École Polytechnique

BACHELOR THESIS IN COMPUTER SCIENCE

Reparametrizing ODE models by scaling

Author:

Milos Oundjian, École Polytechnique

Advisor:

Gleb Pogudin, LIX

Academic year 2023/2024

Abstract

In this thesis, we present a novel implementation of the Hubert and Labahn algorithm for scaling invariants and symmetry reduction in dynamical systems, utilizing the Julia programming language. Our work extends the algorithm's application to non-identifiable models, offering a tool for the simplification and analysis of complex ordinary differential equation (ODE) models prevalent in the natural sciences. By integrating this implementation with the "StructuralIdentifiability.jl" package, we aim to make it easier for scientist to access this algorithm, enabling researchers without extensive computational backgrounds to leverage model simplification and reparameterization techniques. This approach not only facilitates the numerical solution of intricate models but also contributes to a deeper understanding of the underlying principles governing natural phenomena.

Contents

1   Background

1.1   Julia Programming Language

The Julia programming language, introduced in 2012, is a high-level, high-performance language designed for technical and scientific computing. It is designed to combine the usability of python with the speed and efficiency of C [1]. It is well suited for symbolic computation such as operations with polynomials and matrices thanks to sophisticated mathematical syntax, dynamic typing, and extensive support for numerical operations. Furthermore, libraries such as Nemo.jl make it easy to have access to ecosystems of packages that can all work together to make going from theoretical math to algorithms in code much simpler.

1.2   Nemo

The Nemo package for Julia is designed to support work in commutative algebra and number theory, providing tools for arithmetic operations, polynomial manipulations, and matrix computations [4]. It leverages the Flint library to offer efficient arithmetic capabilities. Nemo is valuable for researchers and mathematicians needing high-performance computational tools. Its integration with Julia enhances its appeal by combining Flint's arithmetic efficiency with Julia's high-level syntax and performance, making it a practical choice for computational mathematics projects, including the StructuralIdentifiability package.

1.3   StructuralIdentifiability

StructuralIdentifiability.jl is a package developed for the Julia programming language, designed to evaluate whether parameters within parametric Ordinary Differential Equation (ODE) models can be uniquely determined, covering both local and global perspectives. It facilitates the calculation of functions that can identify states and parameters. Additionally, the package provides tools for analyzing local identifiability in models based on discrete time. It serves as an introductory resource for understanding structural identifiability.

2   Introduction

In their paper "Scaling Invariants and Symmetry Reductions of Dynamical Systems" [5], they discuss the theory behind the algorithm and how it can be implemented. The algorithm is based on an extension of the Buckingham-π-theorem to enable a systematic algorithm to find the invariants that the Buckingham-π-theorem describes but does not tell us how to obtