$$\Theta(\mathbf{X}) = [1 \quad x \quad y \quad z \quad x^2 \quad xy \quad xz \quad y^2 \quad yz \quad z^2], \tag{2.12}$$

$$\Theta(\mathbf{X}) = \begin{bmatrix} \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & & \vdots \\ 1 & x & y & z & x^2 & xy & xz & y^2 & \dots & z^2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & & \vdots \end{bmatrix}. \tag{2.13}$$

Because each column of $\Theta(\mathbf{X})$ represent x, y and z. Then the we can set up the equation for the algorithm to solve:

$$\dot{\mathbf{X}} = \Theta(\mathbf{X})\Xi. \tag{2.14}$$

First equation (2.14) has to be solved, where Ξ is the unknown. Ξ is a coefficient matrix that chooses the active terms in the library. It's from this matrix we can understand

the solution that SINDy suggests, and how the dynamical system can be described. Before it's possible to understand we need to solve equation (2.14). There's different ways to solve the equation but the book suggest two different algoritms, the LASSO algorithm or the sequential thresholded least-squares (STLS) algorithm [14, p. 276]. In the Lorenz example the sequential thresholded least-squares algorithm has been utilized to find the Ξ-matrix. The next step is to add sparsity to the Ξ-matrix which is an optimalization problem.

The equation for this optimixation problem is showed in equation 2.10, and when this is solved we get the sparse regression. We are asking to minimize the difference between right hand side and left hand side in (2.14).

After the matrix with the differentiated data, matrix (2.10), we can find the Ξ-matrix using the least-squares algorithm and solve equation (2.14). To promote sparsity to the solution we will solve the following optimization problem:

$$\Xi = argmin_{\Xi} \|\dot{\mathbf{X}} - \Theta(\mathbf{X})\Xi\|_2 + \lambda \|\Xi\|_1. \quad (2.15)$$

Where λ is a sparsity-promoting knob [14, p. 276] which desides how sparse the Ξ-matrix should be [14, p. 276].

When the optimization problem has been solved we get a sparse Ξ-matrix that describes the underlying dynamical system of the data. For the Lorenz system our Ξ-matrix turned out like this:

$$\Xi = \begin{bmatrix} 0 & 0 & 0 \\ -10 & 27.8 & 0 \\ 10 & -1.0 & 0 \\ 0 & 0 & -2.7 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}. \tag{2.16}$$

From matrix (2.16) $\dot{x}$, $\dot{y}$ and $\dot{z}$ can be found, and then it's possible to compare what SINDy found with equation (2.2), (2.3) and (2.4) from the start. From the Ξ-matrix (2.16) the equations the SINDy algorithm suggested can be written as:

$$\dot{x} = -10x + 10y = 10(y - x), \tag{2.17}$$

$$\dot{y} = 27.8x - y - xz \approx x(28 - z) - y, \tag{2.18}$$

$$\dot{z} = -2.7y + xy \approx xy - 8/3z. \tag{2.19}$$

The results from the SINDy algorithm for the Lorenz system is good and it's clear to see that the trajectory of the systems are really similar.

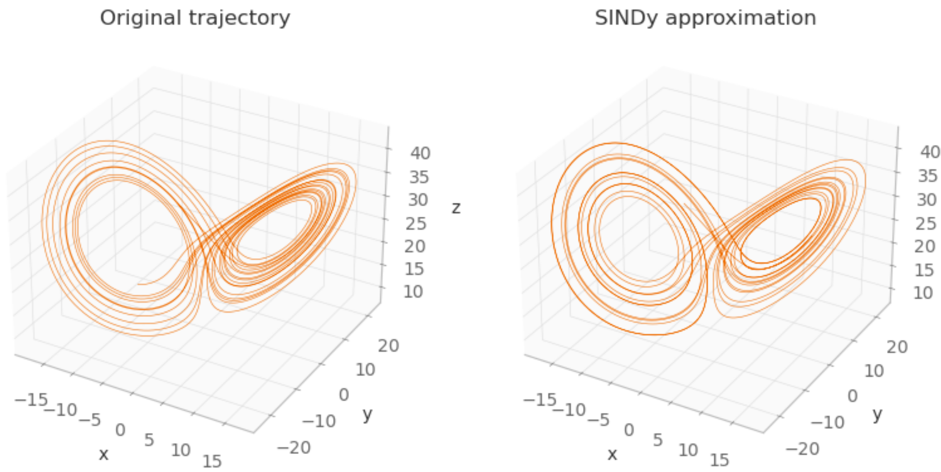


Figure 2.2: Origianl Lorenz system and SINDy approximation.

To get a view over everything a schematic view of the SINDy method demonstrated on the Lorenz equations is illustrated in figure 2.3.

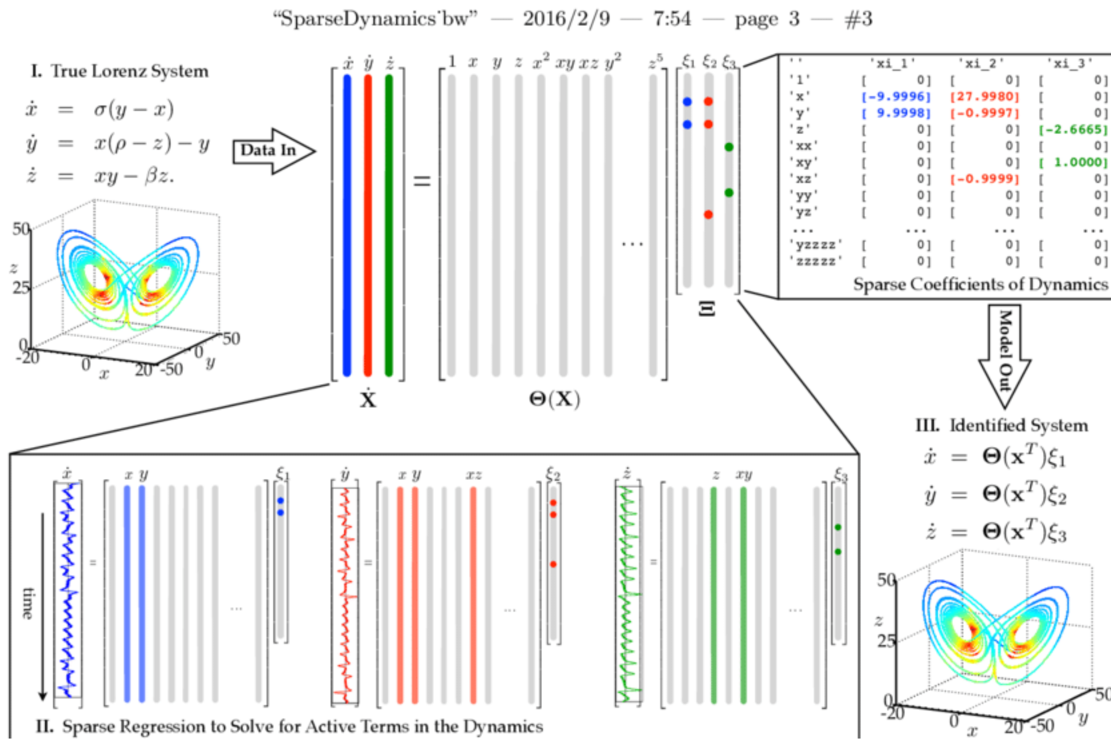


Figure 2.3: Schematic of SINDy algorithm [17].

2.2 PySINDy - A robust Python package for SINDy

This package has been used to compare the results between the implemented SINDy and the package to see if the results are similar.

The PySINDy package is an open source python package made to discover governing dynamical systems models from data, just like the SINDy algorithm. This package is made for researchers and practitioners alike, which makes it accessible to inexperienced practitioners while also usefull for more advanced users. PySINDy includes a number of different options, which means it can be heavily customized to your needs [16].

In 2022 there was a major update to the PySINDy package which implemented several advanced feature that will enable the discovery of more general differential equations from noisy and limited data [11]. They have extended the library of candidate terms for indetification of actuated systems, partial differential equations and implicit differential equations. To enforce and provide inequality constrains and stability a range of new optimization algorithms has also been added [11]. In figure 2.4 there is a summary of what features the PySINDy package has.

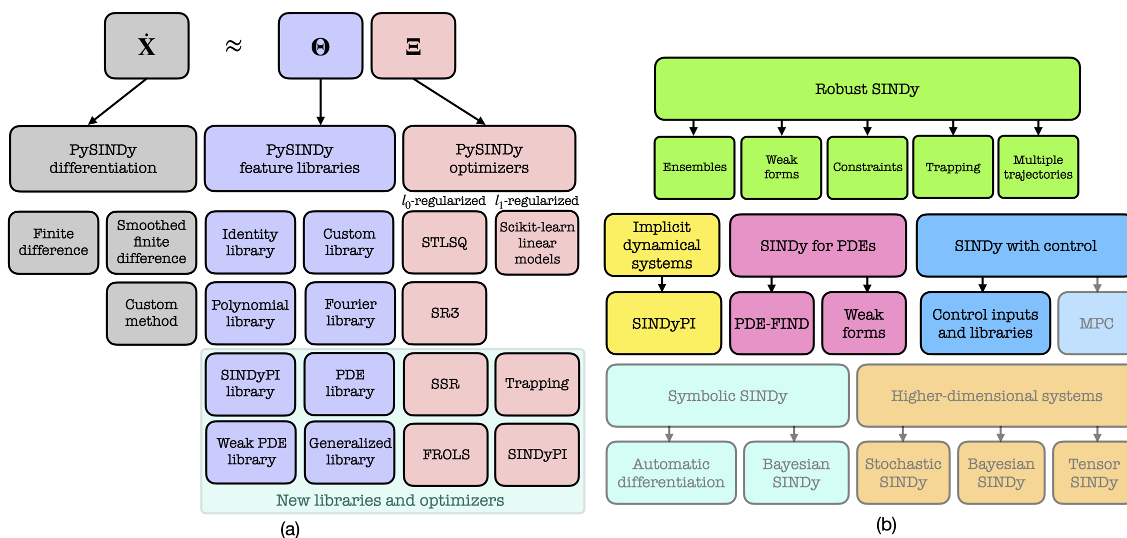


Figure 2.4: Overview of the PySINDy package [11].

In figure 2.5 there is a flow chart of how a user systematically take the right decisions for a specific scientific task [11].

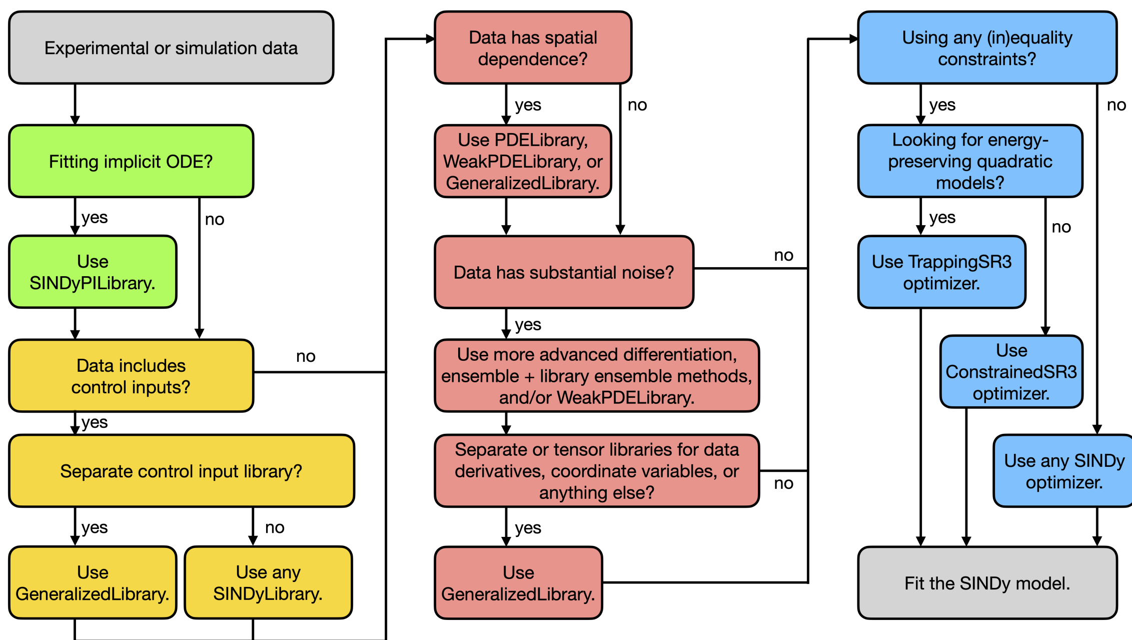


Figure 2.5: Flow chart [11].

Chapter 3

Construction

3.1 SINDy

To implement a toy example for the SINDy method is an important step for the process and a good way to start the project. The first construction of the SINDy algorithm was to recreate the Lorenz system [18]. One using the integrated SINDy method in pySindy and one where the users defines and creates everything.

3.2 Lorenz equation

The data is generated in python, and has no noise. The data has been generated using equation (2.2), (2.3) and (2.4). So this is a theoretical experiment because of the lack of noisy data [18]. Figure 2.1 shows the trajectory of the generated data.

3.3 Covid 19 example

The data for the Covid-19 example has been collected from github and this link, `https://github.com/CSSEGISandData/COVID-19`. The data are called "COVID-19 Data Repository by the Center for Systems Science and Engineering (CSSE) at Johns Hopkins University". They have gathered the data from different sources for example the Worlds Health Organization (WHO) [7]. Figure 3.1 shows the data of infected people

in Afghanistan.

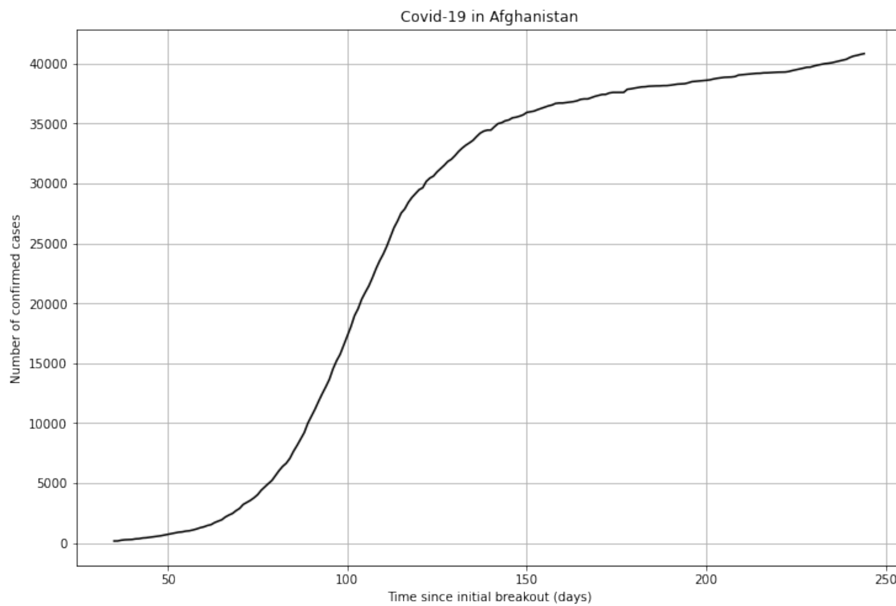


Figure 3.1: Covid data from Afghanistan.

3.4 Petroleum data

All the data used in this experiment for the platforms are downloaded from the Diskos National Data Repository (NDR) which is Norway's national data repository for petroleum data [1]. The Norwegian Petroleum Directorate and oil companies on the Norwegian continent has collaborated to create Diskos, and NPD leads the joint venture. The data that are stored in the NDR are seismic data, well data and production data [1]. The data that has been used for this project is the production data. The data repository this data has been gathered has been closed from 1. June, and Diskos 2.0 went live 17. April [2].

3.4.1 Statfjord Øst

Statfjord Øst is an oil field that was approved for production the 11. December 1990. The production then started the 24. September in 1994, and have been in production since. The companies that have the rights of this field currently are Equinor, Petoro, Vår Energi, INPEX Idemitsu and Wintershall Dea. The current operating company

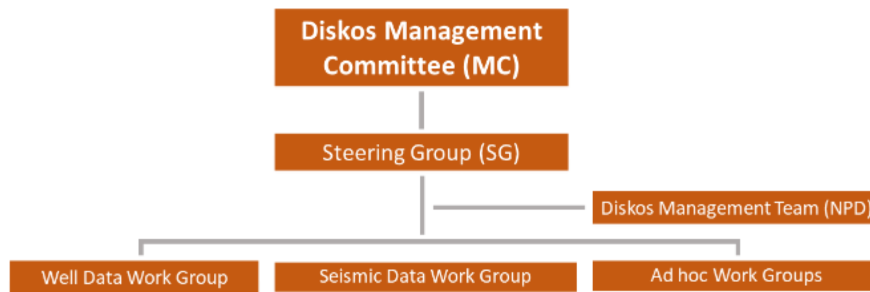


Figure 3.2: Diskos organization map [1].

are Equinor [5]. In 2020 there was a big investment of NOK 3 billion to extend the production of Statfjord Øst towards 2040, and the recovery rate has increased from 56 percent to 62 percent [9]. The wells will be drilled in 2022-2024, and the extended production will start in 2024 [9].

Figure 3.3: Statfjord C-platform [9].

The oil field are located in the northern part of the North Sea, and Statfjord Øst is seven kilometers north-east from Statfjord. The water depth are between 150 and 190 meter [5]. The field has been built out with two seabed frameworks for production and one for water injection which is connected to the Statfjord C facility. In addition to this there has been drilled two productions well from Statfjord C as well [5]. In 2024 there will be an increased production from Statfjord Øst to the Statfjord C platform according to new investments from December 2022 [9].

The depth of the reservoir are 2400 meters, and the oil are from sandstone of Middle

Jurassic age in the Brent group. The well flow goes in two pipelines to Statfjord C facility where it is processed, stored and exported. Tankers come and collect the oil while the gas is exported through the Tampen Link and Far North Liquids and Gas System (FLAGS) pipeline to the UK [5]. In figure 3.4 below, there is a image where Statfjord Øst oil field is located on the map

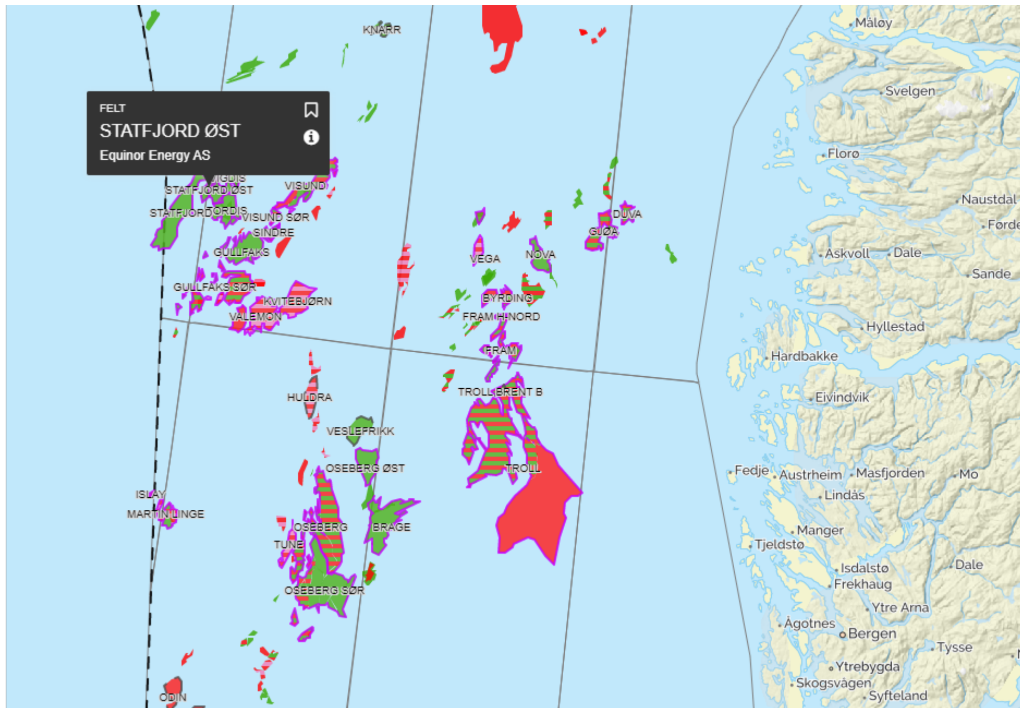


Figure 3.4: Statfjord Øst field. Map taken from Norsk Petroleum. `https://www.norskpetroleum.no/interaktivt-kart-og-arkiv/interaktivt-kart/`. (accessed: 14.05.23).

Statfjord Øst is provided with three sub-sea templates, K, L and M. The K structure is for water injection, while the other two handles oil production. It has been common to utilize water injection in oil fields and in 2019, 41 out of 78 fields on the Norwegian Continental Shelf (NCS) utilizes it. Water injection is used to pressure maintenance of the reservoir and displacing the oil from injection towards production wells [6]. On Statfjord Øst there are 20 registered production wells and four water injection wells registered. First it produced with water injection, but has changed to pressure depletion in later years [5].

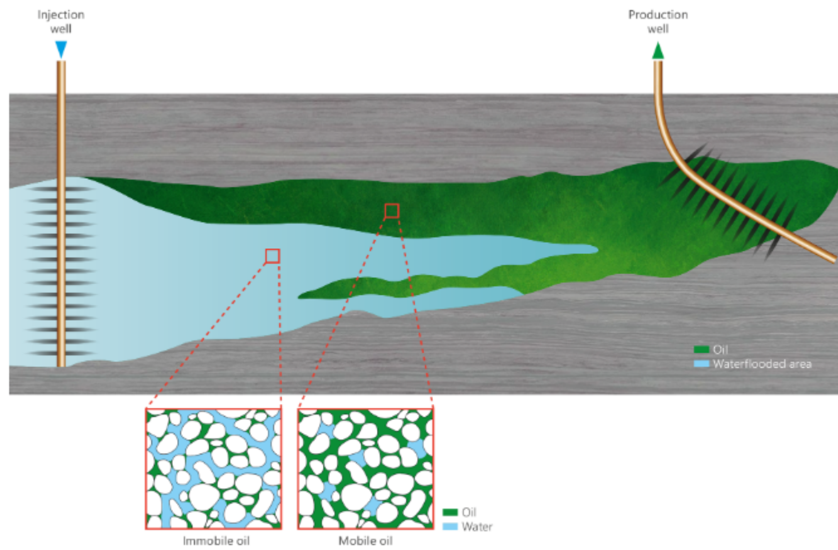


Figure 3.5: Model of water injection of a well [6].

Figure 3.6 shows the production data from three wells on the Statfjord Øst field.

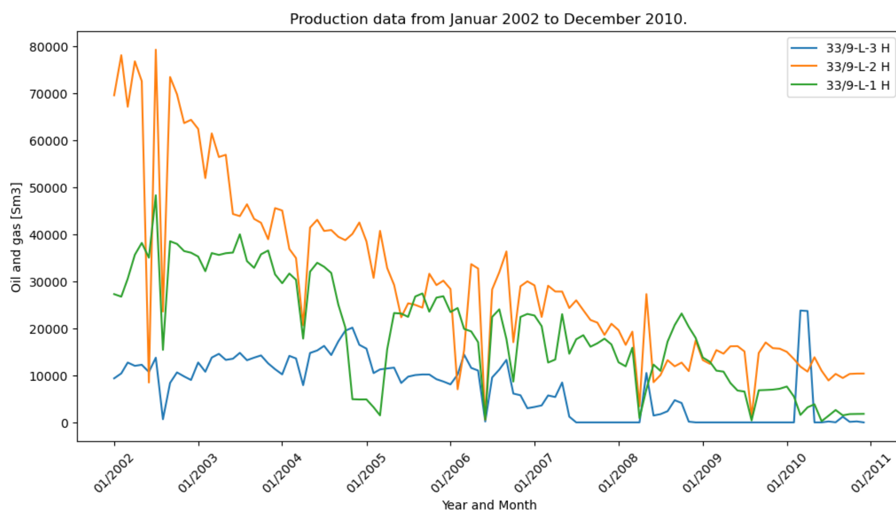


Figure 3.6: Production data from Statfjord Øst.

3.4.2 Draugen

Draugen is an oil field where the production started in 1993, and it's located in the southern part of the Norwegian Sea. It was discovered in 1984, and then approved in 1988. The production comes from to formations and the main reservoir is in sandstone of Late Jurassic age in the Rogn Formation. The western reservoir of the field produces

from sandstone of Middle Jurassic age in the Garn Formation. They both have good reservoir quality [4]. The operators for this field are Petoro AS, OKEA ASA and M Vest Energy AS, with Petoro and OKEA as the biggest operators with over 90 percentages of the total shares [13].

Figure 3.7: Draugen platform [12].

This field uses water injection and water from the base formations to produce with pressure. In 2020 Draugen had a production reliability of 99 percent [13]. In figure 3.8 it is possible to see where the Draugen oil field is located on a map.

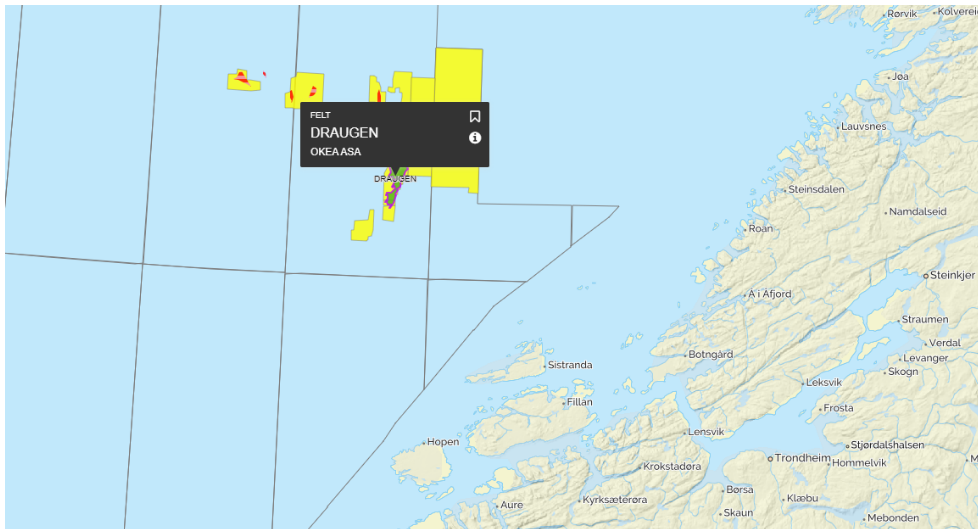


Figure 3.8: Draugen field. Map taken from Norsk Petroleum. https://www.norskpetroleum.no/interaktivt-kart-og-arkiv/interaktivt-kart/. (accessed: 14.05.23)

The current status of the production on Draugen is that there need to be made investments to the facilities to maintain the forecasted production profile [4]. The newest well was drilled in 2015, and the production started in 2017 on this one [13].

Figure 3.9 shows the production data of three wells at the Draugen field. These data has been utilized in the SINDy method.

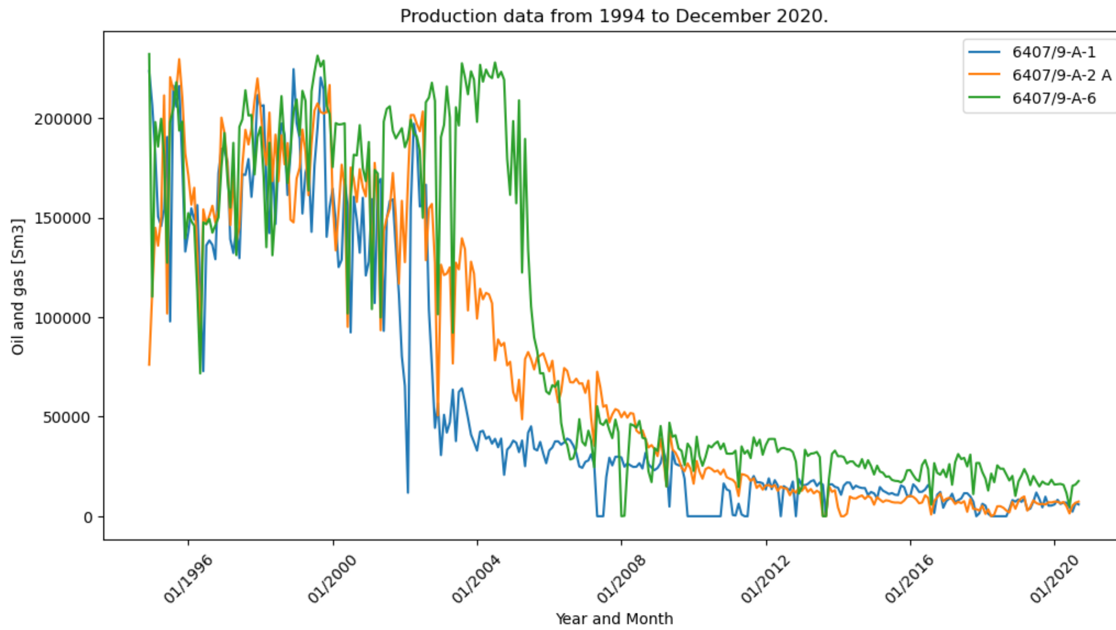


Figure 3.9: Production data from Draugen.

Chapter 4

Implementation

In this chapter we will explain the implementation of the SINDy algorithm for the Lorenz system. The code is inspired by Bea Stollnitz [18] and Steve L. Brunton and J. Nathan Kutz [14].

Then based of the work done for the Lorenz system the code for the SINDy method for the well data from DISKOS was implemented [1]. There was made some changes to how the derivative data was found, filtering was added and some other changes. The main different from the Lorenz system were that the data was real data with some noise and a spiky dataset, compared to the data from Lorenz system with very smooth generated data.

The code and the data sets can be found in a Github repository using this link, `https://github.com/SanderSondeland/Master_thesis`.

4.1 Implementation used for the Lorenz system

First we will have a look on how the system was implemented to solve the Lorenz system. The code can be found in appendix A.1 or in the Github repository.

As much as possible in this notebook has been made as functions where the important parameters and data are passed in. It is easy to make changes to the parameters and other data instead of changing it inside the functions.

The first part of the notebook is to import all the packages needed to get everything to work.

```
import numpy as np
import logging
from scipy.integrate import solve_ivp
from derivative import dxdt
from typing import Tuple
import matplotlib.pyplot as plt
```

Listing 4.1: Imported packages.

The Lorenz equations are put in, so it can be used to generate the data, u. When the data is generated in generate_u() the first three values are set to -8, 8 and 27.

```
def lorenz(_: float, u: np.ndarray, sigma: float, rho: float, beta:
    float) -> np.ndarray:

    x = u[0]
    y = u[1]
    z = u[2]
    dx_dt = sigma * (y - x)
    dy_dt = x * (rho - z) - y
    dz_dt = x * y - beta * z

    return np.hstack((dx_dt, dy_dt, dz_dt))
```

The solve_ivp is used to generate the data, u. Solve_vip is a function in the scipy-package which numerically integrates a system of ordinary differential equations given an initial value [15]. t_spand defines the interval of the integration. y0 is set to the initial values, which are u0 which is an array of three numbers [-8, 8, 27].

```
def generate_u(t: np.ndarray) -> np.ndarray:
u0 = np.array([-8, 8, 27])
result = solve_ivp(fun=lorenz,
                   t_span=(t[0], t[-1]),
                   y0=u0,
                   t_eval=t,
                   args=(SIGMA, RHO, BETA))
u = result.y.T
```

```
9       return u
```

To find the derivative of u, a function called calculate_finite_difference_derivatives was constructed and the function dxdt from derivative package was used. The differentiation kind was set to finite difference with central differencing using 3 points. The derivative package are actually a part of the pySINDy-package [11].

```
1       uprime = dxdt(u.T, t, kind="finite_difference", k=1).T
```

Listing 4.2: Derivative-function

We use both of these functions to generate u and u_prime. In this function t0, dt and tmax is decided.

```
1  def generate_data() -> Tuple[np.ndarray, np.ndarray]:
2      """ Generates data u, and calculates its derivatives.
3      """
4      t0 = 0.001
5      dt = 0.001
6      tmax = 100
7      n = int(tmax / dt)
8      t = np.linspace(start=t0, stop=tmax, num=n)
9
10     # Step 1: Generate data u.
11     u = generate_u(t)
12
13     # Step 2: Calculate u' from u.
14     uprime = calculate_finite_difference_derivatives(u, t)
15
16     return (u, uprime)
```

Listing 4.3: Data generator for Lorenz System

The library of candidate terms consists of polynomial terms and can go up to fifth polynomial order. The input of the library function is the dataset and polynomial_order. Parts of the library function, create_library() is shown below. The whole function is not shown because it repeats itself by building out the polynomial order. The output of this function is a matrix called theta with many candidate terms. The size of the output matrix, theta, is decided by the polynomial order and the size of input data.

If the polynomial_order = 2 and the input data, u, is a matrix that consists of three dimensions, x, y and z, then the terms would be 1, x, y, z, x^2, xy, xz, y^2, yz and z^2.

```
def create_library(u: np.ndarray, polynomial_order: int) -> np.ndarray:
    """Creates a matrix containing a library of candidate functions.
    """
    (m, n) = u.shape
    theta = np.ones((m, 1))

    # Polynomials of order 1.
    theta = np.hstack((theta, u))

    # Polynomials of order 2.
    if polynomial_order >= 2:
        for i in range(n):
            for j in range(i, n):
                theta = np.hstack((theta, u[:, i:i + 1] * u[:, j:j + 1]))
    .
    .
    .
    return theta
```

Listing 4.4: The library generator.

After the library has been created the next step is to find the Ξ matrix. We have to solve equation 2.14 using the least-square algorithm, the same function used in generate_u(). uprime are for the input for the differentiated matrix.

```
xi = np.linalg.lstsq(theta, uprime, rcond=None)[0]
```

Listing 4.5: The least-square algorithm to solve equation 2.14.

$\dot{\mathbf{X}}$ and Θ is known and will be used to find Ξ matrix. After the matrix has been found then the sparsity will be added. How much sparsity will be added is decided by max iterations, which decides how many times the algorithm will be run and try to zero out small terms. And the threshold will effect which values will be set to zero. If threshold has been set to 0.001 then values under 0.001 will be set to zero in the Ξ matrix.

```python
        for _ in range(max_iterations):
        small_indices = np.abs(xi) < threshold
        xi[small_indices] = 0
        for j in range(n):
            big_indices = np.logical_not(small_indices[:, j])
            xi[big_indices, j] = np.linalg.lstsq(theta[:, big_indices],
                                                 uprime[:, j],
                                                 rcond=None)[0]
```

Listing 4.6: This algorithm is adding sparsity to the Ξ-matrix.

These are added together to mare the function calculate_regression(), and which uses theta, uprime, threshold and max_iterations as input values. The function will then return a sparse Ξ-matrix.

Compute_trajectory is the code that returns the approximated signal based on u_prime, Θ and Ξ. The input for the function is u0, Ξ and the polynomial order.

```python
    def compute_trajectory(u0: np.ndarray, xi: np.ndarray,
   polynomial_order: int) -> np.ndarray:
    t0 = 0.001
    dt = 0.001
    tmax = 100
    n = int(tmax / dt + 1)

    t = np.linspace(start=t0, stop=tmax, num=n)
    result = solve_ivp(fun=lorenz_approximation,
                       t_span=(t0, tmax),
                       y0=u0,
                       t_eval=t,
                       args=(xi, polynomial_order))
    u = result.y.T

    return u
```

Listing 4.7: This is the which computes the approximated u from the SINDy method.

This computes the approximated trajectory, and are the final step in the SINDy method. In SciPy's solve_ivp package fun is the right hand side of the system, and here there has been used a self made right hand system to fit what's needed [15].

```
def lorenz_approximation(_: float, u: np.ndarray, xi: np.ndarray,
                         polynomial_order: int) -> np.ndarray:
    theta = create_library(u.reshape((1, 3)), polynomial_order)
    return theta @ xi
```

Listing 4.8: This function returns the right hand side of equation 2.14.

This code returns the right hand side of equation 2.14 to be used in SciPy's solve_ivp function to find the approximated trajectory.

The input for this is u0 and xi, and those are being used in the create_library function to create a new theta. The new theta has been reshaped because u0 has been reshaped in the input. That is becuase the matrix multiplication can be done with the xi-matrix. And theta*xi is equal to $\dot{\mathbf{X}}$ from equation 2.14, then the compute_trajectory can be used to solve and find the approximated trajectory.

The only remain is to illustrate the trajectories and compare the results from the SINDy algorithm. For graphing the trajectories a function using matplot.pyplot was made and called graph_results().

```
axis3d = fig.add_subplot(1, 2, 2, projection="3d")
x = u_approximation[0:sample_count, 0]
y = u_approximation[0:sample_count, 1]
z = u_approximation[0:sample_count, 2]
axis3d.plot3D(x, y, z, orange, linewidth=0.4)
axis3d.set_title("SINDy approximation")
style_axis3d(axis3d)
```

Listing 4.9: Code for plotting the approximation in 3D.

This code is the main part of the graph_results function, and the code is utilizing the matplotlib package to get this three dimensional plot.

The flowchart in figure 4.1 shows which functions runs and what parameters has to be set to different values.

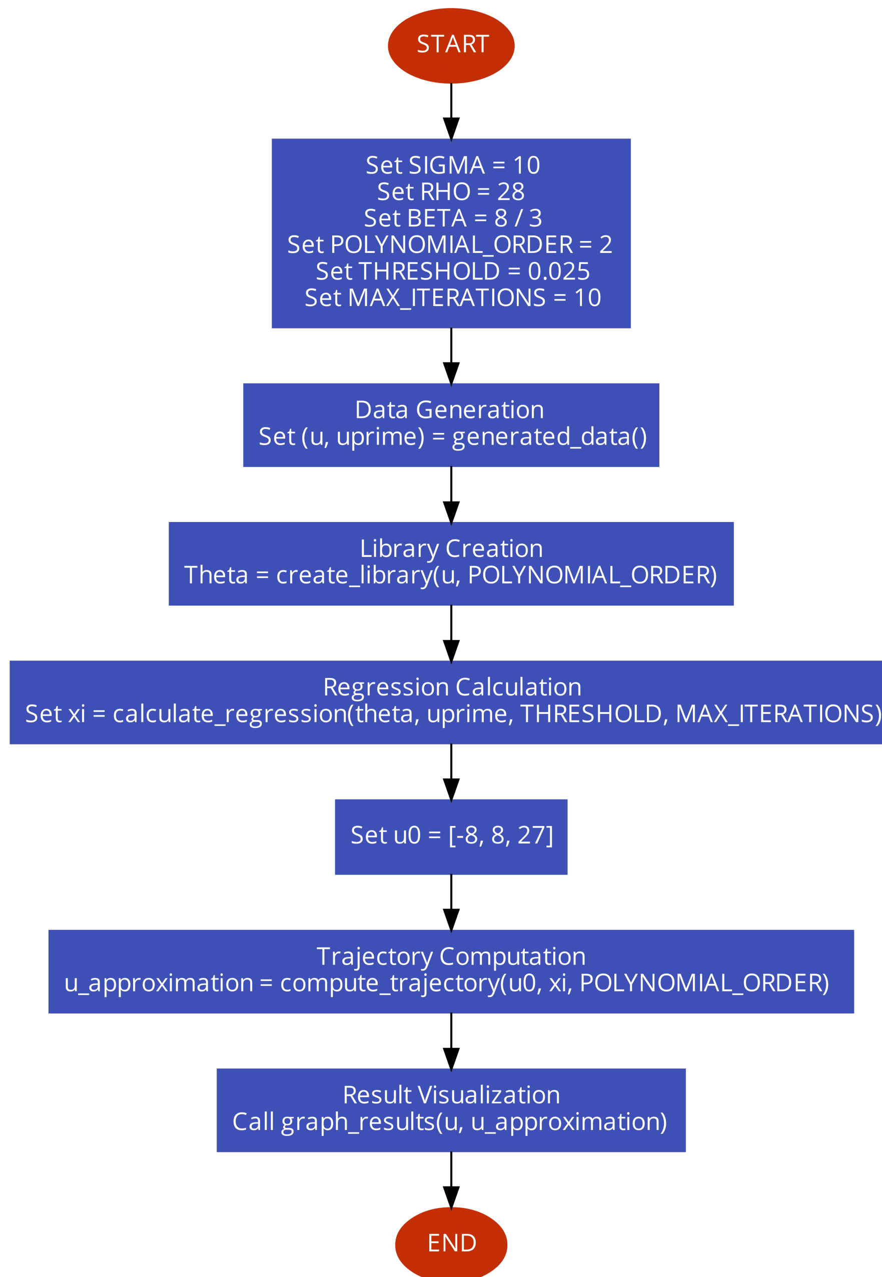


Figure 4.1: Flowchart for the Lorenz system.

4.2 Implementation for Draugen and Statfjord Øst

The framework from the implementation used for the Lorenz system was worked on more to fit the data from Draugen and Statfjord Øst. The algorithm has been changed as well to fit this data better. The code can be found in appendix A.4 or in the Github repository.

There has been more packages imported in this implementation because of the use of filters and dates in the dataset. And some of the extra packages was to load the dataset from Github.

```
import numpy as np
import pandas as pd
import requests
import io
import datetime
from scipy.integrate import solve_ivp
from derivative import dxdt
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
from scipy.ndimage import median_filter
```

Listing 4.10: Packages that were imported.

The datasets was downloaded via Github, and the link to the raw file was used to download the dataset into Visual Studio Code. This has to be done each time the notebook is restarted, and then the link to the raw data has gone out of date so it has to be copied from Github and pasted into Visual Studio Code again. The link is pasted in where it says url =' '.

```
url = ''
download = requests.get(url).content
data = pd.read_csv(io.StringIO(download.decode('utf-8')))
```

Listing 4.11: Code for downloading the dataset from Github.

The next step is to gathered the data that is needed and remove the rest of the dataset that we do not need for this project. The production data is gathered one time a month, and we then want the production of oil and gas. Then they are added together to create a new column of the total production of the well. Oil and gas has to have the same units so the gas data has to be divided by 1000 to get the same units for both. Groupby has been used to group by period, which is now YYYY-MM. The example below is for well 6407/9-A-1 in the Draugen field, but it is similar for every well. It is just the name that has to be changed due to similarities on the data sets downloaded from Diskos.

```
x_data = data1[data1['name'] == '6407/9-A-1']
x_data.loc[:, 'period'] = x_data['year'].astype(str) + '-' + x_data['month'].astype(str)
x_data.loc[:, 'tot_prod'] = (x_data['oil'] + x_data['gas']/1000).round(1)
x_data = x_data.groupby('period').sum()
x_data
```

Listing 4.12: Getting data for a specific well and adding oil and gas production together.

The datetime package was used to use the dates by defining the start year and end year. Then the months that is not in the data set are removed in line two below.

```
dates = [datetime.datetime(year=int(year), month=int(month), day=1) for year in range(1994, 2021) for month in range(1, 13)]
dates = dates[11:-3]
```

Listing 4.13: Using datetime to get the timeinterval of the production.

Filtering these data are important because the signal is very spiky, and they have to be smoothed out both before and during differentiation. If this is not done the derivative gets really large values due to some of the spikes in the production. To smooth out the data before the differentiation a median filter was used. The window size defined how large of an effect the filter would make, and a larger window size would mean more smoothing. Then it is to find a good window size for the data and to make sure that to much information is not lost during the filtering.

```
md_x = median_filter(tot_prod_x, size=4)
md_y = median_filter(tot_prod_y, size=4)
md_z = median_filter(tot_prod_z, size=4)
```

Listing 4.14: Median_filter that was used for all production data to smooth out the data.

To differentiate the data a t-variable is needed, and that was generated out of the dates. It was also converted to a numpy array due to the needs of the derivation function.

```
i = len(dates)
t = list(range(1, i+1))
```

```
    t_np = np.array(t)
```

Listing 4.15: Defining t and t_np to use in the derivation function afterwards.

The derivative module has a function called dxdt which has been used to differentiate the data. The input in this has to be a numpy array so the total production has been set to a numpy array.

The function also has different ways to derivative the data. Here the kalman derivative has been used to smooth out the data even more. The smoothing has been set to 2.

```
    x_der = dxdt(md_x_np, t_np, kind="kalman", alpha=2)
    y_der = dxdt(md_y_np, t_np, kind="kalman", alpha=2)
    z_der = dxdt(md_z_np, t_np, kind="kalman", alpha=2)
```

Listing 4.16: Derivation with smoothing using the kalman method.

Another derivative method that could be used to smooth the data is the finite difference method.

```
    x_der = dxdt(md_x_np, t_np, kind="finite_difference", k=2)
    y_der = dxdt(md_y_np, t_np, kind="finite_difference", k=2)
    z_der = dxdt(md_z_np, t_np, kind="finite_difference", k=2)
```

Listing 4.17: Derivation with smoothing using the finite difference method.

The results from listing 4.16 and 4.17 are displayed in figure 4.2 and 4.3.

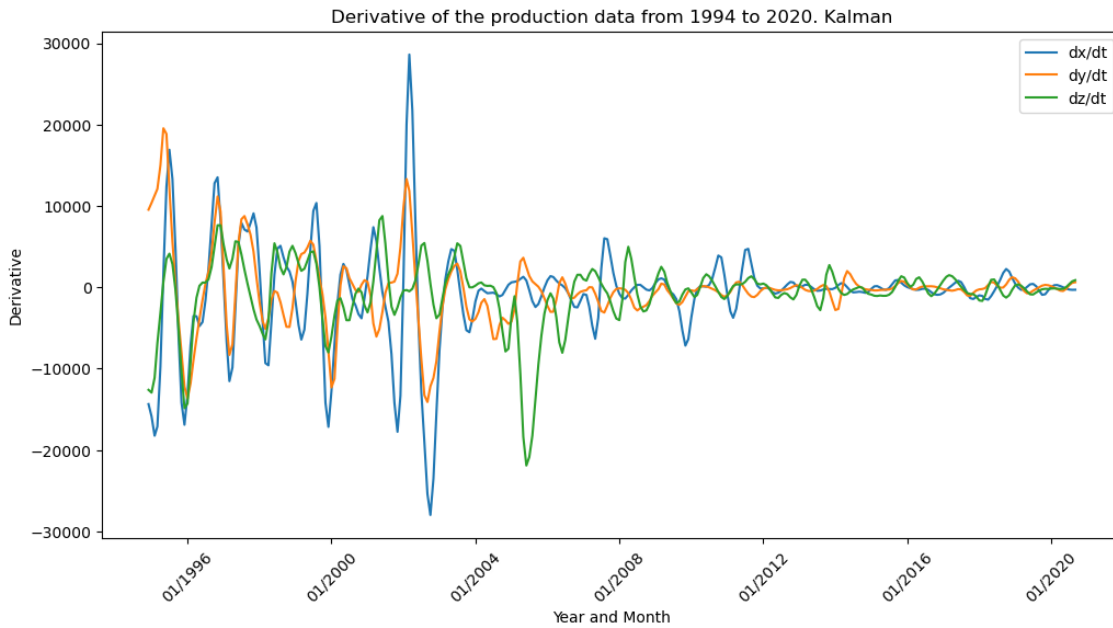


Figure 4.2: Derivaitve with smoothing using the kalman method on data from Draugen.

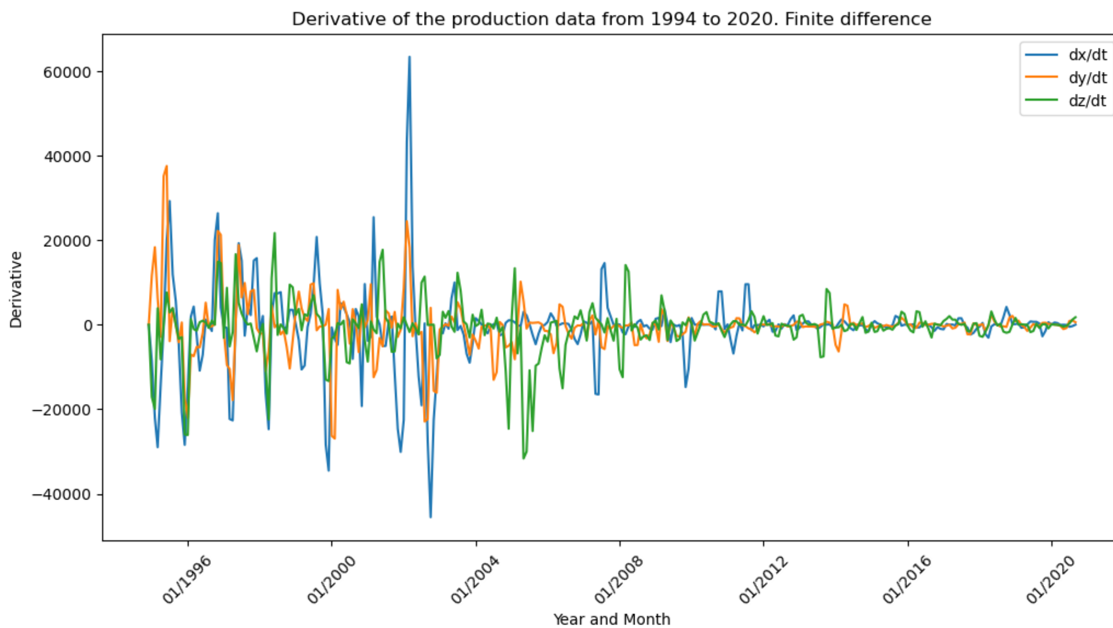


Figure 4.3: Derivaitve with smoothing using the finite difference method on data from Draugen.

In figure 4.2 and 4.3 two plots of the differentiated data is shown. The kalman method was utilized in this project due to the spikes in the finite difference method. Compared to the kalman method in figure 4.2 the values of the spikes are much higher, and the

smoothest output was selected, which was the kalman method. The highest peak from the finite differnce method is over 6000, but for the kalman method it is just about 3000.

Two matrices are created with the filtered data and the derivated data. X, y and z data has been put toghether to form a 310 x 3 matrix so it can be used in the SINDy method. The same thing has been done to the derivated data as well, now we have to matrices to use in the SINDy method.

```
mat = np.zeros((310,3))

mat[:, 0] = md_x
mat[:, 1] = md_y
mat[:, 2] = md_z
```

Listing 4.18: Creating a matrix for the filtered data.

The library is similar to the implementation for the Lorenz system, but two new features has been added. Now it is possible to add one sin and one cosine function and two exponential functions as well. If use_trig is set to True then the sine and cosine function will be added at the end, and if use_exp is set to true the exponential functions are added at the very end of the library. The exponential equation that will be added is e^{-u} and e^{-u^2}. Then the code was looking like this afterwards.

```
def create_library(u: np.ndarray, polynomial_order: int, use_trig: bool, use_exp: bool) -> np.ndarray:
    """Creates a matrix containing a library of candidate functions.
    """
    (m, n) = u.shape
    theta = np.ones((m, 1))

    # Polynomials of order 1.
    theta = np.hstack((theta, u))
    .
    .
    .
    if use_trig:
        for i in range(1, 11):
```

```
14                 theta = np.hstack((theta, np.sin(i * u), np.cos(i * u))
   )
15
16         if use_exp:
17             for i in range(n):
18                 theta = np.hstack((theta, np.exp(-u[:, i:i+1]), np.exp
   (-u[:, i:i+1]**2)))
19
20     return theta
```

Listing 4.19: New library function with sine cosine and exponential functions.

After theta has been generated the next step is to find Ξ -matrix. The way to find the Ξ -matrix is the same as for the Lorenz system, where the least-square algorithm is used to solve the main equation for the SINDy method, which is equation 2.14. The first Ξ -matrix that is found is not sparse.

The same method as in section 4.1 was utilized to add sparsity to the Ξ -matrix. How sparse the matrix will be afterwards are decided by the threshold value and can also be affected by max iterations. Max iterations decides how many times the algorithm will run to find the sparsest solution.

```
1  def calculate_regression(theta: np.ndarray, uprime: np.ndarray,
2                          threshold: float, max_iterations: int) -> np.
   ndarray:
3      # Solve theta * xi = uprime in the least-squares sense.
4      xi = np.linalg.lstsq(theta, uprime, rcond=None)[0]
5      n = xi.shape[1]
6
7      # Add sparsity.
8      for _ in range(max_iterations):
9          small_indices = np.abs(xi) < threshold
10         xi[small_indices] = 0
11         for j in range(n):
12             big_indices = np.logical_not(small_indices[:, j])
13             xi[big_indices, j] = np.linalg.lstsq(theta[:, big_indices],
    uprime[:, j], rcond=None)[0]
14
```

```
15     return xi
```

Listing 4.20: These functions finds the Ξ-matrix and then add sparsity.

After the sparse Ξ -matrix has been found the next is to make an approximation. Once again the same method as in section 4.1 has been used to make a prediciton. This method is divided in two different functions. In listing 4.7 and 4.8 it's displayed how the code has been implemted.

Chapter 5

Results

Different parameters, data and wells has been tested with the SINDy method. The PySINDy package has also been utilized to compare the results from PySINDy to the code inspired by Data-Driven Science and Engineering [14].

The SINDy algorithm was first made for the Lorenz system with inspiration from the book and Bea Stollnitz and her blog which is also inspired by Data-Driven Science and Engineering from S. L. Brunton and J. N. Kutz [14]. Then this code was worked with more to fit the data from Diskos and the wells [3].

In chapter 2.1 the SINDy method used on the Lorenz system was used to give a better understanding of the algorithm and show how the algorithm work on dynamical systems.

5.1 Covid example

During covid a lot of data was gathered from many different countries around the world. Here we will look at an easy example from Afghanistan and see how SINDy approximated the confirmed infected people compared to the real data.

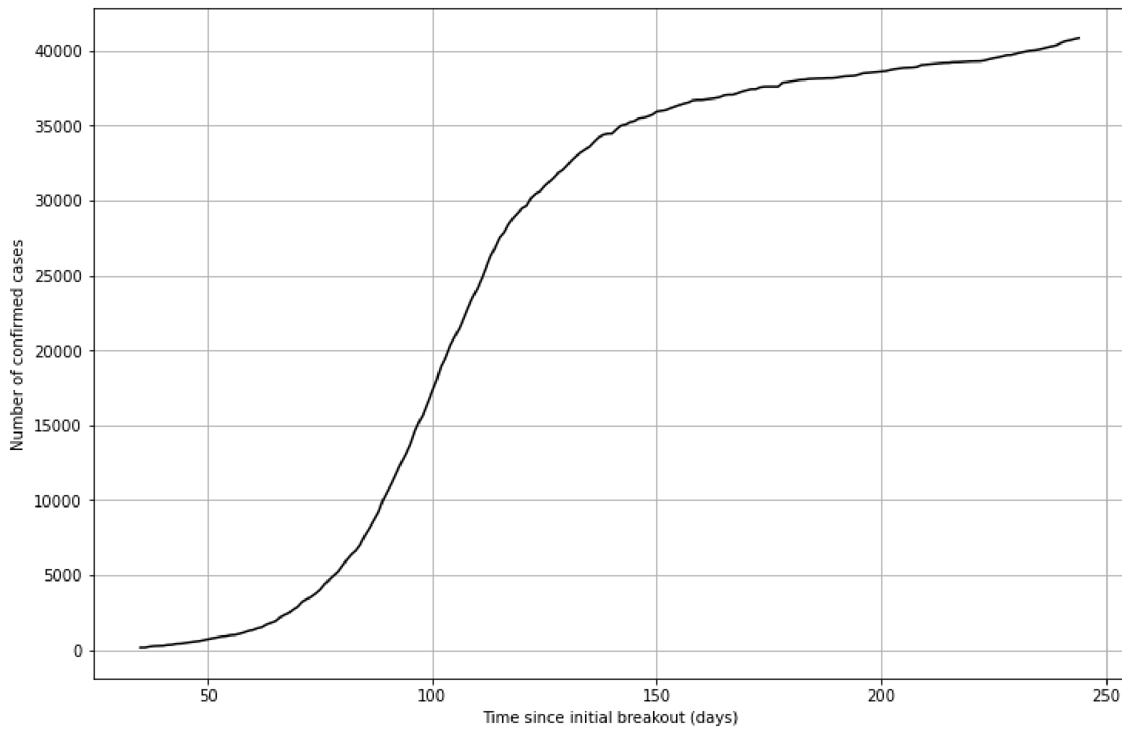


Figure 5.1: Covid data from Afghanistan.

The parameters used for this test was a higher polynomial order, and a low threshold value. All the parameters used in this test is listed below.

Polynomial order = 3

USE TRIG = False

USE EXP = False

Threshold = 0.000001

Max iterations = 10

Results

Then the Ξ -matric became:

$$\Xi = \begin{bmatrix} \dot{x}: \\ 4.75e+01 \\ -3.18e-02 \\ -1.57e-06 \\ 0.00 \end{bmatrix} \begin{matrix} \\ 1 \\ x \\ x^2 \\ x^3 \end{matrix} . \quad (5.1)$$

$$\dot{x} = 47.5 - 0.0318x - 0.00000157x^2 \quad (5.2)$$

The results for this test was very promising. Even though the last value are pretty small the SINDy method came up with an approximation that matched the data really good and described the dynamics of the system.

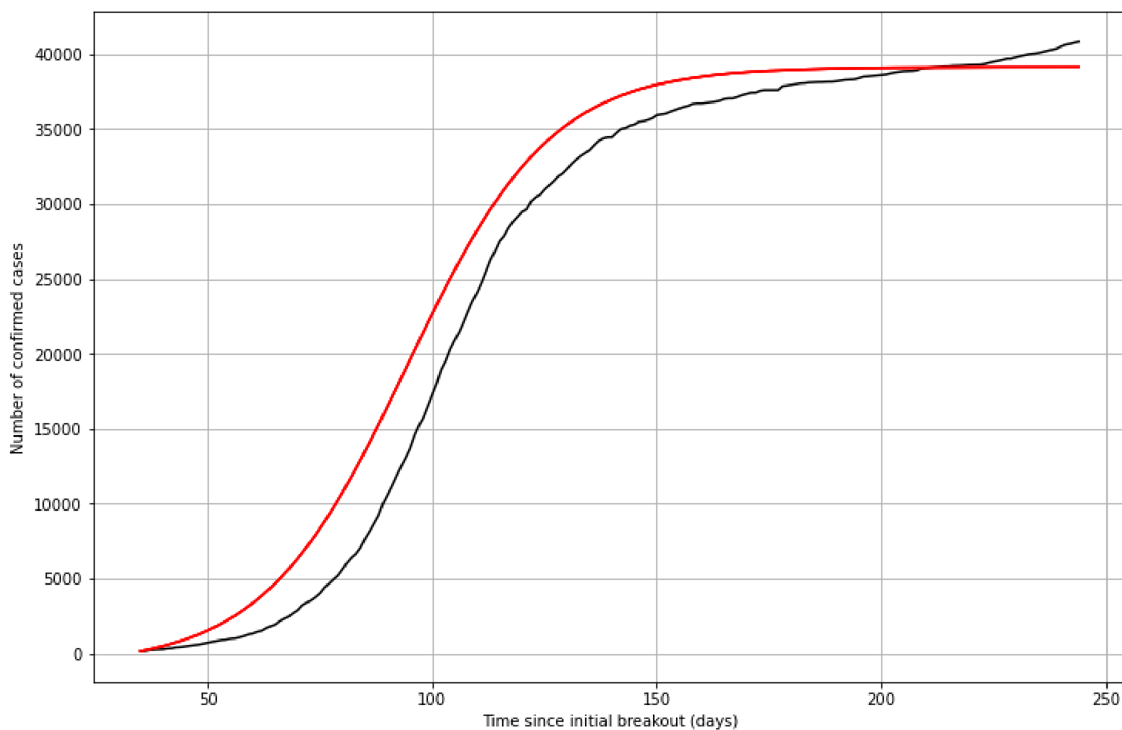


Figure 5.2: Covid results using data from Afghanistan.

Discussion

These result from the covid data is really good, and are looking promising. This is a easier and less complex problem compared to the other tests, but the results are really good. The data that has been gathered are relatively smooth, and only a small filter was added. This makes the SINDy method even better when the data are smooth and not spiky. The only filtering that has been done in this test is filtering in the differentiation process. And in the beginning of the data the data set was 1 a lot of days so the first 35 days are removed which makes the approximation better.

5.2 Statfjord Øst

In this section we will take a look at the different results using data from Statfjord Øst. In the different tests that has been done, the main thing that changes are the candidate terms in the library, and the threshold value and max iterations also varies.

In the first test of the SINDy method data from three different wells were used. The wells were located close to each other and according by the npd.no. As shown in figure 5.3, the furthest distance between the wells are between 33/9-L-1 H and 33/9-L-3 H with a distance of 19.5 meters. Meanwhile the distance from 33/9-L-3 H to 33/9-L-2 H are only 8.1 meters, and from 33/9-L-2 H to 33/9-L-1 H there are 17.1 meters apart.

Figure 5.3: Distance between the three wells during the tests for Statfjord Øst. Map taken from Norwegian Petroleum Directorade. https://factmaps.npd.no/factmaps/3_0/. (accessed: 22.05.23).

The data was as mentioned these three wells, and were set up like this:

x = 33/9-L-3 H

y = 33/9-L-2 H

z = 33/9-L-1 H

5.2.1 Test 1

The candidate library was set to a polynomial order of 2, and did not use sin or cos function and neither exponential functions in this test. The threshold was set to 0.005 and 10 max iterations. A summary of the parameters are listed below.

Polynomial order = 2

USE TRIG = False

USE EXP = False

Threshold = 0.005

Max iterations = 10

The Ξ-matrix became:

$$\Xi = \begin{bmatrix} \dot{x}: & \dot{y}: & \dot{z}: \\ -3.22e+02 & -1.43e+02 & -6.57e+02 \\ -1.66e-02 & 0.00 & -7.43e-02 \\ 0.00 & -1.28e-02 & 3.00e-03 \\ 1.70e-02 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 \end{bmatrix} \begin{matrix} \\ 1 \\ x \\ y \\ z \\ x^2. \\ xy \\ xz \\ y^2 \\ yz \\ z^2 \end{matrix} \tag{5.3}$$

$$\dot{x} = -322 - 0.0166x + 0.0170z, \tag{5.4}$$

$$, \dot{y} = -143 - 0.0128y \tag{5.5}$$

$$\dot{z} = -657 - 0.0743x + 0.003y. \tag{5.6}$$

Results

The results were promising, and the SINDy algorithm made a good approximation based of the data. From the Ξ -matrix there is no terms for the second order polynomials so the polynomial_order could be set to 1 and the results would be the same.

The results was better for y- and z-data, but that's probably because the x-data consist of a lot of zeros in the later years of production compared to the two other wells. The result on x-data is not good, and the graph is non-physical because the graph has a negative value at the end, and the production numbers can not go below zero.

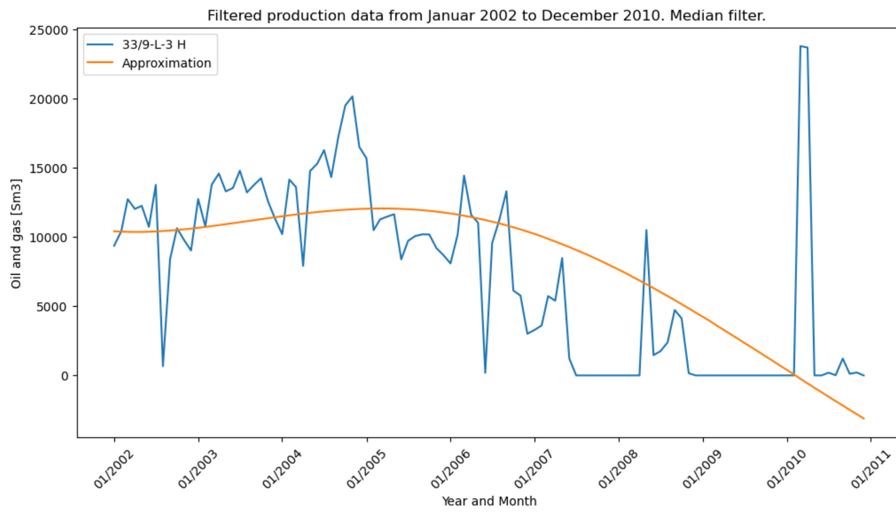


Figure 5.4: The result for x-data

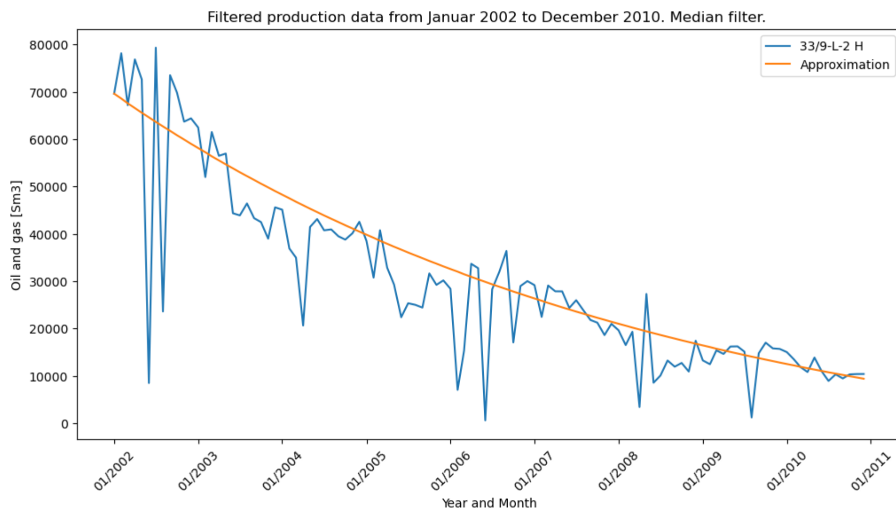


Figure 5.5: The result for y-data

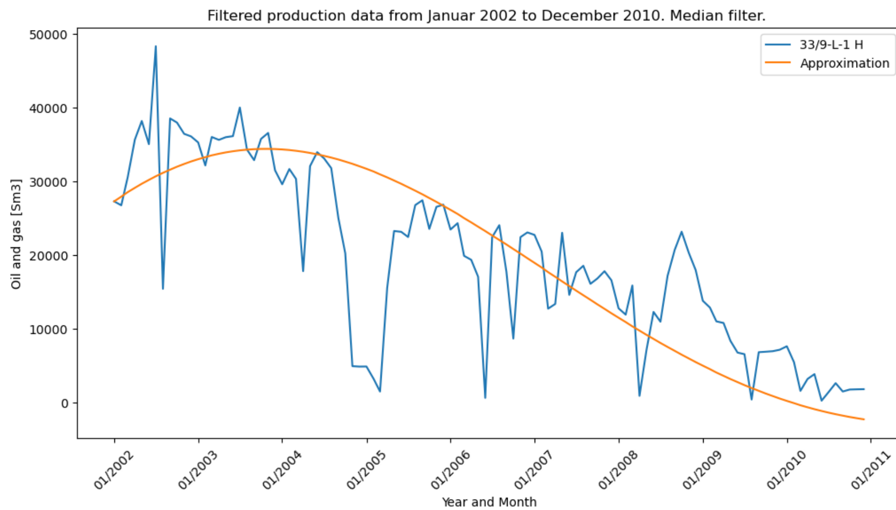


Figure 5.6: The result for z-data

Discussion of test 1

The results here are promising, and the trajectory follow the main trend of the data. It is hard for the approximation to follow completely due to a very spiky data set. The data has been filtered before differentiation and during differentiation to smooth out the data. This is because we want to decrease the big spikes because that will give us big numbers in the differentiated data.

The best results was with the y- and z-data, but that's most likely because of the amount of zero-data in the x-data set. Where y- and z-data has more consistent data compared to x-data.

5.2.2 Test 2

For the second test of Statfjord Øst the same wells has been used, but the candidate terms in the library has been changed. Now $exp(-u)$ and $exp(-u^2)$ has been added, where u is the data (x, y and z) so the library has now 6 more candidate terms. The threshold value and max iterations are still the same as before. Also the filters are still the same, and the differentiation method are also the same as test 1.

Polynomial order = 2

USE TRIG = False

USE EXP = True

Threshold = 0.005

Max iterations = 10

Then the Ξ-matrix became:

$$\Xi = \left[\begin{array}{ccc|c} \dot{x}: & \dot{y}: & \dot{z}: & \\ -4.38+e02 & -1.44e+02 & 2.05e+02 & 1 \\ -6.94e-02 & 0.00 & -1.57e-02 & x \\ 0.00 & -1.28e-02 & 3.31e-02 & y \\ 1.69e-02 & 0.00 & 0.00 & z \\ 0.00 & 0.00 & 0.00 & x^2 \\ 0.00 & 0.00 & 0.00 & xy \\ 0.00 & 0.00 & 0.00 & xz \\ 0.00 & 0.00 & 0.00 & y^2 \\ 0.00 & 0.00 & 0.00 & yz \\ 0.00 & 0.00 & 0.00 & z^2 \\ 1.01e+02 & 6.40e-01 & -7.91e+02 & exp(-x) \\ 1.01e+02 & 6.40e-01 & -7.91e+02 & exp(-y) \\ 0.00 & 0.00 & 0.00 & exp(-z) \\ 0.00 & 0.00 & 0.00 & exp(-x^2) \\ 0.00 & 0.00 & 0.00 & exp(-y^2) \\ 0.00 & 0.00 & 0.00 & exp(-z^2) \end{array}\right]. \tag{5.7}$$

$$\dot{x} = -438 - 0.0694x + 0.0169z + 101e^{-x} + 101e^{-y}, \tag{5.8}$$

$$\dot{y} = -144 - 0.0128y + 0.640e^{-x} + 0.640e^{-y}, \tag{5.9}$$

$$\dot{z} = 205 - 0.0157x + 0.0331y - 791e^{-x} - 791e^{-y}. \tag{5.10}$$

Results

The results for this test was similar to the results for test 1, in section 5.2.1. The graph for x-data is a better because the graph is physical, in test 1 the graph went below zero which is not realistic.

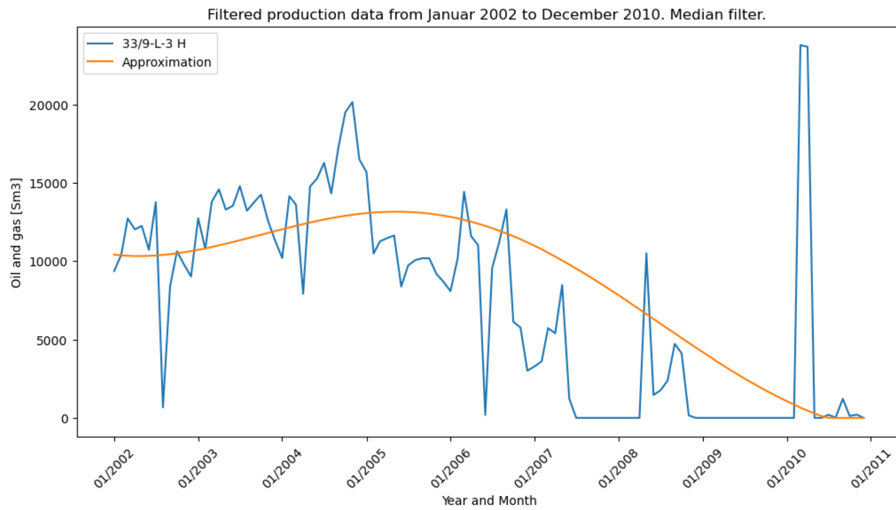


Figure 5.7: The result for x-data

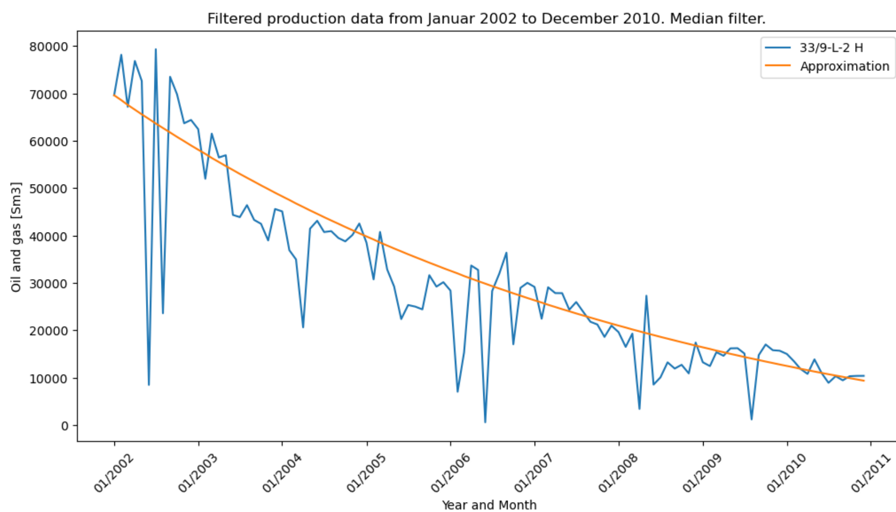


Figure 5.8: The result for y-data

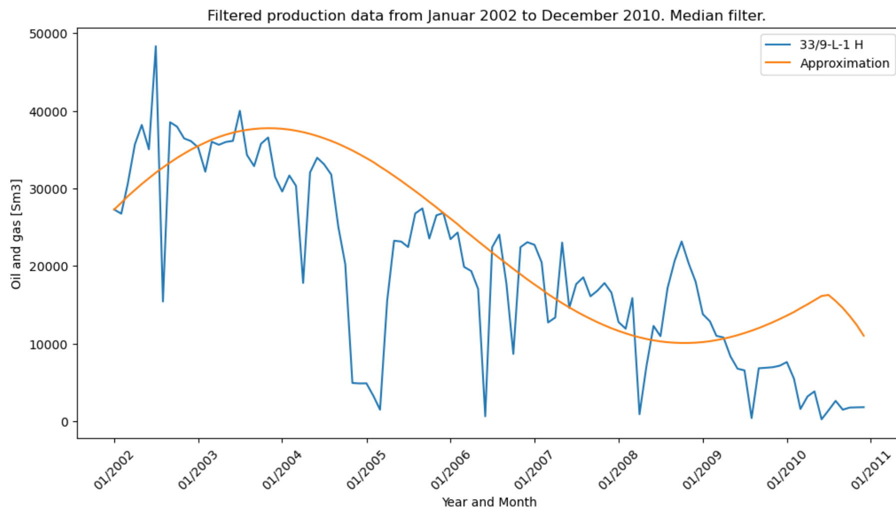


Figure 5.9: The result for z-data

Discussion of test 2

The extra terms added to the library results in more complex outcome compared to test 1 in section 5.2.1. But the results is not that much better compared to the results from test 1, but a one thing that is much better is that is not unphysical any more. The trajectory does not goes below 0 anymore.

The trajectory for x-data is very similar to test 1, but instead of going below 0 at the end it goes to 0 and stays there. For the y-data the results are similar, and the exponential terms does not add any significant changes to the trajectory.

The worst result with this change is for z-data, and that is also the approximation that changed the most as well compared to test 1.

5.2.3 Test 3

The third test using the same data from Statfjord Øst, but yet again the candidate terms in the library has changed. The use_trig is set to true, which means that sine and cosine are now included in the library as well. The terms sin(u) and cos(u), where u is the data (x, y and z), so the library has 6 more candidate terms. The threshold and max iterations are the same as previous tests.

x = 33/9-L-3 H

y = 33/9-L-2 H

z = 33/9-L-1 H

Polynomial order = 2

USE TRIG = True

USE EXP = False

Threshold = 0.005

Max iterations = 10

Then the Ξ -matrix turned out like this:

$$\Xi = \begin{bmatrix} \dot{x}: & \dot{y}: & \dot{z}: \\ -3.38+e02 & -5.18e+01 & -8.03e+02 \\ -1.44e-02 & -1.29e-02 & -5.59e-02 \\ 0.00 & -8.26e-03 & 3.69e-02 \\ 1.61e-02 & -5.93e-03 & -1.47e-02 \\ 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 \\ -1.08e+01 & 1.35e+02 & -3.19e+01 \\ 4.36e+01 & -1.01e+02 & -6.28e+02 \\ 2.29e+01 & -2.99e+02 & -7.87e+01 \\ 1.11e+02 & 1.88e+01. & 3.01e+1 \\ 1.55e+01 & 2.11e+02 & -3.96e+02 \\ 1.44e+02 & 3.17e+02 & 3.36e+01 \end{bmatrix} \begin{matrix} \\ 1 \\ x \\ y \\ z \\ x^2 \\ xy \\ xz \\ y^2 \\ yz \\ z^2 \\ sin(x) \\ sin(y) \\ sin(z) \\ cos(x) \\ cos(y) \\ cos(z) \end{matrix} . \tag{5.11}$$

$$\begin{aligned} \dot{x} = -338 - 0.0144x + 0.0161z - 10.8sin(x) + 43.6sin(y) + \\ 22.9sin(z) + 111cos(x) + 15.5cos(y) + 144cos(z), \end{aligned} \tag{5.12}$$

$$\dot{y} = -51.8 - 0.0129x - 0.00826y - 0.00593z - 13.5sin(x) - 101sin(y) - 299sin(z) + 18.8cos(x) + 211cos(y) + 317cos(z), \tag{5.13}$$

$$\dot{z} = -803 - 0.0559x + 0.0369y - 0.0147z - 31.9sin(x) - 628sin(y) - 78.7sin(z) + 30.1cos(x) - 396cos(y) + 33.6cos(z). \tag{5.14}$$

Results

The results are good, but the sine and cosine functions add 6 more terms to each equation for $\dot{x}$, $\dot{y}$ and $\dot{z}$ which makes them more complex. Also these equations are harder to understand.

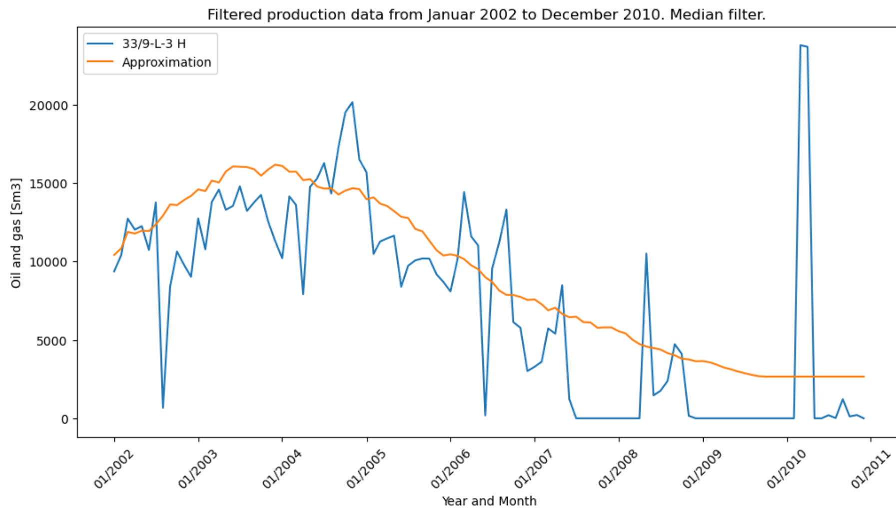


Figure 5.10: The result for x-data

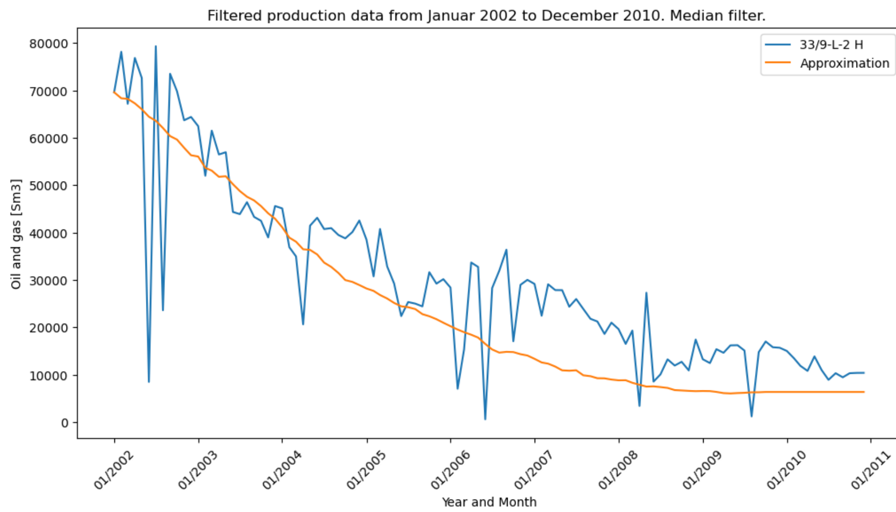


Figure 5.11: The result for y-data

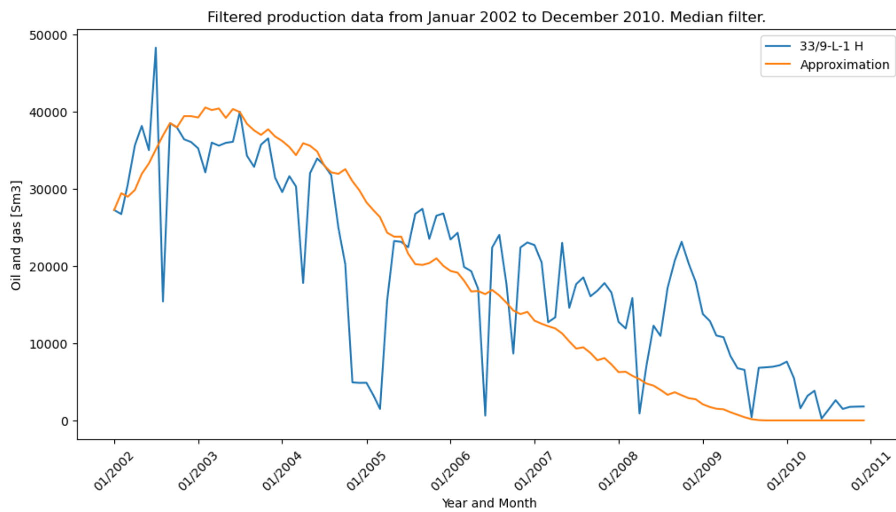


Figure 5.12: The result for z-data

Discussion of test 3

When the sine and cosine function was added it resulted in a lot more complex and advanced equations which describes the trajectories. They made it harder to understand. There was some change in the approximation, but also they were more crooked compared to the smooth trajectories on previous tests. To summarize there was nothing much gained by the more complex equations when sine and cosine was added, and the results was not better than any of the other tests.

5.3 Draugen

This section we will look at the results from applying the SINDy method on data from three different Draugen oil wells. Using different candidate libraries and different settings in the algorithm to look at the results and trajectories compared to the production data. The production data contains data from both oil and gas production.

The wells used in this section is close to eachother again. This is to see if the SINDy algorithm can find some interference between them in the output equations which describes the production.

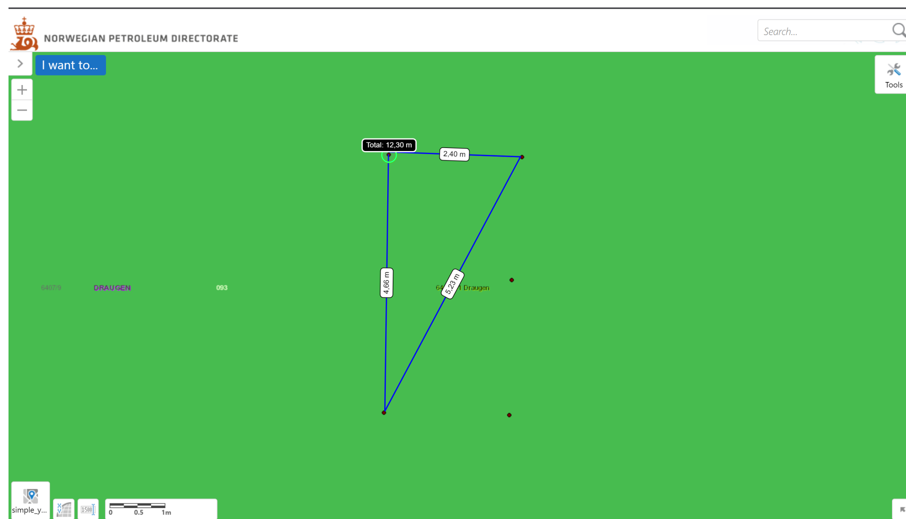


Figure 5.13: Distance between the three wells during the tests on Draugen. Map taken from Norwegian Petroleum Directorade. `https://factmaps.npd.no/factmaps/3_0/`. (accessed: 22.05.23).

In figure 5.13, it's easy to see the distance between the wells where the data is gathered from. The 6407/9-A-1 well is in the upper right hand corner and are 2.40 meters away from well 6407/9-A-2 A. This well is then just 4.66 meters away from 6407/9-A-6. Then there is 5.23 meters between 6407/9-A-6 and 6407/9-A-1.

The median filter size used on these tests are of size 4. **Unsure if this is needed.**

The data has called x, y and x like this for the tests:

x = 6407/9-A-1

y = 6407/9-A-2 A

z = 6407/9-A-6

5.3.1 Test 1

For the first test the polynomial order was set to 1, and the there was no candidate terms with exponential functions, sine or cosine. The threshold was set to 0.005 and max iterations was set to 10. All the parameters are listed below.

Polynomial order = 1
USE TRIG = FALSE
USE EXP = FALSE
Threshold = 0.005
Max iterations = 10
median filter size = 4

Then the Ξ -matrix became:

$$\Xi = \begin{bmatrix} \dot{x}: & \dot{y}: & \dot{z}: \\ 9.18e+01 & -2.84e+02 & -3.78e+02 \\ -3.18e-02 & 3.52e-02 & 0.00 \\ 5.83e-02 & -3.65e-02 & 1.88e-02 \\ -3.42e-02 & 5.65e-03 & -1.85e-02 \end{bmatrix} \begin{matrix} \\ 1 \\ x \\ y \\ z \end{matrix}. \tag{5.15}$$

$$\dot{x} = 91.8 - 0.0318x + 0.0583y - 0.0342z, \tag{5.16}$$

$$\dot{y} = -284 + 0.0352x - 0.0365y + 0.00565z, \tag{5.17}$$

$$\dot{z} = -378 + 0.0188y - 0.0185z. \tag{5.18}$$

Results

The results for this case is good, and the approximated trajectories follows the main trends in the data. The data has some spikes to it, and the approximation does not follow these, but follows the trend of the data more smoothly except for z-data where it does not follows as good as the two others.

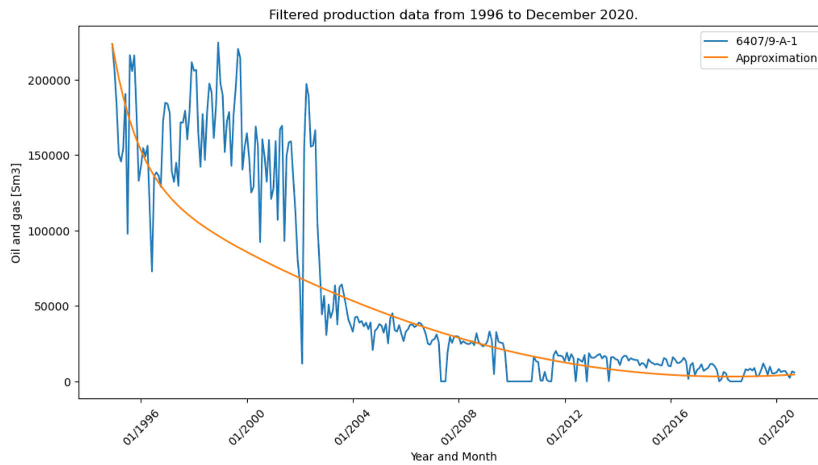


Figure 5.14: The result for x-data

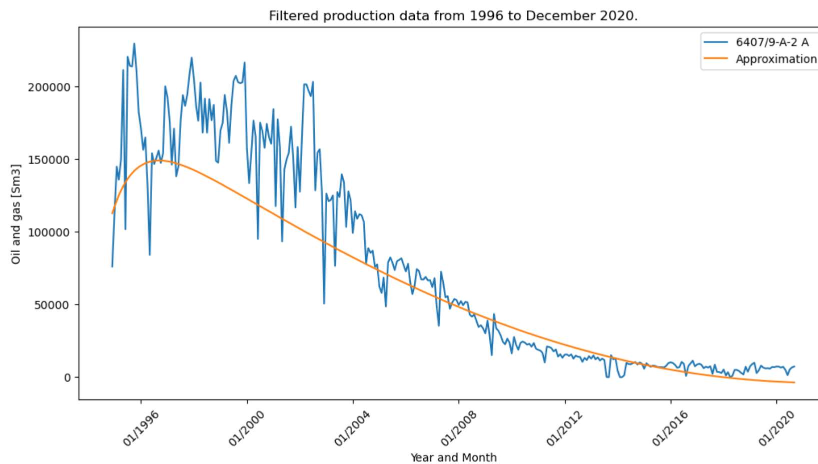


Figure 5.15: The result for y-data

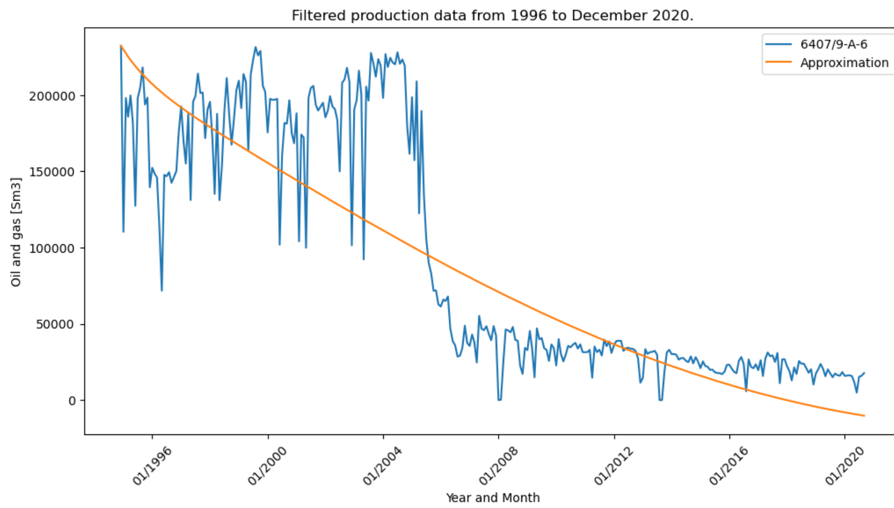


Figure 5.16: The result for z-data

Discussion

Decent results for both x and y data, but not as good for z-data.

The worst result in this test is for the z data where it can look like the solution is a litte to sparse and it does not match the trend of the data very well. For the x and y data if follows the trend nicely, and especially from the middle and out. Maybe the results could be better for z-data is the polynomial order was set higher to 2 for example. This will be tested in test 2 below.

Even though the results was not that good for the z-data, the other two was pretty good with a sparse solution of only four terms in the output equations that describes the trajectories.

5.3.2 Test 2

For the second test everything is similar to test 1 except the polynomial order is set to 2. This is to see if the added candidate terms can make the results for z-data better while keeping the good results for the x and y-approximation.

Polynomial order = 2

USE TRIG = FALSE

USE EXP = FALSE

Threshold = 0.005

Max iterations = 10

median filter size = 4

Results

Then the Ξ -matrix became:

$$\Xi = \begin{bmatrix} \dot{x}: & \dot{y}: & \dot{z}: \\ 9.96e+01 & -2.73e+02 & -3.83e+02 \\ -3.22e-02 & 3.70e-02 & 0.00 \\ 5.87e-02 & -3.83e-02 & 1.90e-02 \\ -3.43e-02 & 5.80e-03 & -1.86e-02 \\ 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 \end{bmatrix} \begin{matrix} \\ 1 \\ x \\ y \\ z \\ x^2. \\ xy \\ xz \\ y^2 \\ yz \\ z^2 \end{matrix} \tag{5.19}$$

$$\dot{x} = 99.6 - 0.0322x + 0.0587y - 0.0343y, \tag{5.20}$$

$$\dot{y} = -273 + 0.0370x - 0.0383y + 0.00580z, \tag{5.21}$$

$$\dot{z} = -383 + 0.0190y - 0.0186z. \tag{5.22}$$

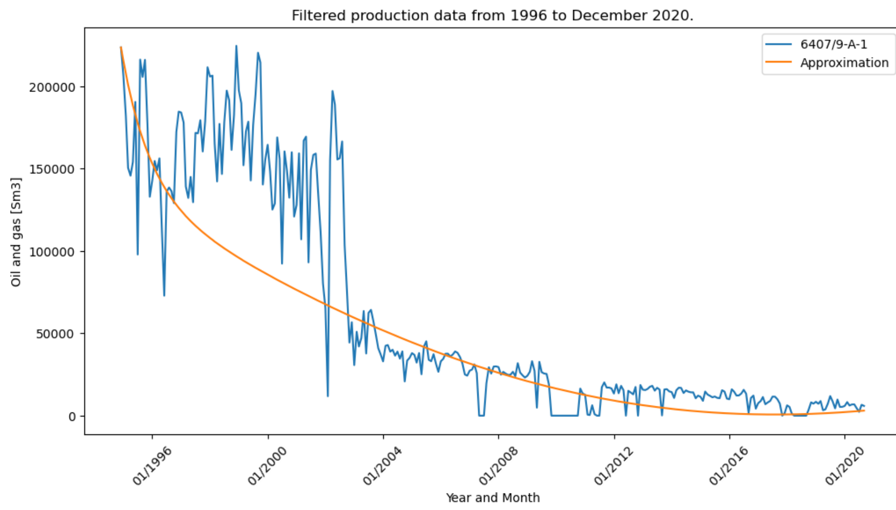


Figure 5.17: The result for x-data

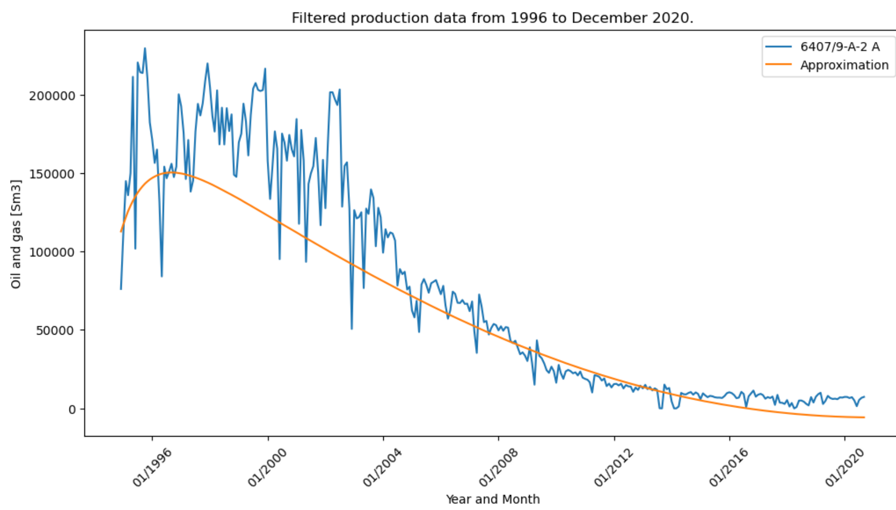


Figure 5.18: The result for y-data

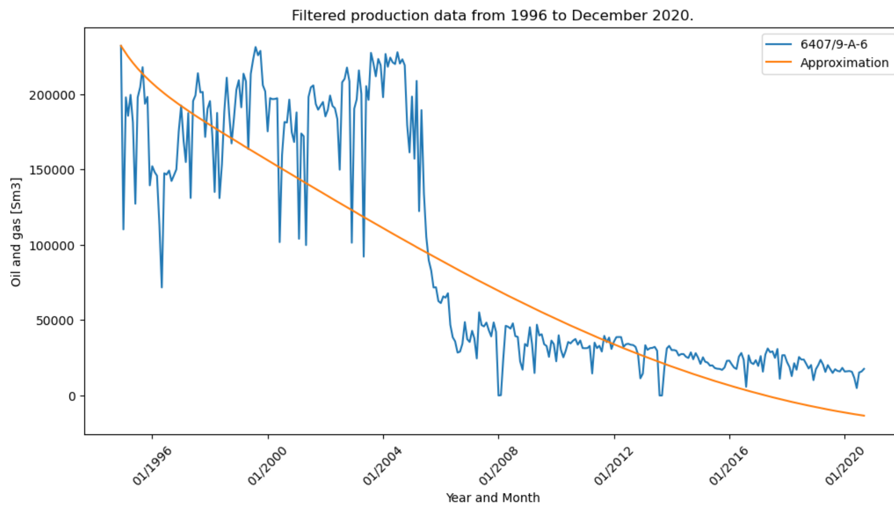


Figure 5.19: The result for z-data

Discussion

These results are almost excactly the same as test 1, and the added candidate terms in the library does not make any difference. There are some difference in the values the SINDy method found, but not a big difference. If the threshold was set lower and included smaller numbers that could make more terms appear. Even though more terms would appear the results would not be sparse anymore and most likely not better.

5.3.3 Test 3

In this test two wells located close to each other and a third well from a further distance was added. This well has a different production timeline. The production has been set to 0 when there is not any data. Due to the need of the data matrix has to be the same size.

For x and y the data are from the same wells as above, but the z-data has been changed to data from another well. It is from well 6407/9-E-1 H, which has been producicong from about 2003. This well is located over 10 kilometers away from the other two wells, see figure 5.20 below.

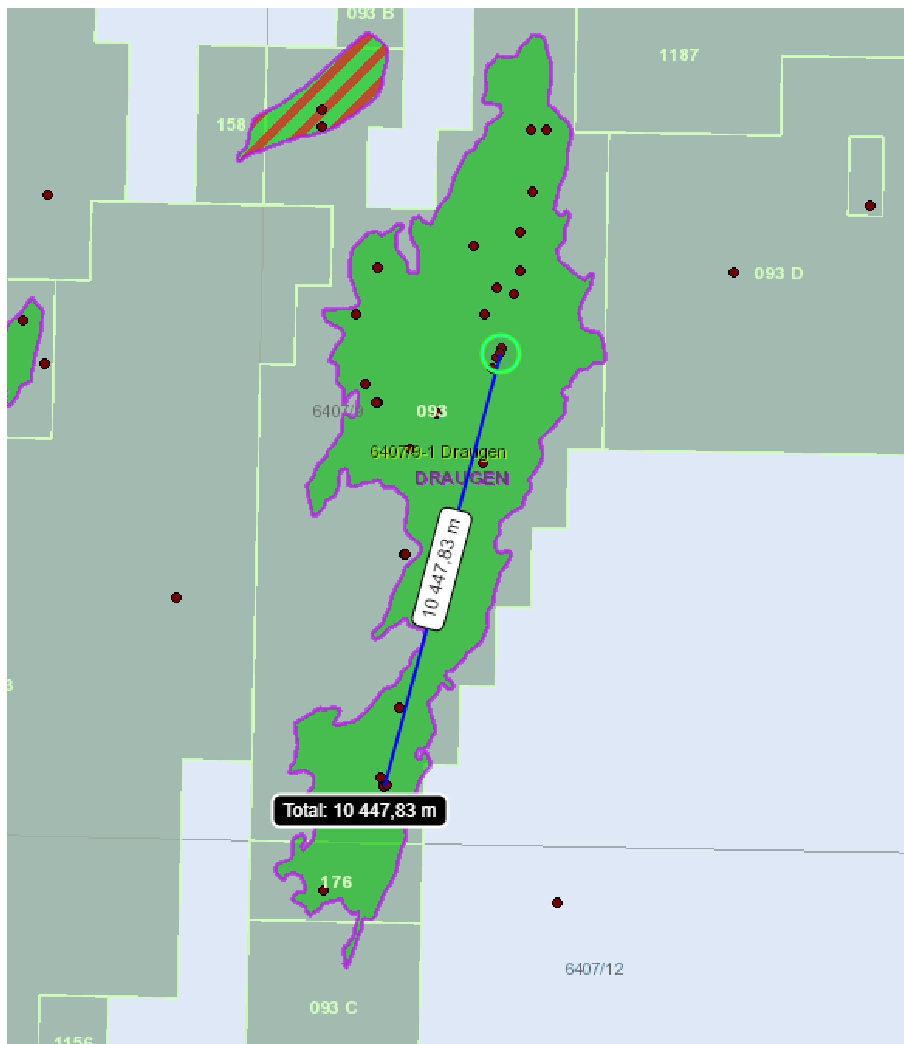


Figure 5.20: Distance between the wells during this test on Draugen. Map taken from Norwegian Petroleum Directorade. `https://factmaps.npd.no/factmaps/3_0/`. (accessed: 22.05.23).

The data and well layot for this test are listed below to make it clear which results are from what data.

x = 6407/9-A-1

y = 6407/9-A-2 A

z = 6407/9-E-1 H

There was some change to the parameters in this test, but not much. The threshold was put down to 0.001 instead of 0.005, and the polynomial order was set back to 1 again. All the parameters was:

Polynomial order = 1

USE TRIG = FALSE

USE EXP = FALSE

Threshold = 0.001

Max iterations = 10

median filter size = 4

Results

Using these data and parameters and the Ξ -matrix became:

$$\Xi = \begin{bmatrix} \dot{x}: & \dot{y}: & \dot{z}: \\ 4.02e+01 & 3.22e+02 & -1.75e+02 \\ -4.85e-02 & 1.99e-02 & 0.00 \\ 3.63e-02 & -1.82e-02 & 2.04e-03 \\ -1.59e-01 & -9.23e-03 & 2.33e-03 \end{bmatrix} \begin{matrix} \\ 1 \\ x \\ y \\ z \end{matrix}. \tag{5.23}$$

$$\dot{x} = 40.2 - 0.0485x + 0.0363y - 0.159z, \tag{5.24}$$

$$\dot{y} = 322 + 0.0199x - 0.0182y - 0.00923z, \tag{5.25}$$

$$\dot{z} = -175 + 0.00204y + 0.00233z. \tag{5.26}$$

The results here are not promising, and it's clear to see that the SINDy method struggles to find a good approximation of the dynamics. The results are unphysically for z-data when the approximation goes below zero. For the x- and y-data the approximation is decent in the beginning, but there is a change from the middle of the trajectory and to the end. Here the SINDy method approximated an increased production from the well, which is clear that did not happen.

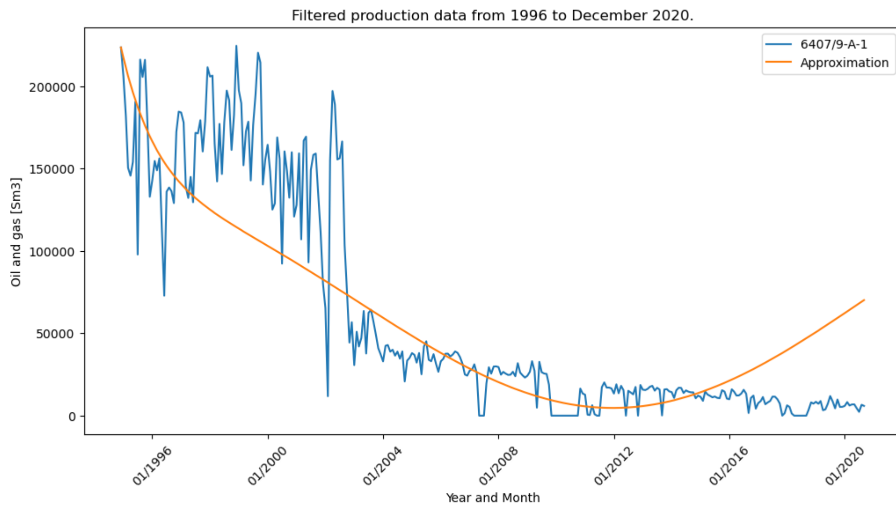


Figure 5.21: The result for x-data

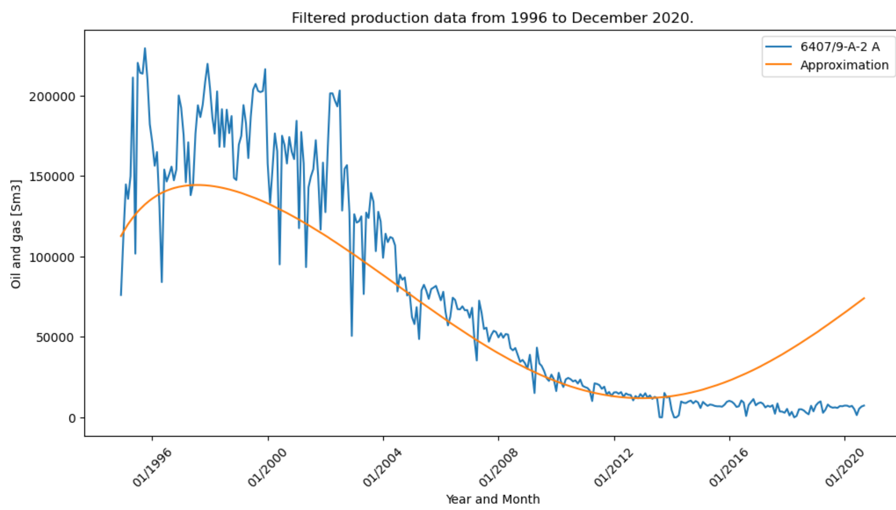


Figure 5.22: The result for y-data

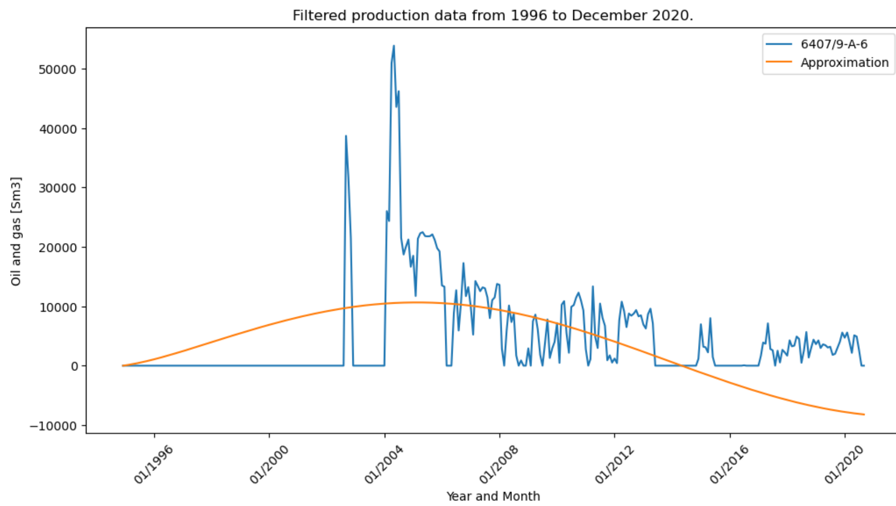


Figure 5.23: The result for z-data

Discussion

One of the reasons of the bad results in this test could be the production timeline, where the x and y data has been producing for a much longer time compared to z-data.

In figure 5.24 below there is a plot of the data used in this test compared to eachother. It is clear that the volume of the production from x and y data are much higher compared to the z-data. That could also make a difference in the approximation.

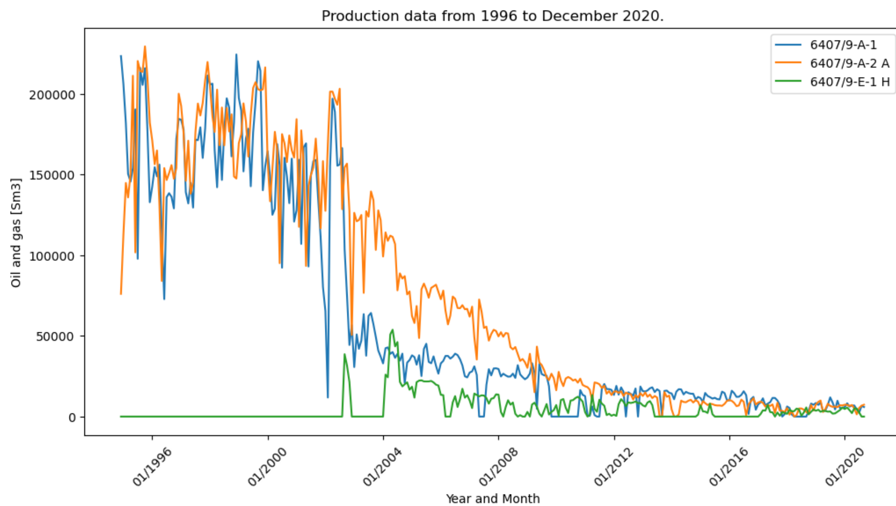


Figure 5.24: The data used in this test.

Also the timeline in these are different, and the data for z-data has been set to zero in the beginning. This could also effect the results, because there are no production for a

long time and then suddenly it starts to produce and then go back to zero again before it produces more regularly.

This test was mainly to see how the distance and different timeline affected the results, and it is clear to see that the results was not as good compared to the other wells from Draugen.

5.3.4 Test 4

During this test the pySINDy package was used with the same parameters as in Test 1 in section 5.3.1. We can use this to see how the results compare with the self implemented SINDy method. The results should be similar.

For the test the polynomial order was set to 1, and the there was no candidate terms with exponential functions, sine or cosine. The threshold was set to 0.005 and max iterations was set to 10. All the parameters are listed below.

Polynomial order = 1
USE TRIG = FALSE
USE EXP = FALSE
Threshold = 0.005
Max iterations = 10
median filter size = 4

$$\dot{x} = 115.4 - 0.045x + 0.076y - 0.041z, \tag{5.27}$$

$$\dot{y} = -255.0 + 0.028x - 0.024y, \tag{5.28}$$

$$\dot{z} = -396.8 - 0.015x + 0.039y - 0.025z. \tag{5.29}$$

Results

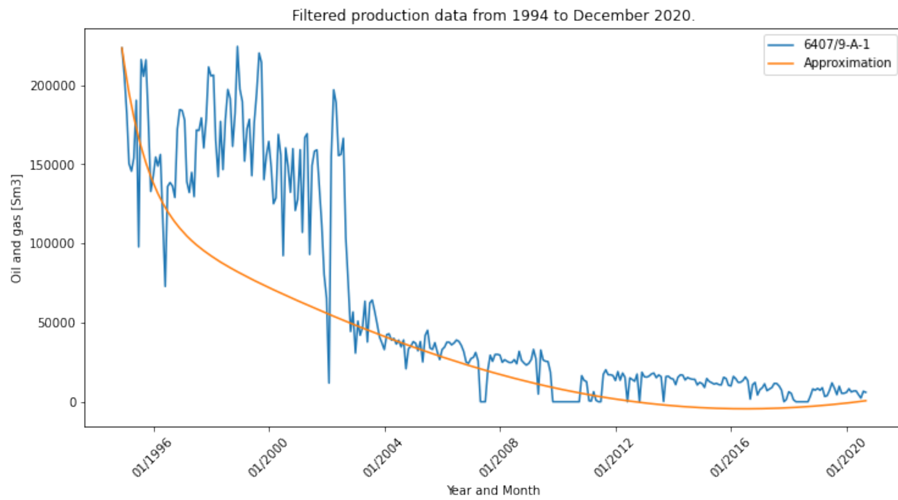


Figure 5.25: The result for x-data

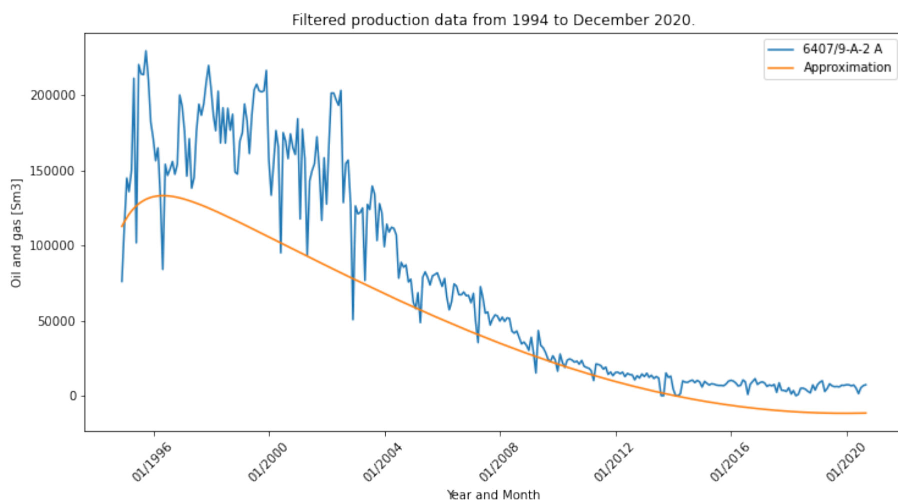


Figure 5.26: The result for y-data

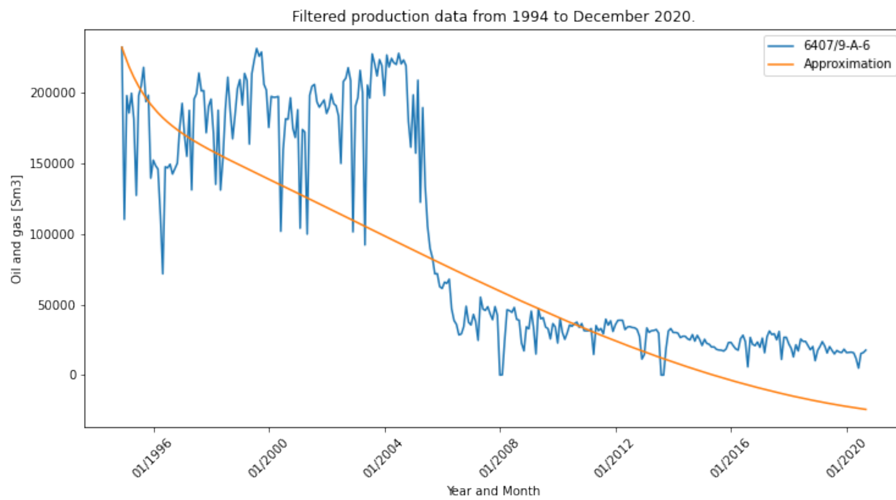


Figure 5.27: The result for z-data

Discussion

The results using the pySINDy package is really similar to the result from the self implemented SINDy method. This shows that the self implemented method works just like the pySINDy package. Even though the pySINDy package has a lot more built in functions, when we use the same parameters and setting the results are similar.

The results that differ most from test 1 in section 5.3.1 is the results for the z-data. Where the trajectory of the approximation goes much further down, and it goes further down in the beginning of the trajectory.

In general the results are similar, but for the results using the PySINDy package the trajectories looks to sit a little lower down in the plot compared to test 1 in section 5.3.1.

Chapter 6

Conclusion

This study have looked into if the SINDy method could be used on well data to extract suitable well models of different complexities. A self implemented SINDy method has been used, and the PySINDy package has also been used to compare the results from the two methods.

The SINDy method can be used to discover suitable models on well data from DISKOS. It is clear that it can follow the data good even though the data is very spiky. When the derivatives are calculated it is a advantage that the data are smooth. To get better results the data are smoothed with a median filter before differentiation and smoothed using kalman smoothing during differentiation.

For this study I used data from three different wells, and the results are good. Added complexity does not always lead to better approximation, and they can be harder to understand compared to the simpler models.

6.1 Future work

For future work there is a lot that can be done and looked at. The study mainly looked at a three dimensional problem using data from three different wells located close to each other. It could be a possibility to add more dimensions to the SINDy method and for example add all the wells that are producing oil and gas in that period of time.

The self implemented SINDy method works fine, and are good for less complex models. If the user want even more complex models it is suggested to use the PySINDy package because it has a lot more built-in libraries and functions that can be utilized. It can be harder to understand what is happening when using the package compared to the self implemented method, but it is designed to fit both new and experienced users of the SINDy method.

The SINDy method can be extended, and it is just our imagination that stops us, and there is a lot of possibilities to be discovered in the future.

Bibliography

[1] Norwegian Petroleum Directorate. *About us. NPD.NO*. URL: https://www.npd.no/en/diskos/About/. (accessed: 22.05.2023).

[2] Norwegian Petroleum Directorate. *Diskos 2.0 is up and running. NPD.NO*. URL: https://www.npd.no/en/diskos/news/diskos-2.0-is-up-and-running/. (accessed: 22.05.2023).

[3] Norwegian Petroleum Directorate. *Diskos. NPD.NO*. URL: https://www.npd.no/en/about-us/organisation/collaboration-projects/diskos/. (accessed: 21.05.2023).

[4] Norwegian Petroleum Directorate. *DRAUGEN. NORSKPETROLEUM.NO*. URL: https://www.norskpetroleum.no/fakta/felt/draugen/. (accessed: 14.03.2023).

[5] Norwegian Petroleum Directorate. *STATFJORD ØST. NORSKPETROLEUM.NO*. URL: https://www.norskpetroleum.no/en/facts/field/statfjord-ost/. (accessed: 14.02.2023).

[6] Norwegian Petroleum Directorate. *Water Injection. NPD.NO*. URL: https://www.npd.no/en/facts/production/improved-oil-recovery-ior/water/. (accessed: 14.02.2023).

[7] Gardner L. Dong E Du H. "An interactive web-based dashboard to track COVID-19 in real time. Lancet Inf Dis. 20(5):533-534." In: (). DOI: 10.1016/S1473-3099(20)30120-1. URL: https://github.com/CSSEGISandData/COVID-19.

[8] Nykamp DQ. *Dynamical system definition. From Math Insight*. URL: https://mathinsight.org/definition/dynamical_system. (accessed: 20.05.2023).

[9] Equinor. *Improving recovery from Statfjord Øst. EQUINOR.COM*. URL: https://www.equinor.com/news/archive/202012-improving-recovery-statfjord-east. (accessed: 12.04.2023).

[10] J. Nathan Kutz Kadierdan Kaheman and Steve L. Brunton. (2020). *SINDy-PI: a robust algorithm for parallel implicit sparse identification of nonlinear dynamics*. URL: https://royalsocietypublishing.org/doi/abs/10.1098/rspa.2020.0279. (accessed: 24.01.2023).

[11] Alan A. Kaptanoglu et al. "PySINDy: A comprehensive Python package for robust sparse system identification". In: *Journal of Open Source Software* 7.69 (2022), p. 3994. DOI: 10.21105/joss.03994. URL: https://doi.org/10.21105/joss.03994.

[12] OKEA. *Draugen er fortsatt drivkraften. OKEA.NO*. URL: https://www.okea.no/stories/draugen-er-fortsatt-drivkraften/. (accessed: 26.04.2023).

[13] OKEA. *Draugen. OKEA.NO*. URL: https://www.okea.no/no/asset/draugen-1/. (accessed: 25.04.2023).

[14] J. N. Kutz S. L. Brunton. *DATA-DRIVEN SCIENCE AND ENGINEERING Machine Learning, Dynamical Systems, and Control, Second Edition*. Cambridge University Press, 2022.

[15] Scipy. *scipy.integrate.solve_ivp*. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.solve_ivp.html.

[16] Brian M. de Silva et al. "PySINDy: A Python package for the sparse identification of nonlinear dynamical systems from data". In: *Journal of Open Source Software* 5.49 (2020), p. 2104. DOI: 10.21105/joss.02104. URL: https://doi.org/10.21105/joss.02104.

[17] Joshua L. Proctor Steve L. Brunton and J. Nathan Kutz. (2016). *Discovering governing equations from data by sparse identification of nonlinear dynamical systems*. URL: https://www.pnas.org/doi/10.1073/pnas.1517384113. (accessed: 22.01.2023).

[18] B. Stollnitz. *Discovering equations from data using SINDy. Bea Stollnitz*. URL: https://bea.stollnitz.com/blog/sindy-lorenz/. (accessed: 19.01.2023).

Appendix A

Appendix A

A.1 Code for Lorenz system

sindy-lorenz-ex

June 14, 2023

1 SINDy utilized on the Lorenz System

[1]:
```
import numpy as np
import logging
from scipy.integrate import solve_ivp
from derivative import dxdt
from typing import Tuple
import matplotlib.pyplot as plt
```

[2]:
```
def lorenz(_: float, u: np.ndarray, sigma: float, rho: float,
           beta: float) -> np.ndarray:

    x = u[0]
    y = u[1]
    z = u[2]
    dx_dt = sigma * (y - x)
    dy_dt = x * (rho - z) - y
    dz_dt = x * y - beta * z

    return np.hstack((dx_dt, dy_dt, dz_dt))

def generate_u(t: np.ndarray) -> np.ndarray:
    u0 = np.array([-8, 8, 27])
    result = solve_ivp(fun=lorenz,
                       t_span=(t[0], t[-1]),
                       y0=u0,
                       t_eval=t,
                       args=(SIGMA, RHO, BETA))
    u = result.y.T
    return u

def calculate_finite_difference_derivatives(u: np.ndarray,
                                            t: np.ndarray) -> np.ndarray:

    logging.info("Using finite difference derivatives.")
    uprime = dxdt(u.T, t, kind="finite_difference", k=1).T
    return uprime
```

[3]:
```
def generate_data() -> Tuple[np.ndarray, np.ndarray]:
    """ Generates data u, and calculates its derivatives uprime.
    """
    t0 = 0.001
    dt = 0.001
    tmax = 100
    n = int(tmax / dt)
    t = np.linspace(start=t0, stop=tmax, num=n)

    # Step 1: Generate data u.
    u = generate_u(t)

    # Step 2: Calculate u' from u.
    uprime = calculate_finite_difference_derivatives(u, t)

    return (u, uprime)
```

[4]:
```
"""Utilities related to graphing."""

def style_axis(axis):
    """Styles a graph's x, y, or z axis."""
    # pylint: disable=protected-access
    axis._axinfo["grid"]["color"] = "#dddddd"
    axis._axinfo["grid"]["linewidth"] = 0.4
    axis._axinfo["tick"]["linewidth"][True] = 0.4
    axis._axinfo["tick"]["linewidth"][False] = 0.4
    axis.set_pane_color((0.98, 0.98, 0.98, 1.0))
    axis.line.set_color("#bbbbbb")
    axis.label.set_color("#333333")
    pass


def style_axis3d(axis3d):
    """Styles a 3D graph."""
    axis3d.set_xlabel("x")
    axis3d.set_ylabel("y")
    axis3d.set_zlabel("z")
    axis3d.tick_params(axis="x", colors="#666666")
    axis3d.tick_params(axis="y", colors="#666666")
    axis3d.tick_params(axis="z", colors="#666666")
    style_axis(axis3d.w_xaxis)
    style_axis(axis3d.w_yaxis)
    style_axis(axis3d.w_zaxis)
    axis3d.set_title(axis3d.get_title(), fontdict={"color": "#333333"})


def graph_results(u: np.ndarray, u_approximation: np.ndarray) -> None:
```

```
    """Graphs two 3D trajectories side-by-side."""
    sample_count = 20000
    orange = "#EF6C00"

    fig = plt.figure(figsize=plt.figaspect(0.5))

    # Graph trajectory from the true model.
    axis3d = fig.add_subplot(1, 2, 1, projection="3d")
    x = u[0:sample_count, 0]
    y = u[0:sample_count, 1]
    z = u[0:sample_count, 2]
    axis3d.plot3D(x, y, z, orange, linewidth=0.4)
    axis3d.set_title("Original trajectory")
    style_axis3d(axis3d)

    # Graph trajectory computed from model discovered by SINDy.
    axis3d = fig.add_subplot(1, 2, 2, projection="3d")
    x = u_approximation[0:sample_count, 0]
    y = u_approximation[0:sample_count, 1]
    z = u_approximation[0:sample_count, 2]
    axis3d.plot3D(x, y, z, orange, linewidth=0.4)
    axis3d.set_title("SINDy approximation")
    style_axis3d(axis3d)

    plt.show()
```

[5]:
```
def calculate_regression(theta: np.ndarray, uprime: np.ndarray,
                         threshold: float, max_iterations: int) -> np.ndarray:
    # Solve theta * xi = uprime in the least-squares sense.
    xi = np.linalg.lstsq(theta, uprime, rcond=None)[0]
    n = xi.shape[1]

    # Add sparsity.
    for _ in range(max_iterations):
        small_indices = np.abs(xi) < threshold
        xi[small_indices] = 0
        for j in range(n):
            big_indices = np.logical_not(small_indices[:, j])
            xi[big_indices, j] = np.linalg.lstsq(theta[:, big_indices],
                                                 uprime[:, j],
                                                 rcond=None)[0]

    return xi
```

[6]:
```
def create_library(u: np.ndarray, polynomial_order: int) -> np.ndarray:
    """Creates a matrix containing a library of candidate functions.
    For example, if our u depends on x, y, and z, and we specify
```

```
    polynomial_order=2, our terms would be:
    1, x, y, z, x^2, xy, xz, y^2, yz, z^2.
    """
    (m, n) = u.shape
    theta = np.ones((m, 1))

    # Polynomials of order 1.
    theta = np.hstack((theta, u))

    # Polynomials of order 2.
    if polynomial_order >= 2:
        for i in range(n):
            for j in range(i, n):
                theta = np.hstack((theta, u[:, i:i + 1] * u[:, j:j + 1]))

    # Polynomials of order 3.
    if polynomial_order >= 3:
        for i in range(n):
            for j in range(i, n):
                for k in range(j, n):
                    theta = np.hstack(
                        (theta, u[:, i:i + 1] * u[:, j:j + 1] * u[:, k:k + 1]))

    # Polynomials of order 4.
    if polynomial_order >= 4:
        for i in range(n):
            for j in range(i, n):
                for k in range(j, n):
                    for l in range(k, n):
                        theta = np.hstack(
                            (theta, u[:, i:i + 1] * u[:, j:j + 1] *
                             u[:, k:k + 1] * u[:, l:l + 1]))

    # Polynomials of order 5.
    if polynomial_order >= 5:
        for i in range(n):
            for j in range(i, n):
                for k in range(j, n):
                    for l in range(k, n):
                        for m in range(l, n):
                            theta = np.hstack(
                                (theta, u[:, i:i + 1] * u[:, j:j + 1] *
                                 u[:, k:k + 1] * u[:, l:l + 1] * u[:, m:m + 1]))

    return theta
```

```python
[7]: def lorenz_approximation(_: float, u: np.ndarray, xi: np.ndarray,
                             polynomial_order: int) -> np.ndarray:
         theta = create_library(u.reshape((1, 3)), polynomial_order)
         return theta @ xi

     def compute_trajectory(u0: np.ndarray, xi: np.ndarray, polynomial_order: int)␣
      ↪-> np.ndarray:
         t0 = 0.001
         dt = 0.001
         tmax = 100
         n = int(tmax / dt + 1)

         t = np.linspace(start=t0, stop=tmax, num=n)
         result = solve_ivp(fun=lorenz_approximation,
                            t_span=(t0, tmax),
                            y0=u0,
                            t_eval=t,
                            args=(xi, polynomial_order))
         u = result.y.T

         return u
```

```python
[8]: SIGMA = 10
     RHO = 28
     BETA = 8 / 3

     POLYNOMIAL_ORDER = 2

     THRESHOLD = 0.025
     MAX_ITERATIONS = 10
```

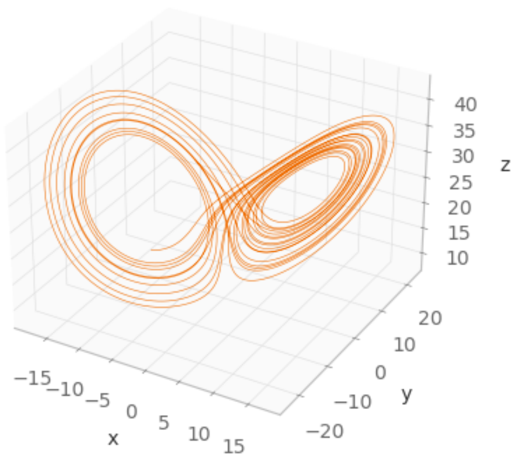
```python
[9]: (u, uprime) = generate_data()
     theta = create_library(u, POLYNOMIAL_ORDER)
     xi = calculate_regression(theta, uprime, THRESHOLD, MAX_ITERATIONS)
     xi
```

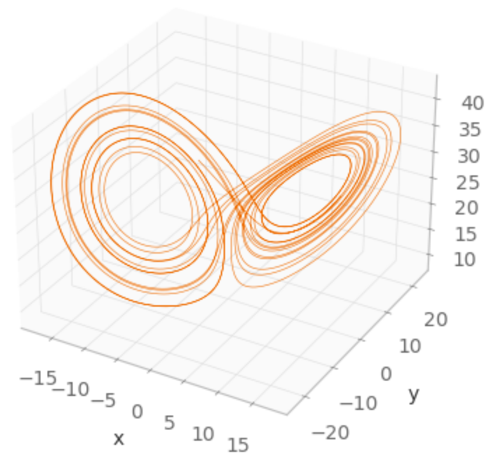
```
[9]: array([[  0.        ,   0.        ,   0.        ],
            [-10.00449518,  27.80496577,   0.        ],
            [ 10.00429024,  -0.95767769,   0.        ],
            [  0.        ,   0.        ,  -2.66700595],
            [  0.        ,   0.        ,   0.        ],
            [  0.        ,   0.        ,   0.99924596],
            [  0.        ,  -0.99347666,   0.        ],
            [  0.        ,   0.        ,   0.        ],
            [  0.        ,   0.        ,   0.        ],
            [  0.        ,   0.        ,   0.        ]])
```

[10]:
```
u0 = np.array([-8, 8, 27])
u_approximation = compute_trajectory(u0, xi, POLYNOMIAL_ORDER)
```

[11]:
```
graph_results(u, u_approximation)
```

```
C:\Users\sande\AppData\Local\Temp\ipykernel_2068\129517075.py:24:
MatplotlibDeprecationWarning: The w_xaxis attribute was deprecated in Matplotlib
3.1 and will be removed in 3.8. Use xaxis instead.
  style_axis(axis3d.w_xaxis)
C:\Users\sande\AppData\Local\Temp\ipykernel_2068\129517075.py:25:
MatplotlibDeprecationWarning: The w_yaxis attribute was deprecated in Matplotlib
3.1 and will be removed in 3.8. Use yaxis instead.
  style_axis(axis3d.w_yaxis)
C:\Users\sande\AppData\Local\Temp\ipykernel_2068\129517075.py:26:
MatplotlibDeprecationWarning: The w_zaxis attribute was deprecated in Matplotlib
3.1 and will be removed in 3.8. Use zaxis instead.
  style_axis(axis3d.w_zaxis)
```





[ ]:

A.2 Code for Covid-19

covid-19

June 14, 2023

1 Covid 19 example

[1]:
```
import requests
import io
import numpy as np
import matplotlib.dates as mdates
import pandas as pd
import matplotlib.pyplot as plt
```

[2]:
```
url = 'https://raw.githubusercontent.com/SanderSondeland/Master/main/
 ↪corona_data_new.csv?token=GHSAT0AAAAAACBRT5KRZAWZGFMMNZ3Y5IBAZEIELWA'
download = requests.get(url).content
data = pd.read_csv(io.StringIO(download.decode('utf-8')), sep='\t')
data
```

[2]:
```
          LOCATION                  TIME  ELAPSED_TIME_SINCE_OUTBREAK  \
0      Afghanistan  2020-02-24 23:59:00                            0
1      Afghanistan  2020-02-25 23:59:00                            1
2      Afghanistan  2020-02-26 23:59:00                            2
3      Afghanistan  2020-02-27 23:59:00                            3
4      Afghanistan  2020-02-28 23:59:00                            4
…              …                   …                          …
44516        Hubei  2020-10-21 23:59:00                          273
44517        Hubei  2020-10-22 23:59:00                          274
44518        Hubei  2020-10-23 23:59:00                          275
44519        Hubei  2020-10-24 23:59:00                          276
44520        Hubei  2020-10-25 23:59:00                          277

       CONFIRMED  DEATHS  RECOVERED
0              1       0          0
1              1       0          0
2              1       0          0
3              1       0          0
4              1       0          0
…            …     …         …
44516      68139    4512      63627
44517      68139    4512      63627
```

```
44518       68139    4512      63627
44519       68139    4512      63627
44520       68139    4512      63627

[44521 rows x 6 columns]
```

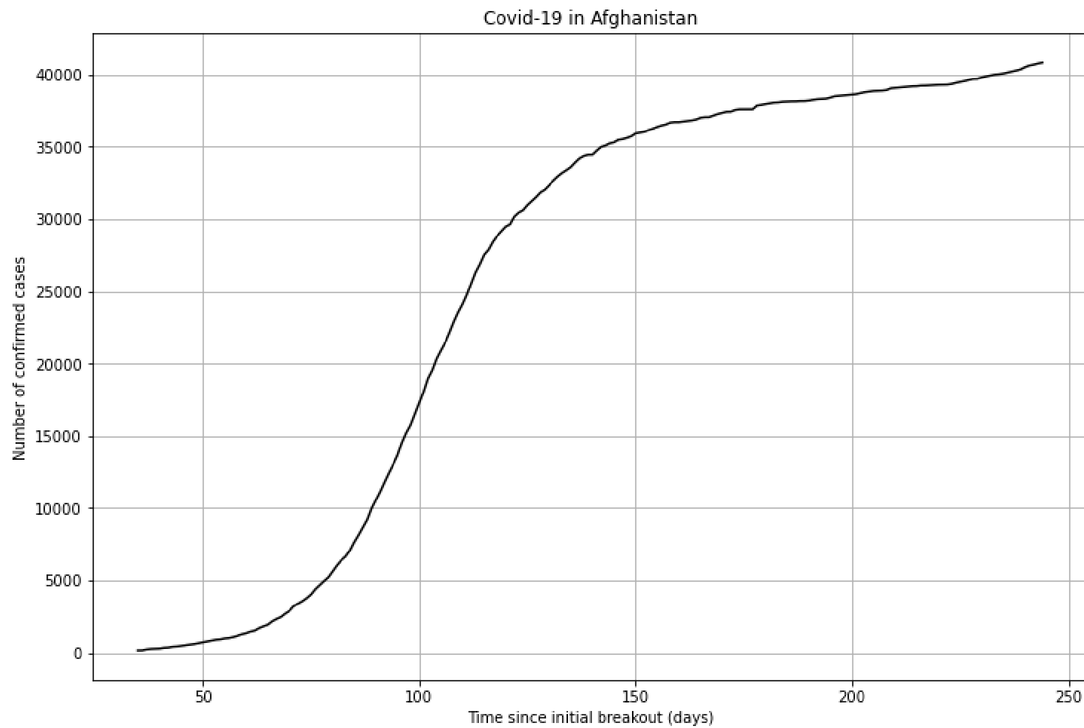
[3]:
```
data_A = data[data['LOCATION'] == 'Afghanistan']
data_A
```

[3]:
```
        LOCATION                 TIME  ELAPSED_TIME_SINCE_OUTBREAK  CONFIRMED  \
0    Afghanistan  2020-02-24 23:59:00                            0          1
1    Afghanistan  2020-02-25 23:59:00                            1          1
2    Afghanistan  2020-02-26 23:59:00                            2          1
3    Afghanistan  2020-02-27 23:59:00                            3          1
4    Afghanistan  2020-02-28 23:59:00                            4          1
..           …                  …                          …        …
240  Afghanistan  2020-10-21 23:59:00                          240      40510
241  Afghanistan  2020-10-22 23:59:00                          241      40626
242  Afghanistan  2020-10-23 23:59:00                          242      40687
243  Afghanistan  2020-10-24 23:59:00                          243      40768
244  Afghanistan  2020-10-25 23:59:00                          244      40833

     DEATHS  RECOVERED
0         0          0
1         0          0
2         0          0
3         0          0
4         0          0
..      …          …
240    1501      33824
241    1505      33831
242    1507      34010
243    1511      34023
244    1514      34129

[245 rows x 6 columns]
```

[4]:
```
time_A = data_A['ELAPSED_TIME_SINCE_OUTBREAK'].to_numpy()
confirmed_A = data_A['CONFIRMED'].to_numpy()
confirmed_A = confirmed_A[35:]
time_A = time_A[35:]
```

[8]:
```
fig, ax = plt.subplots(figsize=(12, 8))
ax.grid()
plt.title('Covid-19 in Afghanistan')
plt.xlabel('Time since initial breakout (days)')
plt.ylabel('Number of confirmed cases')
```

```
ax.plot(time_A, confirmed_A, color='black')

plt.show()
```



2 SINDy method

[7]:
```
from scipy.integrate import solve_ivp
from derivative import dxdt
```

[8]:
```
def create_library(u: np.ndarray, polynomial_order: int,
                   use_trig: bool) -> np.ndarray:
    """Creates a matrix containing a library of candidate functions.
    For example, if our u depends on x, and we specify
    polynomial_order=2 and use_trig=false, our terms would be:
    1, x, x^2.
    """

    u = u.reshape((-1, 1))

    (m, n) = u.shape
    theta = np.ones((m, 1))
```

```python
    # Polynomials of order 1.
    theta = np.hstack((theta, u))

    # Polynomials of order 2.
    if polynomial_order >= 2:
        for i in range(n):
            for j in range(i, n):
                theta = np.hstack((theta, u[:, i:i + 1] * u[:, j:j + 1]))

    # Polynomials of order 3.
    if polynomial_order >= 3:
        for i in range(n):
            for j in range(i, n):
                for k in range(j, n):
                    theta = np.hstack(
                        (theta, u[:, i:i + 1] * u[:, j:j + 1] * u[:, k:k + 1]))

    # Polynomials of order 4.
    if polynomial_order >= 4:
        for i in range(n):
            for j in range(i, n):
                for k in range(j, n):
                    for l in range(k, n):
                        theta = np.hstack(
                            (theta, u[:, i:i + 1] * u[:, j:j + 1] *
                             u[:, k:k + 1] * u[:, l:l + 1]))

    # Polynomials of order 5.
    if polynomial_order >= 5:
        for i in range(n):
            for j in range(i, n):
                for k in range(j, n):
                    for l in range(k, n):
                        for m in range(l, n):
                            theta = np.hstack(
                                (theta, u[:, i:i + 1] * u[:, j:j + 1] *
                                 u[:, k:k + 1] * u[:, l:l + 1] * u[:, m:m + 1]))

    if use_trig:
        for i in range(1, 11):
            theta = np.hstack((theta, np.sin(i * u), np.cos(i * u)))

    return theta
```

[10]:
```python
def calculate_regression(theta: np.ndarray, uprime: np.ndarray, threshold:␣
↪float, max_iterations: int) -> np.ndarray:
    # Solve theta * xi = uprime in the least-squares sense.
```

```
    xi = np.linalg.lstsq(theta, uprime, rcond=None)[0]
    xi_before = xi
    n = len(xi)
    #print('xi before sparisty: ', xi)

    # Add sparsity.
    for _ in range(max_iterations):
        small_indices = np.abs(xi) < threshold
        xi[small_indices] = 0
        for j in range(n):
            big_indices = np.logical_not(small_indices)
            xi[big_indices] = np.linalg.lstsq(theta[:, big_indices],
                                              uprime,
                                              rcond=None)[0]

    return xi_before, xi
```

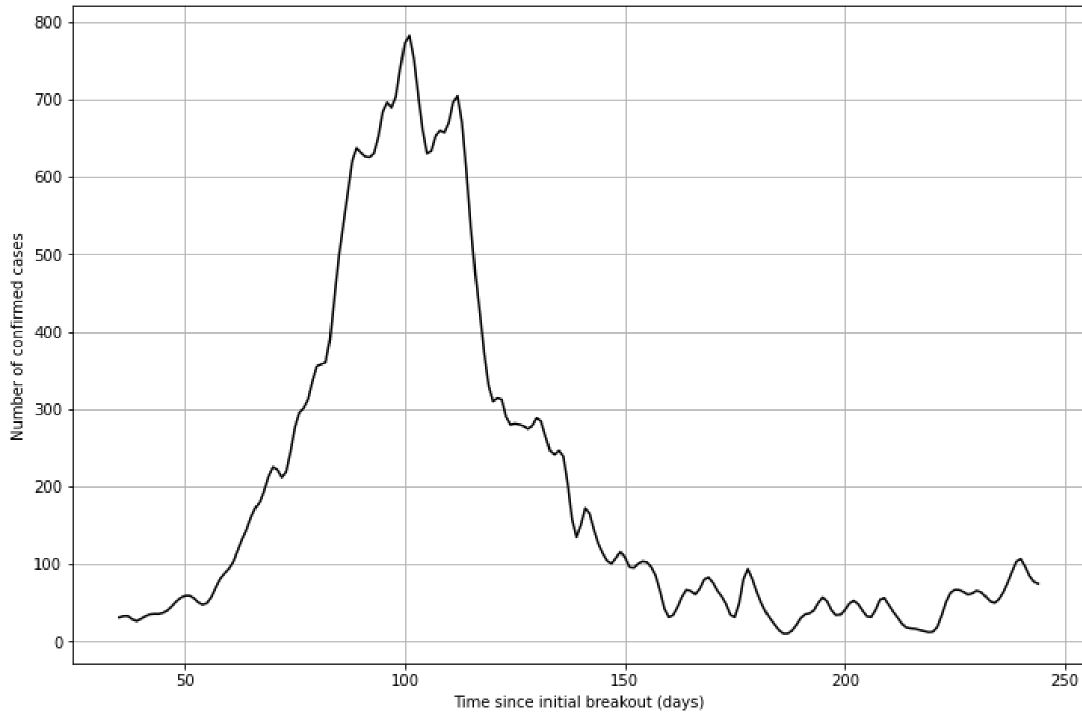
[11]:
```
def approximation(_: float, u: np.ndarray, xi: np.ndarray) -> np.ndarray:
    theta = create_library(u.reshape((1, 3)), POLYNOMIAL_ORDER, USE_TRIG)
    return theta @ xi

def compute_trajectory(u0: np.ndarray, xi: np.ndarray) -> np.ndarray:
    if u0.size == 1:
        u0 = np.repeat(u0, 3)
    else:
        u0 = u0.reshape((3,))
    t0 = 0
    dt = 1
    tmax = 210
    n = int(tmax / dt + 1)

    t = np.linspace(start=t0, stop=tmax, num=n)
    result = solve_ivp(fun=approximation,
                       t_span=(t0, tmax),
                       y0=u0,
                       t_eval=t,
                       args=(xi, ))
    u = result.y.T

    return u
```

[12]:
```
data_A_der = dxdt(confirmed_A, time_A, kind="kalman", alpha=2)
```

[13]:
```
fig, ax = plt.subplots(figsize=(12, 8))
ax.grid()
ax.set_xlabel('Time since initial breakout (days)')
ax.set_ylabel('Number of confirmed cases')
```

```
ax.plot(time_A, data_A_der, color='black')
```

[13]: [<matplotlib.lines.Line2D at 0x24e616251c0>]



[14]:
```
POLYNOMIAL_ORDER = 3
USE_TRIG = False
USE_EXP = False
T_ORDER = 1

theta = create_library(confirmed_A, POLYNOMIAL_ORDER, USE_TRIG)
```

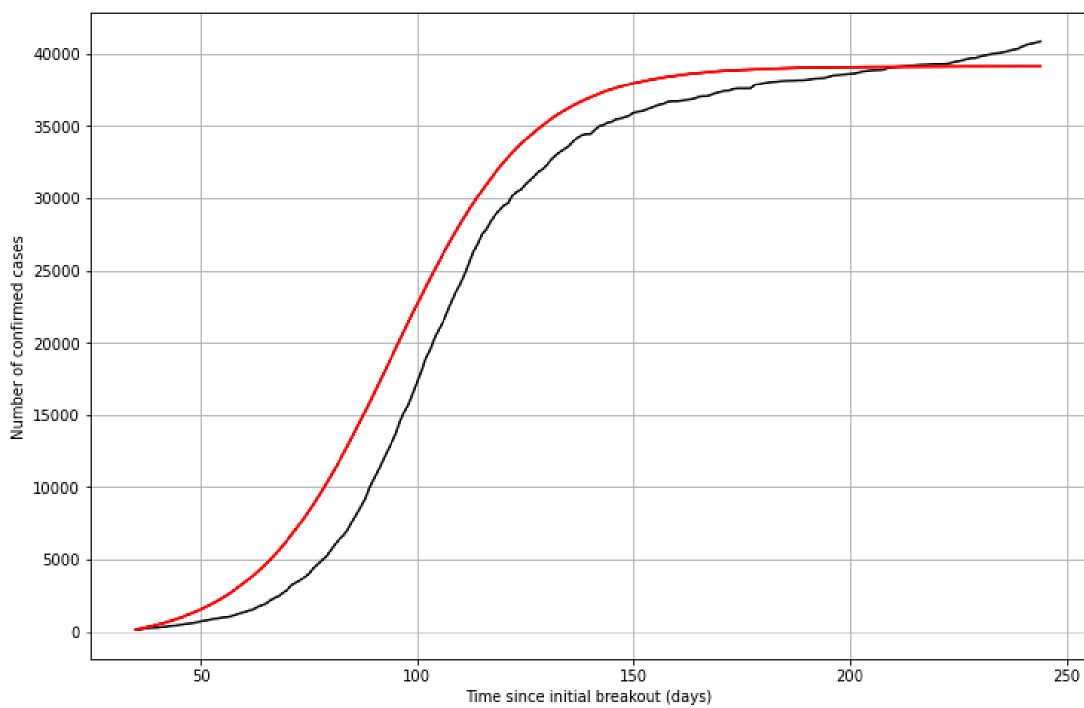
[25]:
```
THRESHOLD = 0.000001
MAX_ITERATIONS = 10

xi_before, xi = calculate_regression(theta, data_A_der, THRESHOLD,
    MAX_ITERATIONS)

mat_0 = confirmed_A[0]
u_approximation = compute_trajectory(mat_0, xi)
xi
```

(4,)

[25]: array([4.74599839e+01, 6.02929056e-02, -1.57168749e-06, 0.00000000e+00])

[28]:
```
fig, ax = plt.subplots(figsize=(12, 8))
ax.grid()
ax.set_xlabel('Time since initial breakout (days)')
ax.set_ylabel('Number of confirmed cases')
ax.plot(time_A, confirmed_A, color='black')
ax.plot(time_A, u_approximation[:-1], color='red')
```

[28]:
```
[<matplotlib.lines.Line2D at 0x24e66ef89d0>,
 <matplotlib.lines.Line2D at 0x24e66ef8a00>,
 <matplotlib.lines.Line2D at 0x24e66ef8b20>]
```



[ ]:

A.3 Code for SINDy on Statfjord Øst

threewells

June 14, 2023

1 Testing on three wells.

Testing on three wells to see if it is possible to find some interference between the wells using SIDNy.

[1]:
```
# Import modules

import numpy as np
import pandas as pd
import requests
import io
import logging
from scipy.integrate import solve_ivp
from derivative import dxdt
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
from scipy.ndimage import median_filter
```

[2]:
```
url = 'https://raw.githubusercontent.com/SanderSondeland/Master/main/
↪20230118_WellBore_monthlyFacility.csv?
↪token=GHSAT0AAAAAACD4GTTLIUBPTGRR75LLYNE4ZEJZVTA'
download = requests.get(url).content
data = pd.read_csv(io.StringIO(download.decode('utf-8')))
data
```

[2]:
```
              name  npdId          field  year  month  operationTime  \
0     33/9-C-16 A   7878  STATFJORD ØST  2016     11          25.38
1     33/9-C-16 A   7878  STATFJORD ØST  2016     12            NaN
2     33/9-C-16 A   7878  STATFJORD ØST  2017      1          19.98
3     33/9-C-16 A   7878  STATFJORD ØST  2017      2          13.06
4     33/9-C-16 A   7878  STATFJORD ØST  2017      3            NaN
…             …     …            …     …      …              …
2100  33/9-M-4 AH   2813  STATFJORD ØST  2020      5           5.23
2101  33/9-M-4 AH   2813  STATFJORD ØST  2020      6           0.00
2102  33/9-M-4 AH   2813  STATFJORD ØST  2020      7           0.00
2103  33/9-M-4 AH   2813  STATFJORD ØST  2020      8           0.00
2104  33/9-M-4 AH   2813  STATFJORD ØST  2020      9            NaN

     operationTimeUom wellStatus    oil oilUom      gas gasUom  condensate  \
```

```
0                    d  producing   4603     Sm3   652469     Sm3          NaN
1                    d        NaN  14006     Sm3  1987611     Sm3          NaN
2                    d  producing   6172     Sm3   876347     Sm3          NaN
3                    d  producing   1546     Sm3   219285     Sm3          NaN
4                    d        NaN   2839     Sm3   402431     Sm3          NaN
…                 …        …    …      …       …     …          …
2100                 d  producing   1776     Sm3   483947     Sm3          NaN
2101                 d     closed      0     Sm3        0     Sm3          NaN
2102                 d     closed      0     Sm3        0     Sm3          NaN
2103                 d     closed      0     Sm3        0     Sm3          NaN
2104                 d        NaN      0     Sm3        0     Sm3          NaN

     condensateUom  water waterUom
0              Sm3   5794      Sm3
1              Sm3  28841      Sm3
2              Sm3  14953      Sm3
3              Sm3   9226      Sm3
4              Sm3  13529      Sm3
…              …    …       …
2100           Sm3   6399      Sm3
2101           Sm3      0      Sm3
2102           Sm3      0      Sm3
2103           Sm3      0      Sm3
2104           Sm3      0      Sm3

[2105 rows x 16 columns]
```

[3]:
```
data.name.unique()
```

[3]:
```
array(['33/9-C-16 A', '33/9-C-33 A', '33/9-L-1 H', '33/9-L-2 H',
       '33/9-L-3 H', '33/9-L-4 H', '33/9-M-1 AH', '33/9-M-1 H',
       '33/9-M-2 AH', '33/9-M-2 BH', '33/9-M-2 H', '33/9-M-3 H',
       '33/9-M-4 AH'], dtype=object)
```

[4]:
```
data1 = data.drop(columns=['npdId', 'field', 'operationTime',␣
 ↪'operationTimeUom','wellStatus','oilUom', 'gasUom', 'condensate',
'condensateUom', 'water', 'waterUom'])
data1
```

[4]:
```
             name  year  month    oil      gas
0     33/9-C-16 A  2016     11   4603   652469
1     33/9-C-16 A  2016     12  14006  1987611
2     33/9-C-16 A  2017      1   6172   876347
3     33/9-C-16 A  2017      2   1546   219285
4     33/9-C-16 A  2017      3   2839   402431
…           …   …     …    …      …
2100  33/9-M-4 AH  2020      5   1776   483947
```

```
2101  33/9-M-4 AH  2020      6      0        0
2102  33/9-M-4 AH  2020      7      0        0
2103  33/9-M-4 AH  2020      8      0        0
2104  33/9-M-4 AH  2020      9      0        0

[2105 rows x 5 columns]
```

[5]:

```
x_data = data1[data1['name'] == '33/9-L-3 H']
x_data.loc[:, 'period'] = x_data['year'].astype(str) + '-' + x_data['month'].
 ↪astype(str)
x_data.loc[:, 'tot_prod'] = (x_data['oil'] + x_data['gas']/1000).round(1)
x_data.groupby('period').sum()
x_data
```

```
C:\Users\sande\AppData\Local\Temp\ipykernel_2448\4116822974.py:2:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  x_data.loc[:, 'period'] = x_data['year'].astype(str) + '-' +
x_data['month'].astype(str)
C:\Users\sande\AppData\Local\Temp\ipykernel_2448\4116822974.py:3:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  x_data.loc[:, 'tot_prod'] = (x_data['oil'] + x_data['gas']/1000).round(1)
C:\Users\sande\AppData\Local\Temp\ipykernel_2448\4116822974.py:4: FutureWarning:
The default value of numeric_only in DataFrameGroupBy.sum is deprecated. In a
future version, numeric_only will default to False. Either specify numeric_only
or select only columns which should be valid for the function.
  x_data.groupby('period').sum()
```

[5]:

```
            name  year  month     oil       gas   period  tot_prod
677   33/9-L-3 H  1994     10   51473   7547000  1994-10   59020.0
678   33/9-L-3 H  1994     11   90972  13310000  1994-11  104282.0
679   33/9-L-3 H  1994     12  110652  16155000  1994-12  126807.0
680   33/9-L-3 H  1995      1   99745  14158000   1995-1  113903.0
681   33/9-L-3 H  1995      2   86007  12235000   1995-2   98242.0
..           …     …      …       …         …        …
950   33/9-L-3 H  2019      7       0         0   2019-7       0.0
951   33/9-L-3 H  2019      8       0         0   2019-8       0.0
952   33/9-L-3 H  2019      9       0         0   2019-9       0.0
```

```
953  33/9-L-3 H  2019     10        0          0  2019-10        0.0
954  33/9-L-3 H  2019     11        0          0  2019-11        0.0

[278 rows x 7 columns]
```

[6]:
```
y_data = data1[data1['name'] == '33/9-L-2 H']
y_data.loc[:, 'period'] = y_data['year'].astype(str) + '-' + y_data['month'].
 ↪astype(str)
y_data.loc[:, 'tot_prod'] = (y_data['oil'] + y_data['gas']/1000).round(1)
y_data.groupby('period').sum()
y_data
```

```
C:\Users\sande\AppData\Local\Temp\ipykernel_2448\2731263958.py:2:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  y_data.loc[:, 'period'] = y_data['year'].astype(str) + '-' +
y_data['month'].astype(str)
C:\Users\sande\AppData\Local\Temp\ipykernel_2448\2731263958.py:3:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  y_data.loc[:, 'tot_prod'] = (y_data['oil'] + y_data['gas']/1000).round(1)
C:\Users\sande\AppData\Local\Temp\ipykernel_2448\2731263958.py:4: FutureWarning:
The default value of numeric_only in DataFrameGroupBy.sum is deprecated. In a
future version, numeric_only will default to False. Either specify numeric_only
or select only columns which should be valid for the function.
  y_data.groupby('period').sum()
```

[6]:
```
           name  year  month    oil       gas   period  tot_prod
411  33/9-L-2 H  1996      8  18896   2723000   1996-8   21619.0
412  33/9-L-2 H  1996      9  87941  12497000   1996-9  100438.0
413  33/9-L-2 H  1996     10  83026  11800000  1996-10   94826.0
414  33/9-L-2 H  1996     11  79860  11361000  1996-11   91221.0
415  33/9-L-2 H  1996     12  89757  12765000  1996-12  102522.0
..          ...   ...    ...    ...       ...      ...       ...
672  33/9-L-2 H  2020      5      0         0   2020-5       0.0
673  33/9-L-2 H  2020      6      0         0   2020-6       0.0
674  33/9-L-2 H  2020      7      0         0   2020-7       0.0
675  33/9-L-2 H  2020      8      0         0   2020-8       0.0
676  33/9-L-2 H  2020      9      0         0   2020-9       0.0
```

```
[266 rows x 7 columns]
```

[7]:
```
z_data = data1[data1['name'] == '33/9-L-1 H']
z_data.loc[:, 'period'] = z_data['year'].astype(str) + '-' + z_data['month'].
 ↪astype(str)
z_data.loc[:, 'tot_prod'] = (z_data['oil'] + z_data['gas']/1000).round(1)
z_data.groupby('period').sum()
z_data
```

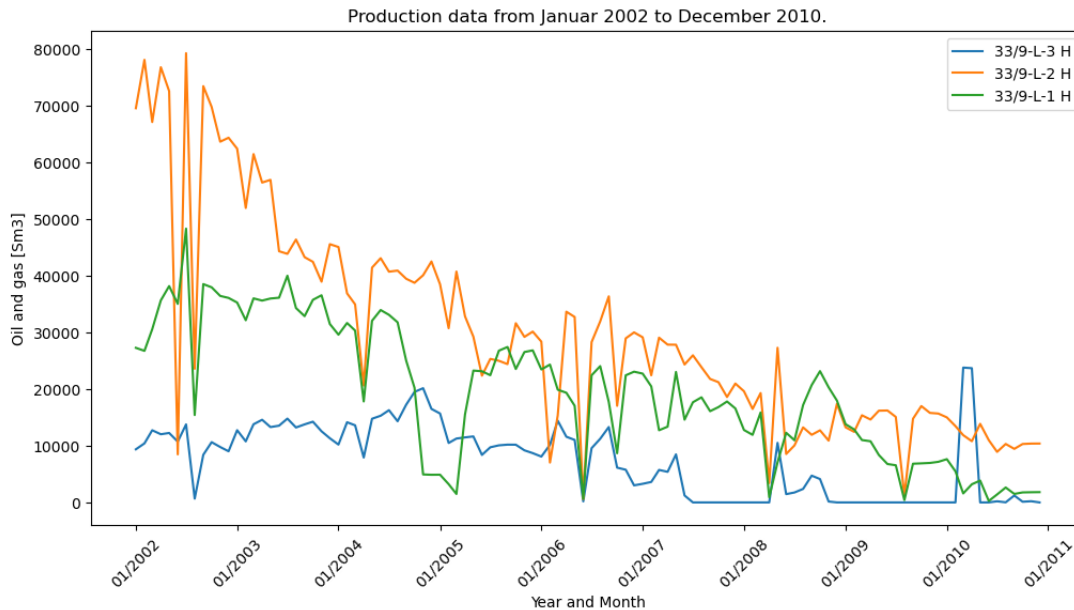
```
C:\Users\sande\AppData\Local\Temp\ipykernel_2448\912022984.py:2:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  z_data.loc[:, 'period'] = z_data['year'].astype(str) + '-' +
z_data['month'].astype(str)
C:\Users\sande\AppData\Local\Temp\ipykernel_2448\912022984.py:3:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  z_data.loc[:, 'tot_prod'] = (z_data['oil'] + z_data['gas']/1000).round(1)
C:\Users\sande\AppData\Local\Temp\ipykernel_2448\912022984.py:4: FutureWarning:
The default value of numeric_only in DataFrameGroupBy.sum is deprecated. In a
future version, numeric_only will default to False. Either specify numeric_only
or select only columns which should be valid for the function.
  z_data.groupby('period').sum()
```

[7]:
```
           name  year  month     oil       gas   period  tot_prod
181  33/9-L-1 H  1999      8   30380   4313000   1999-8   34693.0
182  33/9-L-1 H  1999      9  100684  14297000   1999-9  114981.0
183  33/9-L-1 H  1999     10   99318  14103000  1999-10  113421.0
184  33/9-L-1 H  1999     11   99453  14122000  1999-11  113575.0
185  33/9-L-1 H  1999     12   87483  12388000  1999-12   99871.0
..          …   …      …       …         …        …
406  33/9-L-1 H  2020      5       0         0   2020-5       0.0
407  33/9-L-1 H  2020      6       0         0   2020-6       0.0
408  33/9-L-1 H  2020      7       0         0   2020-7       0.0
409  33/9-L-1 H  2020      8       0         0   2020-8       0.0
410  33/9-L-1 H  2020      9       0         0   2020-9       0.0

[230 rows x 7 columns]
```

[8]:
```
import datetime
dates = [datetime.datetime(year=int(year), month=int(month), day=1) for year in␣
↪range(1999, 2000) for month in range(1, 13)]
dates = dates[7:]
dates2 = [datetime.datetime(year=int(year), month=int(month), day=1) for year␣
↪in range(2002, 2011) for month in range(1,13)]
dates2 = dates2[:]
```

[9]:
```
test_x = x_data[63:-107]
tot_prod_x = test_x['tot_prod']

test_y = y_data[41:-117]
tot_prod_y = test_y['tot_prod']


test_z = z_data[5:-117]
tot_prod_z = test_z['tot_prod']
```

[10]:
```
fig, ax = plt.subplots(figsize=(12, 6))
ax.plot(dates2, tot_prod_x, label='33/9-L-3 H')
ax.plot(dates2, tot_prod_y, label='33/9-L-2 H')
ax.plot(dates2, tot_prod_z, label='33/9-L-1 H')

date_fmt = mdates.DateFormatter('%m/%Y')
ax.xaxis.set_major_formatter(date_fmt)
plt.xticks(rotation=45)

# Set plot title and axis labels
plt.title('Production data from Januar 2002 to December 2010.')
plt.legend()
plt.xlabel('Year and Month')
plt.ylabel('Oil and gas [Sm3]')

plt.show()
```



1.1 Filtering

Filter the signals to remove the big spikes before deriving the data. Using a median filter to smooth the data.

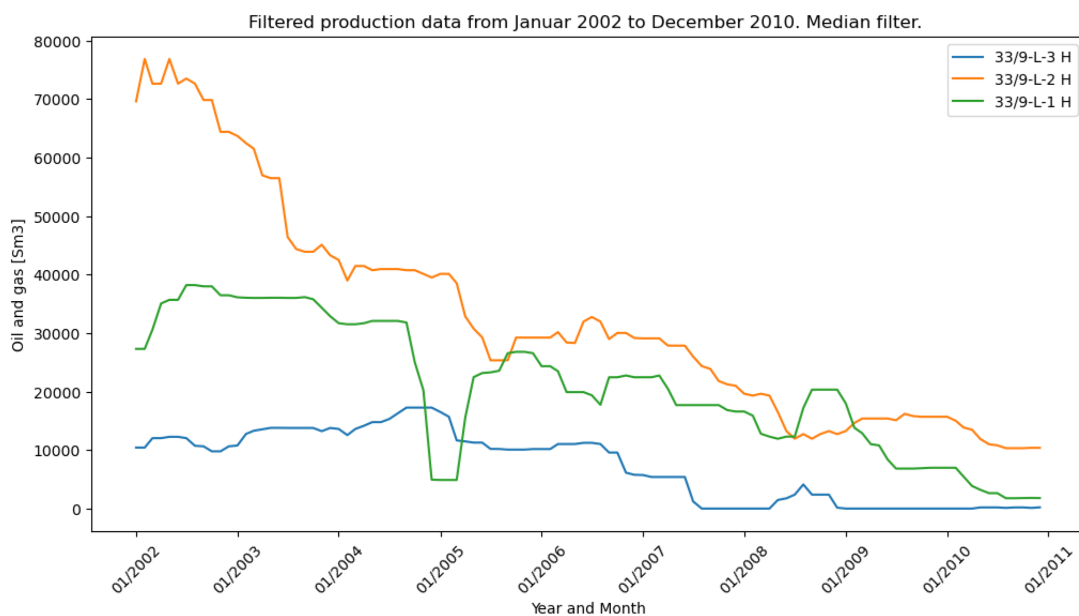
Trying out the savgol_filter below.

[11]:
```
md_x = median_filter(tot_prod_x, size=6)

md_y = median_filter(tot_prod_y, size=6)

md_z = median_filter(tot_prod_z, size=6)
```

[12]:
```
from scipy.signal import savgol_filter

sf_x = savgol_filter(tot_prod_x, 10, 1, mode='nearest')

sf_y = savgol_filter(tot_prod_y, 10, 1, mode='nearest')

sf_z = savgol_filter(tot_prod_z, 10, 1, mode='nearest')
```

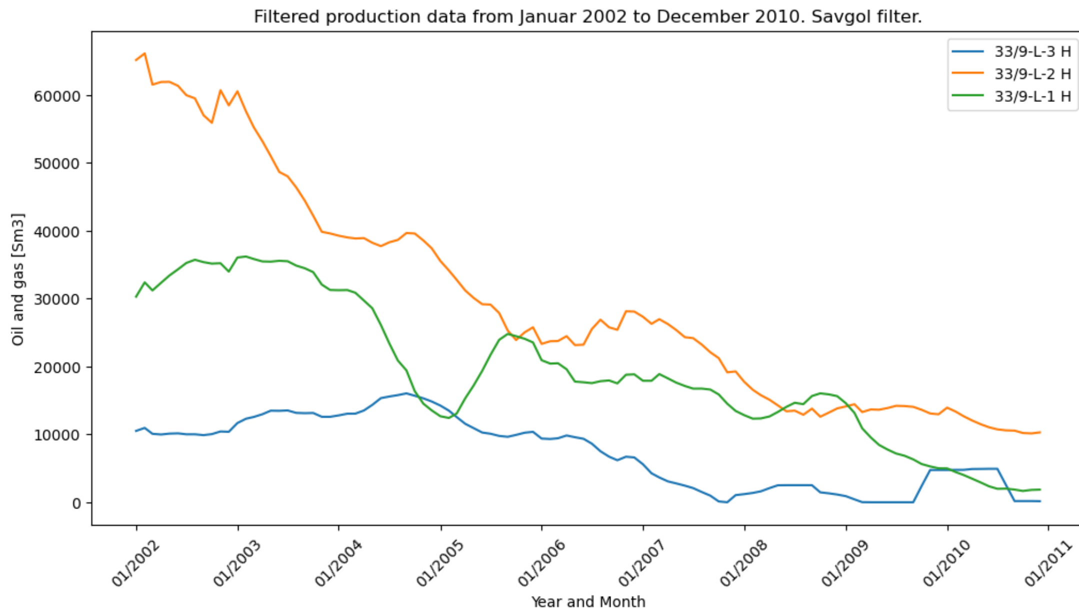
[13]:
```
fig, ax = plt.subplots(figsize=(12, 6))
ax.plot(dates2, md_x, label='33/9-L-3 H')
ax.plot(dates2, md_y, label='33/9-L-2 H')
ax.plot(dates2, md_z, label='33/9-L-1 H')

date_fmt = mdates.DateFormatter('%m/%Y')
```

```
ax.xaxis.set_major_formatter(date_fmt)
plt.xticks(rotation=45)

# Set plot title and axis labels
plt.title('Filtered production data from Januar 2002 to December 2010. Median␣
↪filter. ')
plt.legend()
plt.xlabel('Year and Month')
plt.ylabel('Oil and gas [Sm3]')

plt.show()
```



[14]:
```
fig, ax = plt.subplots(figsize=(12, 6))
ax.plot(dates2, sf_x, label='33/9-L-3 H')
ax.plot(dates2, sf_y, label='33/9-L-2 H')
ax.plot(dates2, sf_z, label='33/9-L-1 H')

date_fmt = mdates.DateFormatter('%m/%Y')
ax.xaxis.set_major_formatter(date_fmt)
plt.xticks(rotation=45)

# Set plot title and axis labels
plt.title('Filtered production data from Januar 2002 to December 2010. Savgol␣
↪filter.')
plt.legend()
plt.xlabel('Year and Month')
```

```
plt.ylabel('Oil and gas [Sm3]')

plt.show()
```



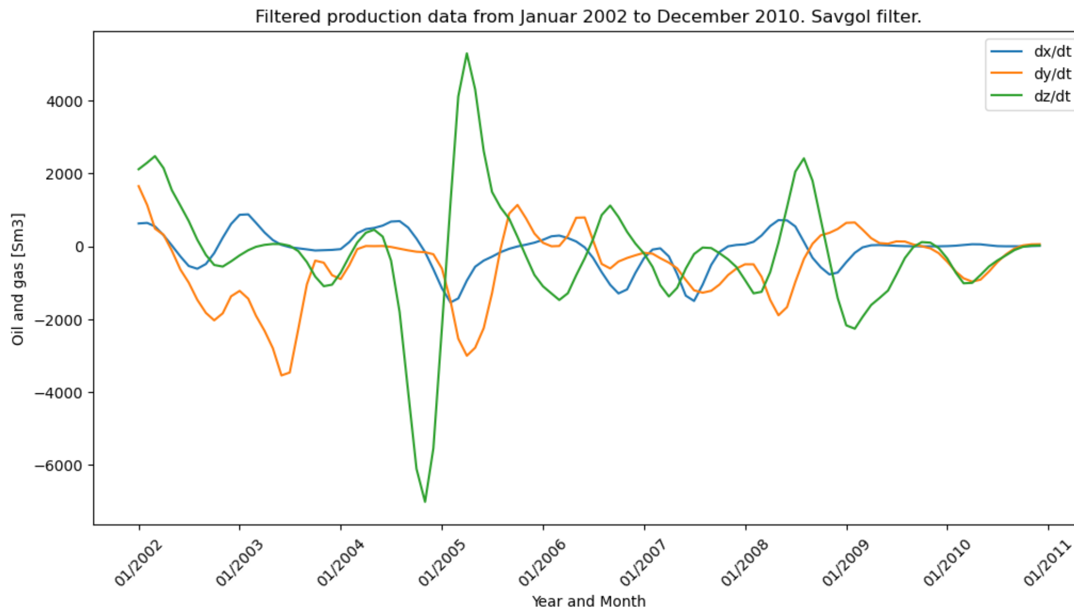
[15]:
```
i = len(dates2)
t = list(range(1, i+1))
t_np = np.array(t)
```

[16]:
```
md_x_np = np.array(md_x)
md_y_np = np.array(md_y)
md_z_np = np.array(md_z)
```

[17]:
```
x_der = dxdt(md_x_np, t_np, kind="kalman", alpha=2)
y_der = dxdt(md_y_np, t_np, kind="kalman", alpha=2)
z_der = dxdt(md_z_np, t_np, kind="kalman", alpha=2)
```

[18]:
```
fig, ax = plt.subplots(figsize=(12, 6))
ax.plot(dates2, x_der, label='dx/dt')
ax.plot(dates2, y_der, label='dy/dt')
ax.plot(dates2, z_der, label='dz/dt')

date_fmt = mdates.DateFormatter('%m/%Y')
ax.xaxis.set_major_formatter(date_fmt)
plt.xticks(rotation=45)

# Set plot title and axis labels
```

```
plt.title('Filtered production data from Januar 2002 to December 2010. Savgol␣
↪filter.')
plt.legend()
plt.xlabel('Year and Month')
plt.ylabel('Oil and gas [Sm3]')

plt.show()
```



2 SINDy

[19]:
```
mat = np.zeros((108,3))

mat[:, 0] = md_x
mat[:, 1] = md_y
mat[:, 2] = md_z
```

[20]:
```
mat_der = np.zeros((108,3))

mat_der[:, 0] = x_der
mat_der[:, 1] = y_der
mat_der[:, 2] = z_der
```

[21]:
```
def create_library(u: np.ndarray, polynomial_order: int,
                   use_trig: bool, use_exp: bool) -> np.ndarray:
    """Creates a matrix containing a library of candidate functions.
```

```
    For example, if our u depends on x, y, and z, and we specify
    polynomial_order=2, use_exp=True and use_trig=False, our terms would be:
    1, x, y, z, x^2, xy, xz, y^2, yz, z^2, exp(-t*x), exp(-t*y), exp(-t*z),
↪exp(-t*x^2), exp(-t*y^2), exp(-t*z^2)
    """
    (m, n) = u.shape
    theta = np.ones((m, 1))

    # Polynomials of order 1.
    theta = np.hstack((theta, u))

    # Polynomials of order 2.
    if polynomial_order >= 2:
        for i in range(n):
            for j in range(i, n):
                theta = np.hstack((theta, u[:, i:i + 1] * u[:, j:j + 1]))

    # Polynomials of order 3.
    if polynomial_order >= 3:
        for i in range(n):
            for j in range(i, n):
                for k in range(j, n):
                    theta = np.hstack(
                        (theta, u[:, i:i + 1] * u[:, j:j + 1] * u[:, k:k + 1]))

    # Polynomials of order 4.
    if polynomial_order >= 4:
        for i in range(n):
            for j in range(i, n):
                for k in range(j, n):
                    for l in range(k, n):
                        theta = np.hstack(
                            (theta, u[:, i:i + 1] * u[:, j:j + 1] *
                             u[:, k:k + 1] * u[:, l:l + 1]))

    # Polynomials of order 5.
    if polynomial_order >= 5:
        for i in range(n):
            for j in range(i, n):
                for k in range(j, n):
                    for l in range(k, n):
                        for m in range(l, n):
                            theta = np.hstack(
                                (theta, u[:, i:i + 1] * u[:, j:j + 1] *
                                 u[:, k:k + 1] * u[:, l:l + 1] * u[:, m:m + 1]))

    if use_trig:
```

```
        for i in range(1,2):
            theta = np.hstack((theta, np.sin(i * u), np.cos(i * u)))

    if use_exp:
        for i in range(n):
            theta = np.hstack((theta, np.exp(-u[:, i:i+1]), np.exp(-u[:, i:
↪i+1]**2)))

    return theta
```

[22]:
```
def approximation(_: float, u: np.ndarray, xi: np.ndarray,
                  polynomial_order: int, use_trig: bool, use_exp: bool)␣
↪-> np.ndarray:
    theta = create_library(u.reshape((1, 3)), polynomial_order, use_trig,␣
↪use_exp)
    return theta @ xi

def compute_trajectory(u0: np.ndarray, xi: np.ndarray, polynomial_order: int,
                       use_trig: bool, use_exp: bool) -> np.ndarray:
    t0 = 0
    dt = 1
    tmax = 108
    n = int(tmax / dt + 1)

    t = np.linspace(start=t0, stop=tmax, num=n)
    result = solve_ivp(fun=approximation,
                       t_span=(t0, tmax),
                       y0=u0,
                       t_eval=t,
                       args=(xi, polynomial_order, use_trig, use_exp))
    u = result.y.T

    return u
```

[23]:
```
def calculate_regression(theta: np.ndarray, uprime: np.ndarray,
                         threshold: float, max_iterations: int) -> np.ndarray:
    # Solve theta * xi = uprime in the least-squares sense.
    xi = np.linalg.lstsq(theta, uprime, rcond=None)[0]
    n = xi.shape[1]

    # Add sparsity.
    for _ in range(max_iterations):
        small_indices = np.abs(xi) < threshold
        xi[small_indices] = 0
        for j in range(n):
            big_indices = np.logical_not(small_indices[:, j])
            xi[big_indices, j] = np.linalg.lstsq(theta[:, big_indices],
```

```
                                                    uprime[:, j],
                                                    rcond=None)[0]

    return xi
```

[31]:
```
POLYNOMIAL_ORDER = 2
USE_TRIG = False
USE_EXP = False

theta = create_library(mat, POLYNOMIAL_ORDER, USE_TRIG, USE_EXP)
```

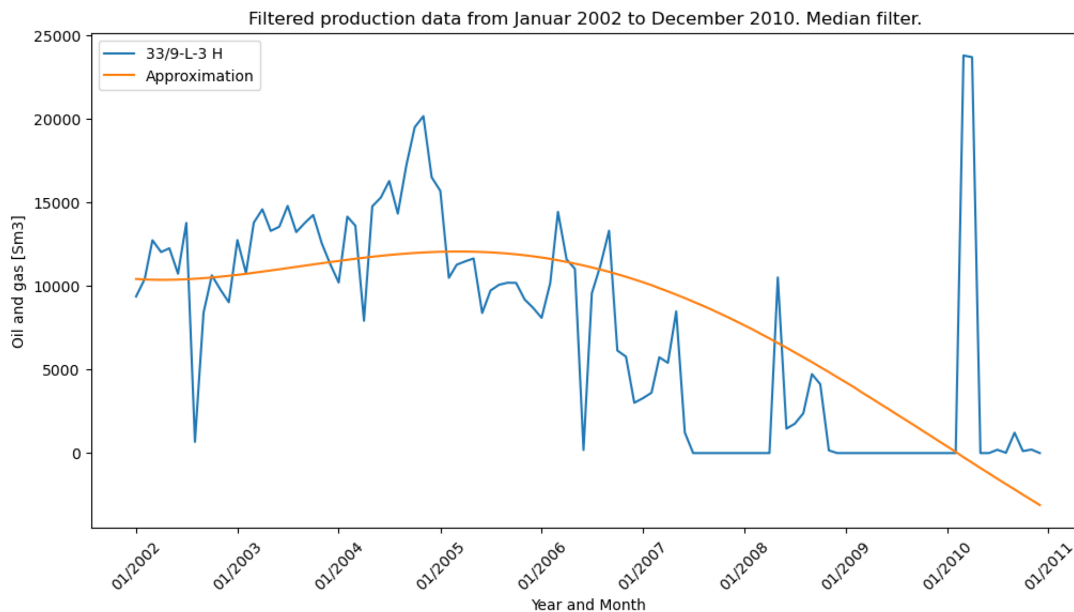
[32]:
```
THRESHOLD = 0.01
MAX_ITERATIONS = 100

xi = calculate_regression(theta, mat_der, THRESHOLD, MAX_ITERATIONS)
xi
```

[32]:
```
array([[-3.22409827e+02, -1.43103085e+02, -6.56608204e+02],
       [-1.65597332e-02,  0.00000000e+00, -7.43393900e-02],
       [ 0.00000000e+00, -1.27730347e-02,  2.99814973e-02],
       [ 1.69081543e-02,  0.00000000e+00,  0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00]])
```

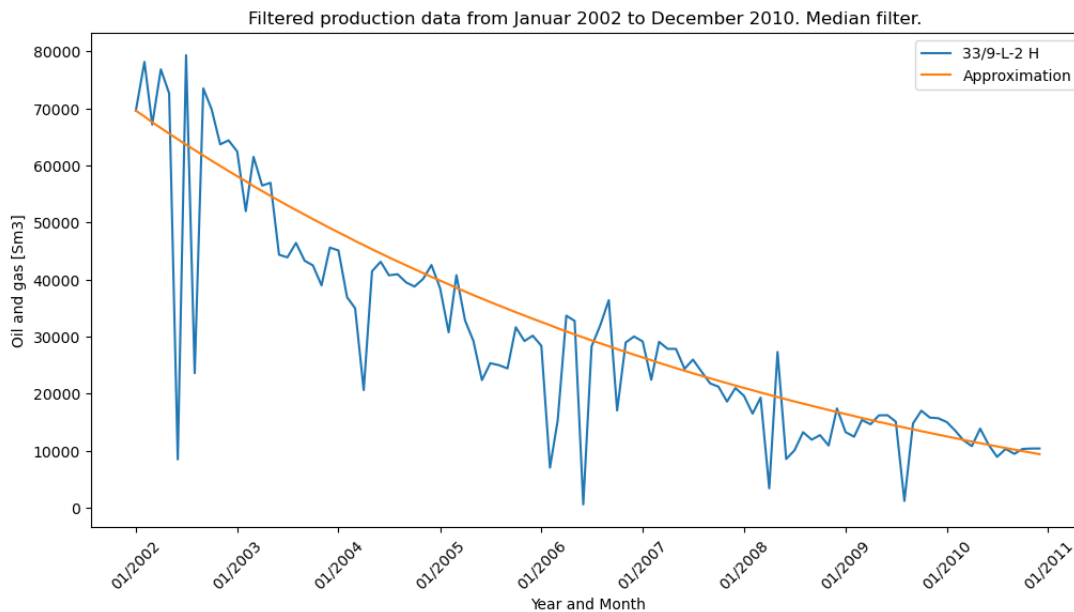
[33]:
```
mat_0 = mat[0]

u_approximation = compute_trajectory(mat_0, xi, POLYNOMIAL_ORDER, USE_TRIG,␣
 ↪USE_EXP)
x_aprox = u_approximation[:,0]
y_aprox = u_approximation[:,1]
z_aprox = u_approximation[:,2]
```

[34]:
```
fig, ax = plt.subplots(figsize=(12, 6))
ax.plot(dates2, tot_prod_x, label='33/9-L-3 H')
ax.plot(dates2, x_aprox[:-1], label='Approximation')


date_fmt = mdates.DateFormatter('%m/%Y')
ax.xaxis.set_major_formatter(date_fmt)
plt.xticks(rotation=45)

# Set plot title and axis labels
```

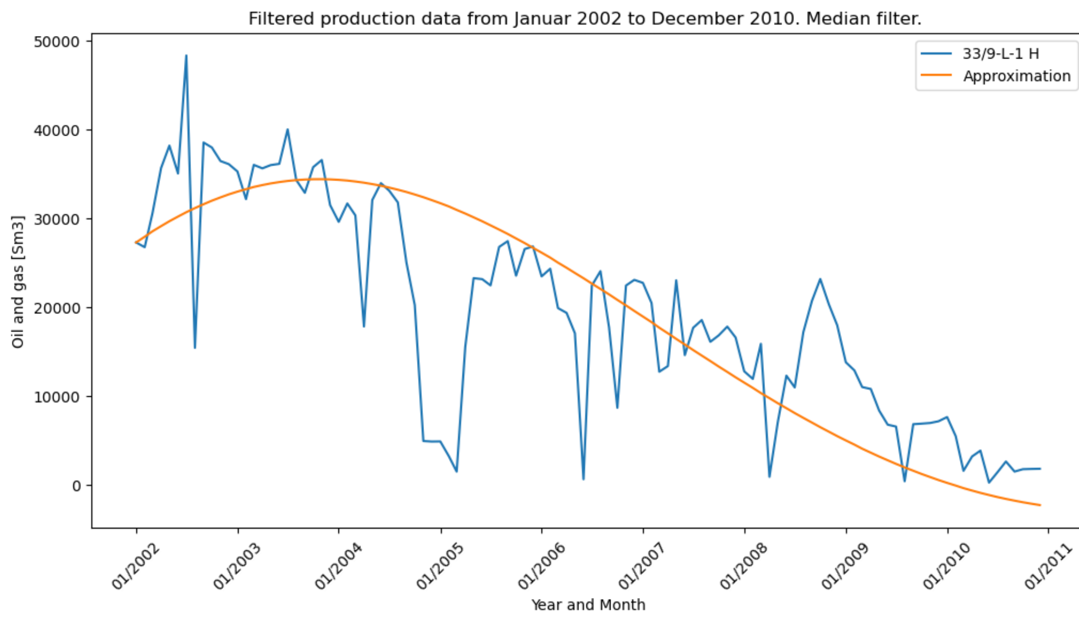
```
plt.title('Filtered production data from Januar 2002 to December 2010. Median␣
↪filter. ')
plt.legend()
plt.xlabel('Year and Month')
plt.ylabel('Oil and gas [Sm3]')

plt.show()
```



[35]:
```
fig, ax = plt.subplots(figsize=(12, 6))
ax.plot(dates2, tot_prod_y, label='33/9-L-2 H')
ax.plot(dates2, y_aprox[:-1], label='Approximation')


date_fmt = mdates.DateFormatter('%m/%Y')
ax.xaxis.set_major_formatter(date_fmt)
plt.xticks(rotation=45)

# Set plot title and axis labels
plt.title('Filtered production data from Januar 2002 to December 2010. Median␣
↪filter. ')
plt.legend()
plt.xlabel('Year and Month')
plt.ylabel('Oil and gas [Sm3]')

plt.show()
```



[36]:
```
fig, ax = plt.subplots(figsize=(12, 6))
ax.plot(dates2, tot_prod_z, label='33/9-L-1 H')
ax.plot(dates2, z_aprox[:-1], label='Approximation')


date_fmt = mdates.DateFormatter('%m/%Y')
ax.xaxis.set_major_formatter(date_fmt)
plt.xticks(rotation=45)

# Set plot title and axis labels
plt.title('Filtered production data from Januar 2002 to December 2010. Median␣
↪filter. ')
plt.legend()
plt.xlabel('Year and Month')
plt.ylabel('Oil and gas [Sm3]')

plt.show()
```


Filtered production data from Januar 2002 to December 2010. Median filter.
33/9-L-1 H
Approximation
Oil and gas [Sm3]
50000
40000
30000
20000
10000
0
01/2002
01/2003
01/2004
01/2005
01/2006
01/2007
01/2008
01/2009
01/2010
01/2011
Year and Month

```
[ ]:
```

A.4 Code for SINDy on Draugen

draugen

June 14, 2023

1 Draugen

[1]:
```
# Import modules

import numpy as np
import pandas as pd
import requests
import io
import datetime
from scipy.integrate import solve_ivp
from derivative import dxdt
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
from scipy.ndimage import median_filter
```

[2]:
```
url = 'https://raw.githubusercontent.com/SanderSondeland/Master/main/
  ↪20230418_WellBore_monthlyFacility%20(1).csv?
  ↪token=GHSAT0AAAAAACD4GTTLM35KNNFLKBPUPOWMZEI5SXA'
download = requests.get(url).content
data = pd.read_csv(io.StringIO(download.decode('utf-8')))
data
```

[2]:
```
              name  npdId    field  year  month  operationTime  \
0       6407/9-A-1   2254  DRAUGEN  1994      6            NaN
1       6407/9-A-1   2254  DRAUGEN  1994      7            NaN
2       6407/9-A-1   2254  DRAUGEN  1994      8            NaN
3       6407/9-A-1   2254  DRAUGEN  1994      9            NaN
4       6407/9-A-1   2254  DRAUGEN  1994     10            NaN
…              …     …        …     …     …              …
3700  6407/9-G-5 H   7715  DRAUGEN  2020      8          31.00
3701  6407/9-G-5 H   7715  DRAUGEN  2020      9          28.93
3702  6407/9-G-5 H   7715  DRAUGEN  2020     10          25.54
3703  6407/9-G-5 H   7715  DRAUGEN  2020     11          30.00
3704  6407/9-G-5 H   7715  DRAUGEN  2020     12          31.00

     operationTimeUom wellStatus     oil oilUom        gas gasUom  condensate  \
0                   d        NaN  172347    Sm3    7931000    Sm3         NaN
```

```
1                     d        NaN  198022    Sm3  10221000    Sm3          NaN
2                     d        NaN  201457    Sm3  11584000    Sm3          NaN
3                     d        NaN  197711    Sm3  11087000    Sm3          NaN
4                     d        NaN  195050    Sm3  10988000    Sm3          NaN
…                …       …     …      …       …     …      …
3700                  d  producing    8851    Sm3    304029    Sm3          NaN
3701                  d  producing    8493    Sm3    270559    Sm3          NaN
3702                  d  producing    7268    Sm3    245919    Sm3          NaN
3703                  d  producing    8109    Sm3    257430    Sm3          NaN
3704                  d  producing    9069    Sm3    284342    Sm3          NaN

     condensateUom  water waterUom
0              Sm3      0      Sm3
1              Sm3      0      Sm3
2              Sm3      0      Sm3
3              Sm3      0      Sm3
4              Sm3      0      Sm3
…             …    …       …
3700           Sm3  66253      Sm3
3701           Sm3  66175      Sm3
3702           Sm3  62792      Sm3
3703           Sm3  73987      Sm3
3704           Sm3  79158      Sm3

[3705 rows x 16 columns]
```

[3]:
```
data.name.unique()
```

[3]:
```
array(['6407/9-A-1', '6407/9-A-2 A', '6407/9-A-3', '6407/9-A-4',
       '6407/9-A-4 A', '6407/9-A-5', '6407/9-A-53 H', '6407/9-A-55 AH',
       '6407/9-A-6', '6407/9-D-1 AH', '6407/9-D-2 H', '6407/9-D-3 H',
       '6407/9-E-1 H', '6407/9-E-2 H', '6407/9-E-3 H', '6407/9-E-4 H',
       '6407/9-G-1 H', '6407/9-G-2 H', '6407/9-G-3 H', '6407/9-G-5 H'],
      dtype=object)
```

[4]:
```
data1 = data.drop(columns=['npdId', 'field', 'operationTime',␣
 ↪'operationTimeUom','wellStatus','oilUom', 'gasUom', 'condensate',
'condensateUom', 'waterUom'])
data1
```

[4]:
```
            name  year  month     oil       gas  water
0     6407/9-A-1  1994      6  172347   7931000      0
1     6407/9-A-1  1994      7  198022  10221000      0
2     6407/9-A-1  1994      8  201457  11584000      0
3     6407/9-A-1  1994      9  197711  11087000      0
4     6407/9-A-1  1994     10  195050  10988000      0
…            …    …      …       …         …     …
```

```
3700  6407/9-G-5 H  2020     8    8851    304029  66253
3701  6407/9-G-5 H  2020     9    8493    270559  66175
3702  6407/9-G-5 H  2020    10    7268    245919  62792
3703  6407/9-G-5 H  2020    11    8109    257430  73987
3704  6407/9-G-5 H  2020    12    9069    284342  79158

[3705 rows x 6 columns]
```

[5]:
```
x_data = data1[data1['name'] == '6407/9-A-1']
x_data.loc[:, 'period'] = x_data['year'].astype(str) + '-' + x_data['month'].
 ↪astype(str)
x_data.loc[:, 'tot_prod'] = (x_data['oil'] + x_data['gas']/1000).round(1)
x_data = x_data.groupby('period').sum()
x_data
```

```
C:\Users\sande\AppData\Local\Temp\ipykernel_16680\3989910507.py:2:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  x_data.loc[:, 'period'] = x_data['year'].astype(str) + '-' +
x_data['month'].astype(str)
C:\Users\sande\AppData\Local\Temp\ipykernel_16680\3989910507.py:3:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  x_data.loc[:, 'tot_prod'] = (x_data['oil'] + x_data['gas']/1000).round(1)
C:\Users\sande\AppData\Local\Temp\ipykernel_16680\3989910507.py:4:
FutureWarning: The default value of numeric_only in DataFrameGroupBy.sum is
deprecated. In a future version, numeric_only will default to False. Either
specify numeric_only or select only columns which should be valid for the
function.
  x_data = x_data.groupby('period').sum()
```

[5]:
```
          year  month     oil       gas   water  tot_prod
period
1994-10   1994     10  195050  10988000       0  206038.0
1994-11   1994     11  196044  10255000       0  206299.0
1994-12   1994     12  176032   9252000       0  185284.0
1994-6    1994      6  172347   7931000       0  180278.0
1994-7    1994      7  198022  10221000       0  208243.0
…         …     …      …       …        …       …
```

```
2020-5    2020       5     6446     420354   132877     6866.4
2020-6    2020       6     4180     279594    95597     4459.6
2020-7    2020       7     2172     143060    40738     2315.1
2020-8    2020       8     6258     352751   124142     6610.8
2020-9    2020       9     5639     295011   115914     5934.0

[318 rows x 6 columns]
```

[6]:

```python
y_data = data1[data1['name'] == '6407/9-A-2 A']
y_data.loc[:, 'period'] = y_data['year'].astype(str) + '-' + y_data['month'].
 ↪astype(str)
y_data.loc[:, 'tot_prod'] = (y_data['oil'] + y_data['gas']/1000).round(1)
y_data.groupby('period').sum()
y_data
```

```
C:\Users\sande\AppData\Local\Temp\ipykernel_16680\4226326607.py:2:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  y_data.loc[:, 'period'] = y_data['year'].astype(str) + '-' +
y_data['month'].astype(str)
C:\Users\sande\AppData\Local\Temp\ipykernel_16680\4226326607.py:3:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  y_data.loc[:, 'tot_prod'] = (y_data['oil'] + y_data['gas']/1000).round(1)
C:\Users\sande\AppData\Local\Temp\ipykernel_16680\4226326607.py:4:
FutureWarning: The default value of numeric_only in DataFrameGroupBy.sum is
deprecated. In a future version, numeric_only will default to False. Either
specify numeric_only or select only columns which should be valid for the
function.
  y_data.groupby('period').sum()
```

[6]:

```
              name  year  month     oil      gas   water   period  tot_prod
318   6407/9-A-2 A  1994     12   72310  3802000       0  1994-12   76112.0
319   6407/9-A-2 A  1995      1  107169  5585000       0   1995-1  112754.0
320   6407/9-A-2 A  1995      2  137400  7487000       0   1995-2  144887.0
321   6407/9-A-2 A  1995      3  128699  7202000       0   1995-3  135901.0
322   6407/9-A-2 A  1995      4  142063  7908000       0   1995-4  149971.0
..             …     …      …       …       …        …        …         …
625   6407/9-A-2 A  2020      8    5055   287806  164523   2020-8    5342.8
```

```
626   6407/9-A-2 A   2020      9     6422    336610   162410    2020-9     6758.6
627   6407/9-A-2 A   2020     10     6954    381451   171507   2020-10     7335.5
628   6407/9-A-2 A   2020     11     6040    315625   157900   2020-11     6355.6
629   6407/9-A-2 A   2020     12     6962    358310   170456   2020-12     7320.3

[312 rows x 8 columns]
```

[7]:

```
z_data = data1[data1['name'] == '6407/9-A-6']
z_data.loc[:, 'period'] = z_data['year'].astype(str) + '-' + z_data['month'].
↪astype(str)
z_data.loc[:, 'tot_prod'] = (z_data['oil'] + z_data['gas']/1000).round(1)
z_data.groupby('period').sum()
z_data
```

```
C:\Users\sande\AppData\Local\Temp\ipykernel_16680\2187305751.py:2:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  z_data.loc[:, 'period'] = z_data['year'].astype(str) + '-' +
z_data['month'].astype(str)
C:\Users\sande\AppData\Local\Temp\ipykernel_16680\2187305751.py:3:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  z_data.loc[:, 'tot_prod'] = (z_data['oil'] + z_data['gas']/1000).round(1)
C:\Users\sande\AppData\Local\Temp\ipykernel_16680\2187305751.py:4:
FutureWarning: The default value of numeric_only in DataFrameGroupBy.sum is
deprecated. In a future version, numeric_only will default to False. Either
specify numeric_only or select only columns which should be valid for the
function.
  z_data.groupby('period').sum()
```

[7]:

	name	year	month	oil	gas	water	period	tot_prod
1897	6407/9-A-6	1994	8	206901	11608000	0	1994-8	218509.0
1898	6407/9-A-6	1994	9	215894	12106000	0	1994-9	228000.0
1899	6407/9-A-6	1994	10	221972	12504000	0	1994-10	234476.0
1900	6407/9-A-6	1994	11	211153	11045000	0	1994-11	222198.0
1901	6407/9-A-6	1994	12	220564	11594000	0	1994-12	232158.0
…	…	…	…	…	…	…	…	…
2208	6407/9-A-6	2020	8	14463	815989	173401	2020-8	15279.0
2209	6407/9-A-6	2020	9	15132	791996	166646	2020-9	15924.0

```
2210  6407/9-A-6  2020     10   15493   2185778  162246  2020-10   17678.8
2211  6407/9-A-6  2020     11   13065   2184933  156661  2020-11   15249.9
2212  6407/9-A-6  2020     12   14127   2428084  158360  2020-12   16555.1

[316 rows x 8 columns]
```

[8]:

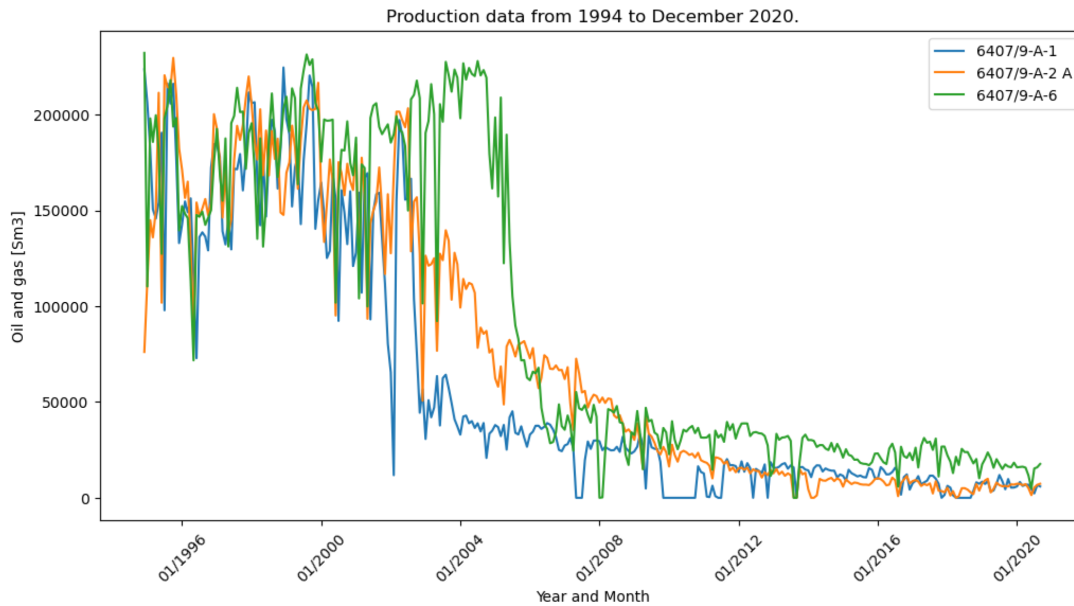
```
dates = [datetime.datetime(year=int(year), month=int(month), day=1) for year in
 range(1994, 2021) for month in range(1, 13)]
dates = dates[11:-3]
```

[9]:

```
test_x = x_data[8:]
tot_prod_x = test_x['tot_prod']

test_y = y_data[:-2]
tot_prod_y = test_y['tot_prod']


test_z = z_data[4:-2]
tot_prod_z = test_z['tot_prod']
```

[10]:

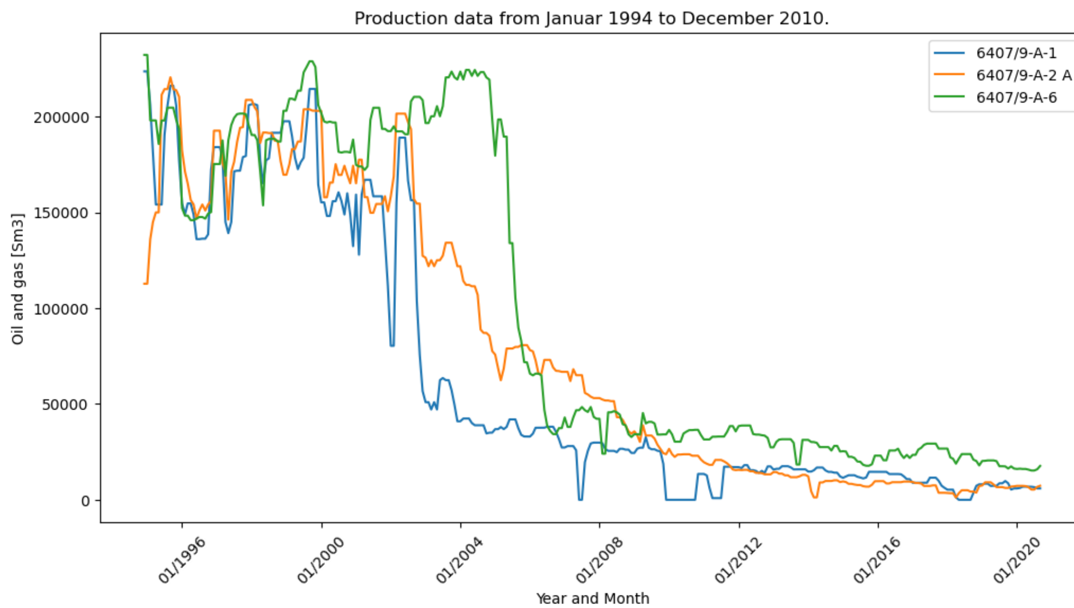
```
fig, ax = plt.subplots(figsize=(12, 6))
ax.plot(dates, tot_prod_x, label='6407/9-A-1')
ax.plot(dates, tot_prod_y, label='6407/9-A-2 A')
ax.plot(dates, tot_prod_z, label='6407/9-A-6')

date_fmt = mdates.DateFormatter('%m/%Y')
ax.xaxis.set_major_formatter(date_fmt)
plt.xticks(rotation=45)

# Set plot title and axis labels
plt.title('Production data from 1994 to December 2020.')
plt.legend()
plt.xlabel('Year and Month')
plt.ylabel('Oil and gas [Sm3]')

plt.show()
```



2 Filtering

[11]:
```
md_x = median_filter(tot_prod_x, size=4)

md_y = median_filter(tot_prod_y, size=4)

md_z = median_filter(tot_prod_z, size=4)
```

[12]:
```
fig, ax = plt.subplots(figsize=(12, 6))
ax.plot(dates, md_x, label='6407/9-A-1')
ax.plot(dates, md_y, label='6407/9-A-2 A')
ax.plot(dates, md_z, label='6407/9-A-6')

date_fmt = mdates.DateFormatter('%m/%Y')
ax.xaxis.set_major_formatter(date_fmt)
plt.xticks(rotation=45)

# Set plot title and axis labels
plt.title('Production data from Januar 1994 to December 2010.')
plt.legend()
plt.xlabel('Year and Month')
plt.ylabel('Oil and gas [Sm3]')

plt.show()
```



[13]:
```
i = len(dates)
t = list(range(1, i+1))
t_np = np.array(t)
```

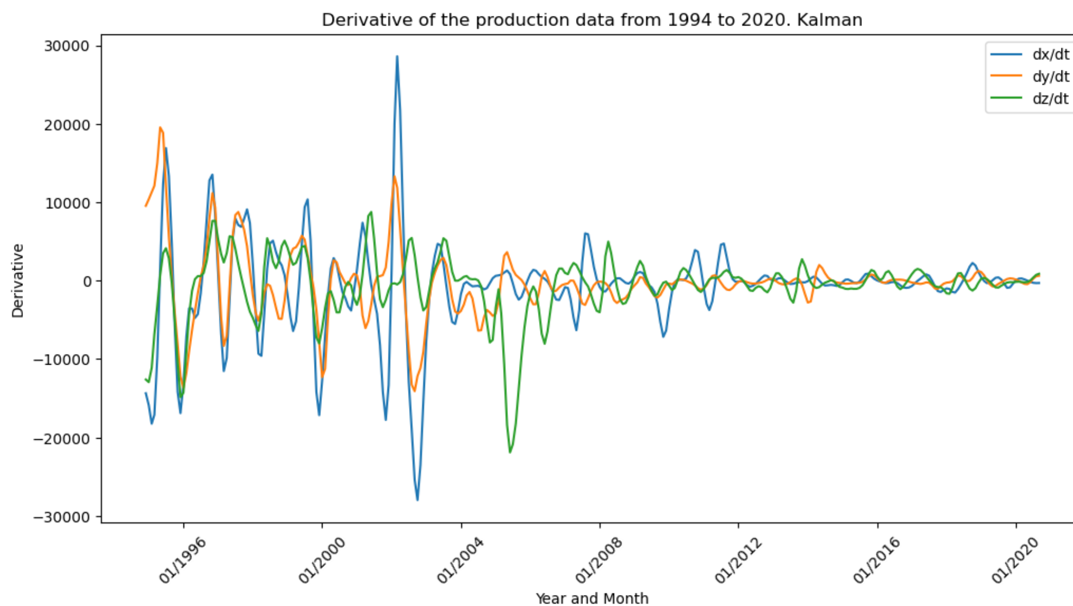
[14]:
```
md_x_np = np.array(md_x)
md_y_np = np.array(md_y)
md_z_np = np.array(md_z)

tot_prod_x_np = np.array(tot_prod_x)
tot_prod_y_np = np.array(tot_prod_y)
tot_prod_z_np = np.array(tot_prod_z)
```

[15]:
```
x_der = dxdt(md_x_np, t_np, kind="kalman", alpha=2)
y_der = dxdt(md_y_np, t_np, kind="kalman", alpha=2)
z_der = dxdt(md_z_np, t_np, kind="kalman", alpha=2)
```

[16]:
```
x_der2 = dxdt(md_x_np, t_np, kind="finite_difference", k=2)
y_der2 = dxdt(md_y_np, t_np, kind="finite_difference", k=2)
z_der2 = dxdt(md_z_np, t_np, kind="finite_difference", k=2)
```

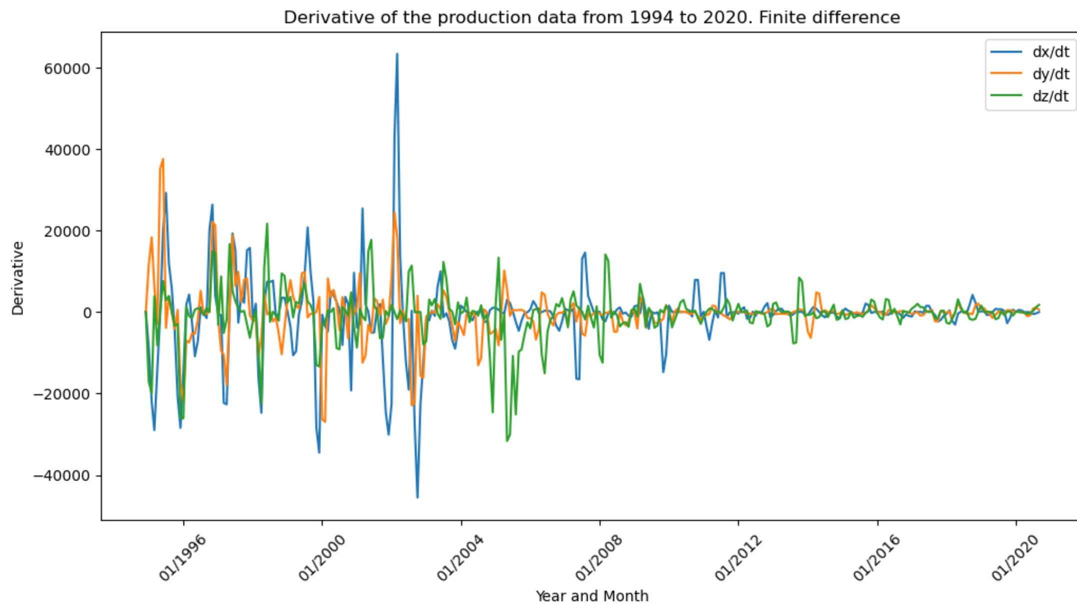
[19]:
```
fig, ax = plt.subplots(figsize=(12, 6))
ax.plot(dates, x_der, label='dx/dt')
ax.plot(dates, y_der, label='dy/dt')
ax.plot(dates, z_der, label='dz/dt')

date_fmt = mdates.DateFormatter('%m/%Y')
```

```
ax.xaxis.set_major_formatter(date_fmt)
plt.xticks(rotation=45)

# Set plot title and axis labels
plt.title('Derivative of the production data from 1994 to 2020. Kalman')
plt.legend()
plt.xlabel('Year and Month')
plt.ylabel('Derivative')

plt.show()
```



[21]:
```
fig, ax = plt.subplots(figsize=(12, 6))
ax.plot(dates, x_der2, label='dx/dt')
ax.plot(dates, y_der2, label='dy/dt')
ax.plot(dates, z_der2, label='dz/dt')

date_fmt = mdates.DateFormatter('%m/%Y')
ax.xaxis.set_major_formatter(date_fmt)
plt.xticks(rotation=45)

# Set plot title and axis labels
plt.title('Derivative of the production data from 1994 to 2020. Finite␣
↪difference')
plt.legend()
plt.xlabel('Year and Month')
plt.ylabel('Derivative')
```

```
plt.show()
```



3 SINDy

[20]:
```
mat = np.zeros((310,3))

mat[:, 0] = md_x
mat[:, 1] = md_y
mat[:, 2] = md_z

mat.shape
```

[20]: (310, 3)

[21]:
```
mat_der = np.zeros((310,3))

mat_der[:, 0] = x_der
mat_der[:, 1] = y_der
mat_der[:, 2] = z_der

mat_der.shape
```

[21]: (310, 3)

```python
[22]: def create_library(u: np.ndarray, polynomial_order: int,
                       use_trig: bool, use_exp: bool) -> np.ndarray:
          """Creates a matrix containing a library of candidate functions.
          """
          (m, n) = u.shape
          theta = np.ones((m, 1))

          # Polynomials of order 1.
          theta = np.hstack((theta, u))

          # Polynomials of order 2.
          if polynomial_order >= 2:
              for i in range(n):
                  for j in range(i, n):
                      theta = np.hstack((theta, u[:, i:i + 1] * u[:, j:j + 1]))

          # Polynomials of order 3.
          if polynomial_order >= 3:
              for i in range(n):
                  for j in range(i, n):
                      for k in range(j, n):
                          theta = np.hstack(
                              (theta, u[:, i:i + 1] * u[:, j:j + 1] * u[:, k:k + 1]))

          # Polynomials of order 4.
          if polynomial_order >= 4:
              for i in range(n):
                  for j in range(i, n):
                      for k in range(j, n):
                          for l in range(k, n):
                              theta = np.hstack(
                                  (theta, u[:, i:i + 1] * u[:, j:j + 1] *
                                   u[:, k:k + 1] * u[:, l:l + 1]))

          # Polynomials of order 5.
          if polynomial_order >= 5:
              for i in range(n):
                  for j in range(i, n):
                      for k in range(j, n):
                          for l in range(k, n):
                              for m in range(l, n):
                                  theta = np.hstack(
                                      (theta, u[:, i:i + 1] * u[:, j:j + 1] *
                                       u[:, k:k + 1] * u[:, l:l + 1] * u[:, m:m + 1]))

          if use_trig:
              for i in range(1, 11):
```

```
            theta = np.hstack((theta, np.sin(i * u), np.cos(i * u)))

    if use_exp:
        for i in range(n):
            theta = np.hstack((theta, np.exp(-u[:, i:i+1]), np.exp(-u[:, i:
↪i+1]**2)))

    return theta
```

[23]:
```
def approximation(_: float, u: np.ndarray, xi: np.ndarray,
                         polynomial_order: int, use_trig: bool, use_exp: bool)␣
↪-> np.ndarray:
    theta = create_library(u.reshape((1, 3)), polynomial_order, use_trig,␣
↪use_exp)
    return theta @ xi

def compute_trajectory(u0: np.ndarray, xi: np.ndarray, polynomial_order: int,
                       use_trig: bool, use_exp: bool) -> np.ndarray:
    t0 = 0
    dt = 1
    tmax = 310
    n = int(tmax / dt + 1)

    t = np.linspace(start=t0, stop=tmax, num=n)
    result = solve_ivp(fun=approximation,
                       t_span=(t0, tmax),
                       y0=u0,
                       t_eval=t,
                       args=(xi, polynomial_order, use_trig, use_exp))
    u = result.y.T

    return u
```

[24]:
```
def calculate_regression(theta: np.ndarray, uprime: np.ndarray,
                         threshold: float, max_iterations: int) -> np.ndarray:
    # Solve theta * xi = uprime in the least-squares sense.
    xi = np.linalg.lstsq(theta, uprime, rcond=None)[0]
    n = xi.shape[1]

    # Add sparsity.
    for _ in range(max_iterations):
        small_indices = np.abs(xi) < threshold
        xi[small_indices] = 0
        for j in range(n):
            big_indices = np.logical_not(small_indices[:, j])
            xi[big_indices, j] = np.linalg.lstsq(theta[:, big_indices],
                                                 uprime[:, j],
```

```
                                                   rcond=None)[0]

    return xi
```

[25]:
```
POLYNOMIAL_ORDER = 2
USE_TRIG = False
USE_EXP = False
T_ORDER = 1

t_np_test = np.array(range(1, 311))

theta = create_library(mat, POLYNOMIAL_ORDER, USE_TRIG, USE_EXP)
theta.size
```

[25]:
```
3100
```

[26]:
```
THRESHOLD = 0.005
MAX_ITERATIONS = 10

xi = calculate_regression(theta, mat_der, THRESHOLD, MAX_ITERATIONS)
xi
```

```
(10, 3)
```

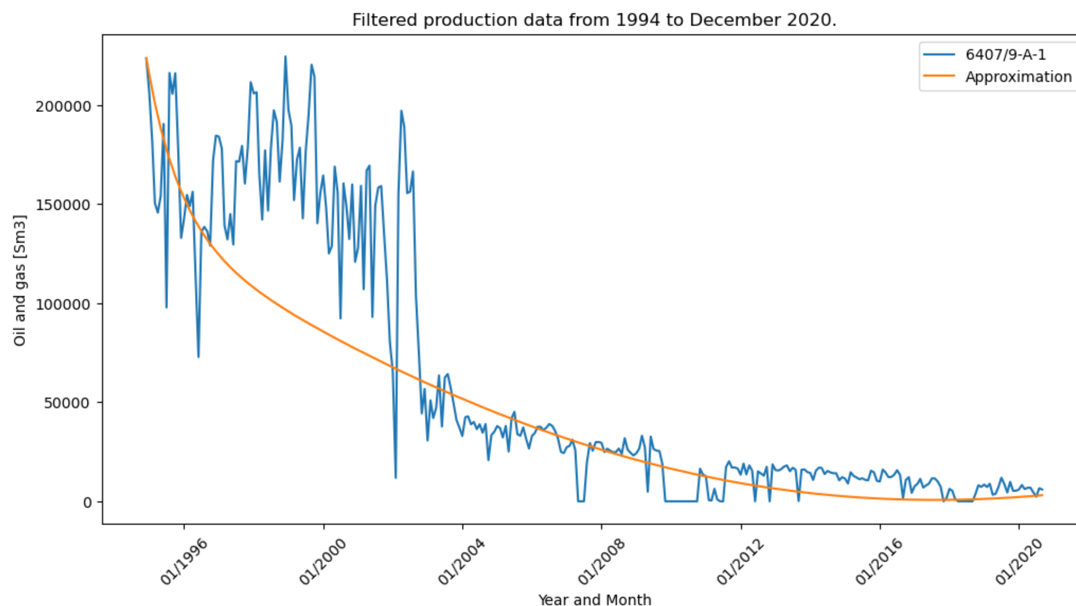
[26]:
```
array([[ 9.96159839e+01, -2.73130752e+02, -3.83454240e+02],
       [-3.22382027e-02,  3.70187759e-02,  0.00000000e+00],
       [ 5.87360444e-02, -3.82739125e-02,  1.90176817e-02],
       [-3.43451987e-02,  5.79877812e-03, -1.86472767e-02],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00]])
```

[27]:
```
mat_0 = mat[0]

u_approximation = compute_trajectory(mat_0, xi, POLYNOMIAL_ORDER, USE_TRIG,␣
 ↪USE_EXP)
x_aprox = u_approximation[:,0]
y_aprox = u_approximation[:,1]
z_aprox = u_approximation[:,2]
```

```
(311, 3)
```

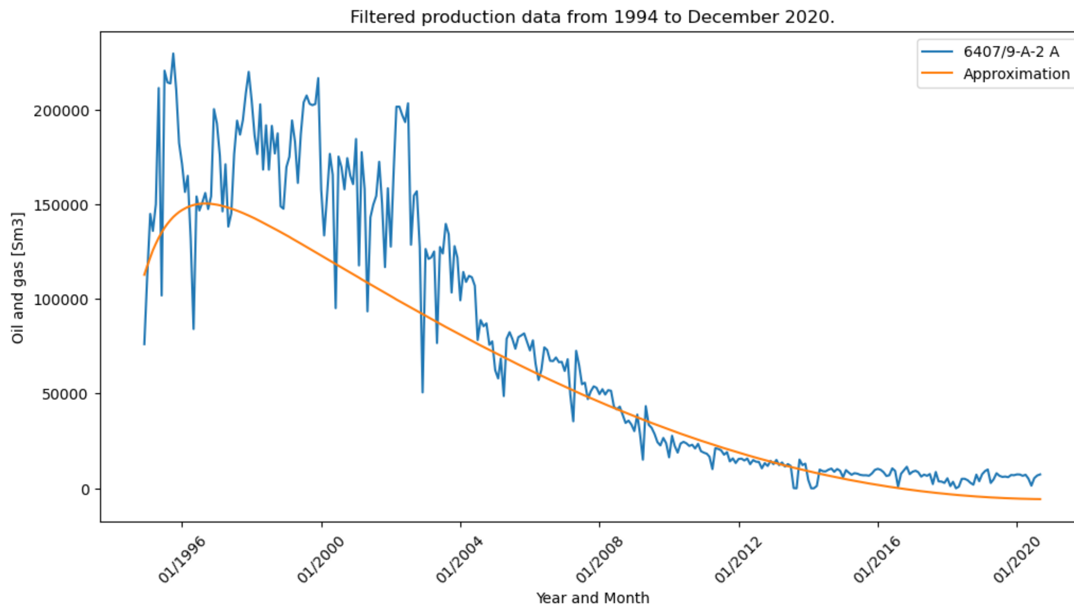
[28]:
```
fig, ax = plt.subplots(figsize=(12, 6))
ax.plot(dates, tot_prod_x, label='6407/9-A-1')
```

```
ax.plot(dates, x_aprox[:-1], label='Approximation')


date_fmt = mdates.DateFormatter('%m/%Y')
ax.xaxis.set_major_formatter(date_fmt)
plt.xticks(rotation=45)

# Set plot title and axis labels
plt.title('Filtered production data from 1994 to December 2020.')
plt.legend()
plt.xlabel('Year and Month')
plt.ylabel('Oil and gas [Sm3]')

plt.show()
```

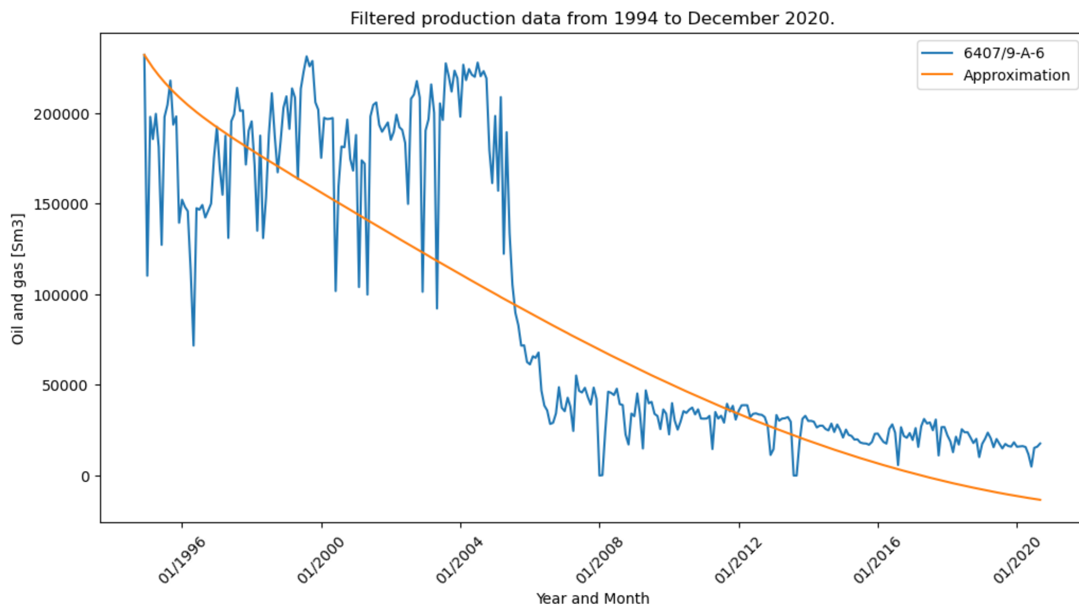


[29]:
```
fig, ax = plt.subplots(figsize=(12, 6))
ax.plot(dates, tot_prod_y, label='6407/9-A-2 A')
ax.plot(dates, y_aprox[:-1], label='Approximation')


date_fmt = mdates.DateFormatter('%m/%Y')
ax.xaxis.set_major_formatter(date_fmt)
plt.xticks(rotation=45)

# Set plot title and axis labels
plt.title('Filtered production data from 1994 to December 2020.')
```

```
plt.legend()
plt.xlabel('Year and Month')
plt.ylabel('Oil and gas [Sm3]')

plt.show()
```



[30]:
```
fig, ax = plt.subplots(figsize=(12, 6))
ax.plot(dates, tot_prod_z, label='6407/9-A-6')
ax.plot(dates, z_aprox[:-1], label='Approximation')


date_fmt = mdates.DateFormatter('%m/%Y')
ax.xaxis.set_major_formatter(date_fmt)
plt.xticks(rotation=45)

# Set plot title and axis labels
plt.title('Filtered production data from 1994 to December 2020.')
plt.legend()
plt.xlabel('Year and Month')
plt.ylabel('Oil and gas [Sm3]')

plt.show()
```


Filtered production data from 1994 to December 2020.
6407/9-A-6
Approximation
Oil and gas [Sm3]
200000
150000
100000
50000
0
01/1996
01/2000
01/2004
01/2008
01/2012
01/2016
01/2020
Year and Month

[ ]:

A.5 Code for PySINDy on Draugen

pysindy-1

June 14, 2023

1 PySINDy package

Using the PySINDy package on production data from Draugen.

[1]:
```
import requests
import io
import matplotlib.dates as mdates
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
```

[2]:
```
url = 'https://raw.githubusercontent.com/SanderSondeland/Master/main/
 ↪20230418_WellBore_monthlyFacility%20(1).csv?
 ↪token=GHSAT0AAAAAACD4GTTLLQ6GOKYVKBIVLCMWZEJQAOQ'
download = requests.get(url).content
data = pd.read_csv(io.StringIO(download.decode('utf-8')))
data
```

[2]:
```
                name  npdId    field  year  month  operationTime  \
0        6407/9-A-1   2254  DRAUGEN  1994      6            NaN
1        6407/9-A-1   2254  DRAUGEN  1994      7            NaN
2        6407/9-A-1   2254  DRAUGEN  1994      8            NaN
3        6407/9-A-1   2254  DRAUGEN  1994      9            NaN
4        6407/9-A-1   2254  DRAUGEN  1994     10            NaN
…                 …      …        …     …      …              …
3700  6407/9-G-5 H   7715  DRAUGEN  2020      8          31.00
3701  6407/9-G-5 H   7715  DRAUGEN  2020      9          28.93
3702  6407/9-G-5 H   7715  DRAUGEN  2020     10          25.54
3703  6407/9-G-5 H   7715  DRAUGEN  2020     11          30.00
3704  6407/9-G-5 H   7715  DRAUGEN  2020     12          31.00

     operationTimeUom wellStatus     oil oilUom        gas gasUom  condensate  \
0                   d        NaN  172347    Sm3    7931000    Sm3         NaN
1                   d        NaN  198022    Sm3   10221000    Sm3         NaN
2                   d        NaN  201457    Sm3   11584000    Sm3         NaN
3                   d        NaN  197711    Sm3   11087000    Sm3         NaN
4                   d        NaN  195050    Sm3   10988000    Sm3         NaN
…                   …          …       …      …          …      …           …
```

```
3700                d  producing   8851    Sm3    304029    Sm3          NaN
3701                d  producing   8493    Sm3    270559    Sm3          NaN
3702                d  producing   7268    Sm3    245919    Sm3          NaN
3703                d  producing   8109    Sm3    257430    Sm3          NaN
3704                d  producing   9069    Sm3    284342    Sm3          NaN

     condensateUom  water  waterUom
0              Sm3      0       Sm3
1              Sm3      0       Sm3
2              Sm3      0       Sm3
3              Sm3      0       Sm3
4              Sm3      0       Sm3
…              …      …       …
3700           Sm3  66253       Sm3
3701           Sm3  66175       Sm3
3702           Sm3  62792       Sm3
3703           Sm3  73987       Sm3
3704           Sm3  79158       Sm3

[3705 rows x 16 columns]
```

[3]:

```
data1 = data.drop(columns=['npdId', 'field', 'operationTime',␣
 ↪'operationTimeUom','wellStatus','oilUom', 'gasUom', 'condensate',
'condensateUom', 'water', 'waterUom'])
data1
```

[3]:

```
              name  year  month     oil       gas
0       6407/9-A-1  1994      6  172347   7931000
1       6407/9-A-1  1994      7  198022  10221000
2       6407/9-A-1  1994      8  201457  11584000
3       6407/9-A-1  1994      9  197711  11087000
4       6407/9-A-1  1994     10  195050  10988000
…              …    …      …      …        …
3700  6407/9-G-5 H  2020      8    8851    304029
3701  6407/9-G-5 H  2020      9    8493    270559
3702  6407/9-G-5 H  2020     10    7268    245919
3703  6407/9-G-5 H  2020     11    8109    257430
3704  6407/9-G-5 H  2020     12    9069    284342

[3705 rows x 5 columns]
```

[4]:

```
x_data = data1[data1['name'] == '6407/9-A-1']
x_data.loc[:, 'period'] = x_data['year'].astype(str) + '-' + x_data['month'].
 ↪astype(str)
x_data.loc[:, 'tot_prod'] = (x_data['oil'] + x_data['gas']/1000).round(1)
x_data = x_data.groupby('period').sum()
x_data
```

```
C:\Users\sande\AppData\Local\Temp\ipykernel_4760\3989910507.py:2:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  x_data.loc[:, 'period'] = x_data['year'].astype(str) + '-' +
x_data['month'].astype(str)
C:\Users\sande\AppData\Local\Temp\ipykernel_4760\3989910507.py:3:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  x_data.loc[:, 'tot_prod'] = (x_data['oil'] + x_data['gas']/1000).round(1)
```

[4]:

	year	month	oil	gas	tot_prod
period					
1994-10	1994	10	195050	10988000	206038.0
1994-11	1994	11	196044	10255000	206299.0
1994-12	1994	12	176032	9252000	185284.0
1994-6	1994	6	172347	7931000	180278.0
1994-7	1994	7	198022	10221000	208243.0
…	…	…	…	…	…
2020-5	2020	5	6446	420354	6866.4
2020-6	2020	6	4180	279594	4459.6
2020-7	2020	7	2172	143060	2315.1
2020-8	2020	8	6258	352751	6610.8
2020-9	2020	9	5639	295011	5934.0

[318 rows x 5 columns]

[5]:

```
y_data = data1[data1['name'] == '6407/9-A-2 A']
y_data.loc[:, 'period'] = y_data['year'].astype(str) + '-' + y_data['month'].
  ↪astype(str)
y_data.loc[:, 'tot_prod'] = (y_data['oil'] + y_data['gas']/1000).round(1)
y_data.groupby('period').sum()
y_data
```

```
C:\Users\sande\AppData\Local\Temp\ipykernel_4760\4226326607.py:2:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
```

```
  y_data.loc[:, 'period'] = y_data['year'].astype(str) + '-' +
y_data['month'].astype(str)
C:\Users\sande\AppData\Local\Temp\ipykernel_4760\4226326607.py:3:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  y_data.loc[:, 'tot_prod'] = (y_data['oil'] + y_data['gas']/1000).round(1)
```

[5]:

```
               name  year  month     oil      gas   period  tot_prod
318  6407/9-A-2 A  1994     12   72310  3802000  1994-12   76112.0
319  6407/9-A-2 A  1995      1  107169  5585000   1995-1  112754.0
320  6407/9-A-2 A  1995      2  137400  7487000   1995-2  144887.0
321  6407/9-A-2 A  1995      3  128699  7202000   1995-3  135901.0
322  6407/9-A-2 A  1995      4  142063  7908000   1995-4  149971.0
..            …     …      …       …        …        …
625  6407/9-A-2 A  2020      8    5055   287806   2020-8    5342.8
626  6407/9-A-2 A  2020      9    6422   336610   2020-9    6758.6
627  6407/9-A-2 A  2020     10    6954   381451  2020-10    7335.5
628  6407/9-A-2 A  2020     11    6040   315625  2020-11    6355.6
629  6407/9-A-2 A  2020     12    6962   358310  2020-12    7320.3

[312 rows x 7 columns]
```

[6]:

```
z_data = data1[data1['name'] == '6407/9-A-6']
z_data.loc[:, 'period'] = z_data['year'].astype(str) + '-' + z_data['month'].
 ↪astype(str)
z_data.loc[:, 'tot_prod'] = (z_data['oil'] + z_data['gas']/1000).round(1)
z_data.groupby('period').sum()
z_data
```

```
C:\Users\sande\AppData\Local\Temp\ipykernel_4760\2187305751.py:2:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  z_data.loc[:, 'period'] = z_data['year'].astype(str) + '-' +
z_data['month'].astype(str)
C:\Users\sande\AppData\Local\Temp\ipykernel_4760\2187305751.py:3:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
```

```
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  z_data.loc[:, 'tot_prod'] = (z_data['oil'] + z_data['gas']/1000).round(1)
```

[6]:

```
             name  year  month     oil       gas   period  tot_prod
1897  6407/9-A-6  1994      8  206901  11608000   1994-8  218509.0
1898  6407/9-A-6  1994      9  215894  12106000   1994-9  228000.0
1899  6407/9-A-6  1994     10  221972  12504000  1994-10  234476.0
1900  6407/9-A-6  1994     11  211153  11045000  1994-11  222198.0
1901  6407/9-A-6  1994     12  220564  11594000  1994-12  232158.0
…           …   …     …      …       …        …         …
2208  6407/9-A-6  2020      8   14463    815989   2020-8   15279.0
2209  6407/9-A-6  2020      9   15132    791996   2020-9   15924.0
2210  6407/9-A-6  2020     10   15493   2185778  2020-10   17678.8
2211  6407/9-A-6  2020     11   13065   2184933  2020-11   15249.9
2212  6407/9-A-6  2020     12   14127   2428084  2020-12   16555.1

[316 rows x 7 columns]
```

[7]:

```
import datetime
dates = [datetime.datetime(year=int(year), month=int(month), day=1) for year in␣
 ↪range(1994, 2021) for month in range(1, 13)]
dates = dates[11:-3]
```

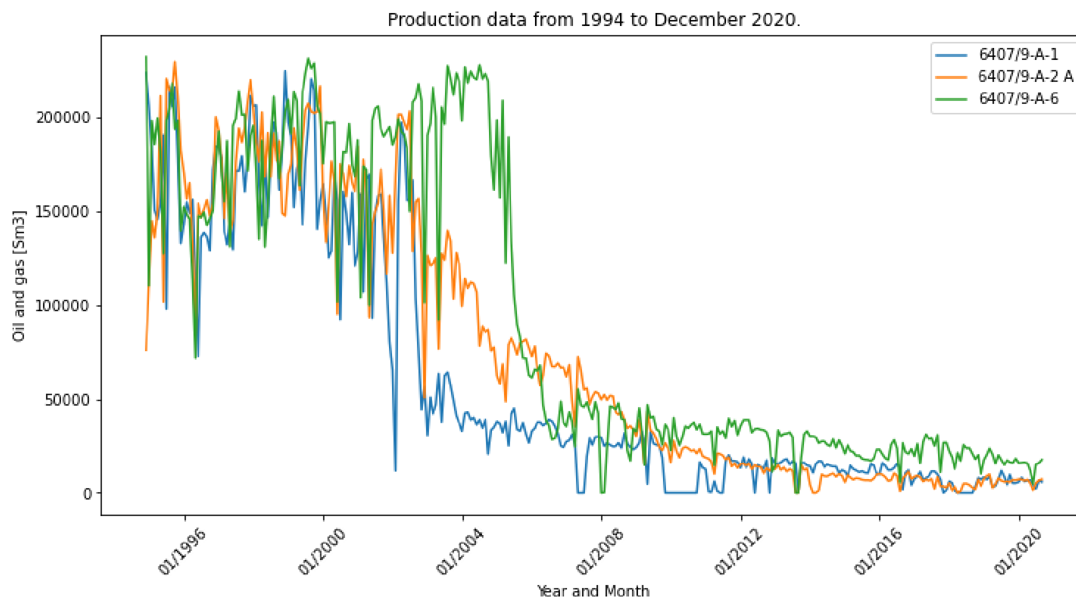
[8]:

```
test_x = x_data[8:]
tot_prod_x = test_x['tot_prod']

test_y = y_data[:-2]
tot_prod_y = test_y['tot_prod']

test_z = z_data[4:-2]
tot_prod_z = test_z['tot_prod']
```

[9]:

```
fig, ax = plt.subplots(figsize=(12, 6))
ax.plot(dates, tot_prod_x, label='6407/9-A-1')
ax.plot(dates, tot_prod_y, label='6407/9-A-2 A')
ax.plot(dates, tot_prod_z, label='6407/9-A-6')

date_fmt = mdates.DateFormatter('%m/%Y')
ax.xaxis.set_major_formatter(date_fmt)
plt.xticks(rotation=45)

# Set plot title and axis labels
plt.title('Production data from 1994 to December 2020.')
plt.legend()
plt.xlabel('Year and Month')
plt.ylabel('Oil and gas [Sm3]')
```

```
plt.show()
```



2 Filtering

[10]:
```
from scipy.ndimage import median_filter
import numpy as np
```

[11]:
```
md_x = median_filter(tot_prod_x, size=4)

md_y = median_filter(tot_prod_y, size=4)

md_z = median_filter(tot_prod_z, size=4)
```

[12]:
```
i = len(dates)
t = list(range(1, i+1))
t_np = np.array(t)
```

[13]:
```
md_x_np = np.array(md_x)
md_y_np = np.array(md_y)
md_z_np = np.array(md_z)

tot_prod_x_np = np.array(tot_prod_x)
tot_prod_y_np = np.array(tot_prod_y)
tot_prod_z_np = np.array(tot_prod_z)
```

3 PySINDy

[14]:
```
import logging
import pysindy as ps
from scipy.integrate import odeint
from pysindy.differentiation import FiniteDifference, SINDyDerivative,␣
↪SmoothedFiniteDifference
from pysindy.optimizers import STLSQ
```

[15]:
```
mat = np.zeros((310,3))

mat[:, 0] = md_x
mat[:, 1] = md_y
mat[:, 2] = md_z

i = len(dates)
t = list(range(1, i+1))
t_np = np.array(t)
```

[16]:
```
Threshold = 0.005
Max_iterations = 10
```

[17]:
```
def fit(u: np.ndarray, t: np.ndarray) -> ps.SINDy:
    """Uses PySINDy to find the equation that best fits the data u.
    """
    optimizer = STLSQ(threshold=Threshold, max_iter=Max_iterations)

    # Finite difference derivatives.
    differentiation_method = FiniteDifference()
    differentiation_method2 = SmoothedFiniteDifference()

    model = ps.SINDy(optimizer=optimizer,
                     differentiation_method=differentiation_method2,
                     feature_names=["x", "y", "z"],
                     discrete_time=False)
    model.fit(u, t=t)
    model.print()

    return model
```

[18]:
```
def compute_trajectory(u0: np.ndarray, model: ps.SINDy) -> np.ndarray:
    """Calculates the trajectory using the model discovered by SINDy.
    """
    t0 = 0
    dt = 1
    tmax = 310
    n = int(tmax / dt + 1)
```

```
    t_eval = np.linspace(start=t0, stop=tmax, num=n)

    u_approximation = model.simulate(u0, t_eval)

    return u_approximation
```

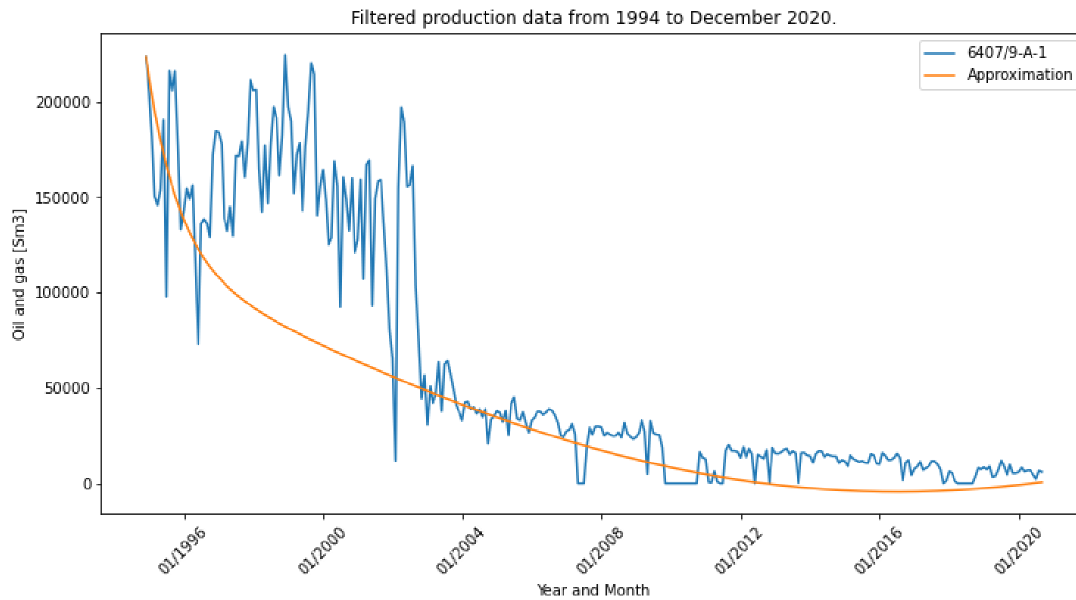
[19]:
```
model = fit(mat, t_np)
```

```
(x)' = 115.442 1 + -0.045 x + 0.076 y + -0.041 z
(y)' = -255.021 1 + 0.028 x + -0.024 y
(z)' = -396.834 1 + -0.015 x + 0.039 y + -0.025 z
```

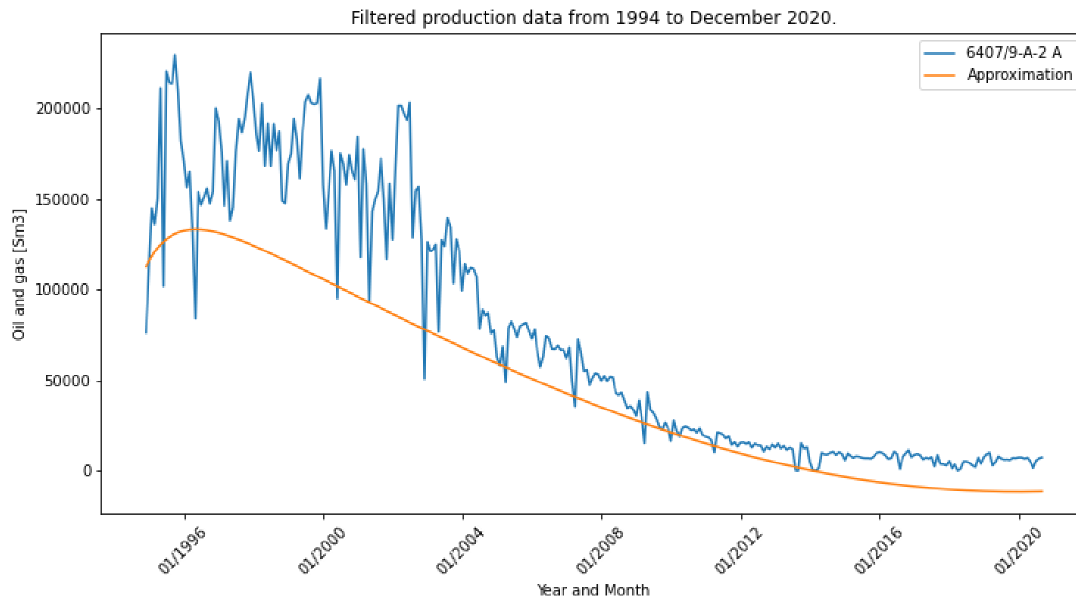
[20]:
```
u0 = mat[0]
u_approximation = compute_trajectory(u0, model)
```

[21]:
```
x_aprox = u_approximation[:,0]
y_aprox = u_approximation[:,1]
z_aprox = u_approximation[:,2]
```

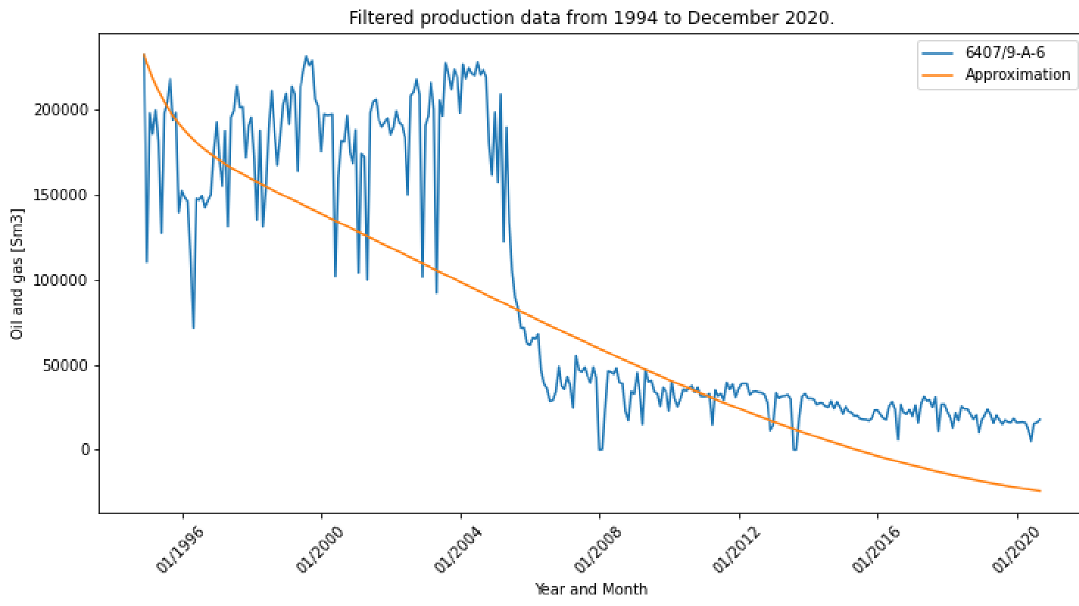
[22]:
```
fig, ax = plt.subplots(figsize=(12, 6))
ax.plot(dates, tot_prod_x, label='6407/9-A-1')
ax.plot(dates, x_aprox[:-1], label='Approximation')


date_fmt = mdates.DateFormatter('%m/%Y')
ax.xaxis.set_major_formatter(date_fmt)
plt.xticks(rotation=45)

# Set plot title and axis labels
plt.title('Filtered production data from 1994 to December 2020.')
plt.legend()
plt.xlabel('Year and Month')
plt.ylabel('Oil and gas [Sm3]')

plt.show()
```



[23]:
```
fig, ax = plt.subplots(figsize=(12, 6))
ax.plot(dates, tot_prod_y, label='6407/9-A-2 A')
ax.plot(dates, y_aprox[:-1], label='Approximation')


date_fmt = mdates.DateFormatter('%m/%Y')
ax.xaxis.set_major_formatter(date_fmt)
plt.xticks(rotation=45)

# Set plot title and axis labels
plt.title('Filtered production data from 1994 to December 2020.')
plt.legend()
plt.xlabel('Year and Month')
plt.ylabel('Oil and gas [Sm3]')

plt.show()
```



[24]:
```
fig, ax = plt.subplots(figsize=(12, 6))
ax.plot(dates, tot_prod_z, label='6407/9-A-6')
ax.plot(dates, z_aprox[:-1], label='Approximation')


date_fmt = mdates.DateFormatter('%m/%Y')
ax.xaxis.set_major_formatter(date_fmt)
plt.xticks(rotation=45)

# Set plot title and axis labels
plt.title('Filtered production data from 1994 to December 2020.')
plt.legend()
plt.xlabel('Year and Month')
plt.ylabel('Oil and gas [Sm3]')

plt.show()
```


Filtered production data from 1994 to December 2020.
6407/9-A-6
Approximation
Oil and gas [Sm3]
200000
150000
100000
50000
0
01/1996
01/2000
01/2004
01/2008
01/2012
01/2016
01/2020
Year and Month

```
[ ]:
```

A.6 Master poster

Data Driven Model Discovery – Petroleum application

Sander André Søndeland
University of Stavanger

Abstract

The Sparse Identification of Nonlinear Dynamics (SINDy) model has proven its efficacy in extracting governing equations from dynamical systems. Through sparsity-promoting techniques, SINDy offers compact and interpretable representations of dynamics. This research study demonstrates the successful application of SINDy on production data from oil wells due to good results.

Introduction

Data Driven model discovery is a field that is rapidly developing with a range of different techniques. One of these are the SINDy algorithm, also known as sparse identification of non-linear dynamical systems.

SINDy method was first introduced in 2016 by S. L. Brunton, J. L. Proctor and J. N. Kutz. This was seen as a powerful technique to identify nonlinear dynamical systems from data.

There has also been developed an open-source python package, pySINDy, to utilize the SINDy method easier. It's open source and has been developed over the last years. This is a good tool both for beginners and advanced users as well.

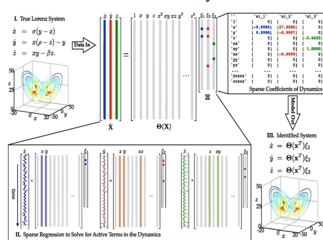
Methodology

The SINDy method uses a set of data as input and differentiate the data. Then a custom library with candidate terms will be generated. These libraries can contain d'th–degree polynomials, exponential terms, sine and cosine. It's only limited by one's imagination.

Using the data and the library of candidate functions, the SINDy algorithm searches for the sparsest set of candidate functions that can represent the dynamics observed in the data.

The result of the SINDy algorithm is a set of governing equations that best capture the system's dynamics.

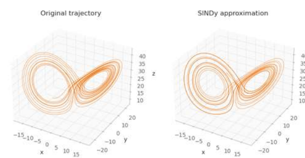
Overview of the SINDy method



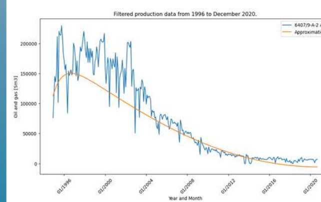
Results

The results are best shown by comparing graphs. There will be results from two different tests. One is a more theoretical result where the data is generated and are the result from the Lorenz system. The two other is based on data from Diskos from a well on the Draugen oil field and the Statfjord Øst field. These system has more noisy data compared to the Lorenz system.

Lorenz system

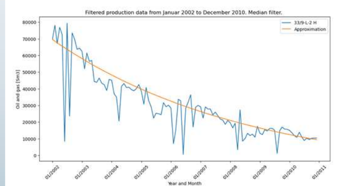


Draugen



More results

Statfjord Øst



Conclusion

The results was promising and good. There can be done a lot of testing with different parameters and changing the library with candidate terms. It shows that even if the data from Diskos was a bit spiky the SINDy method still managed to find some equations to capture the system's dynamics. Even though the results was promising the method is sensible to noise, and a bit of filtering before and during the differentiation was necessary to get good results. Good knowledge about the system is also a benefit when creating the library with candidate functions.

Acknowledgements

I would like to extend a big thank you to my supervisor Aksel Hiorth, who has been a great help during the process of my master's thesis.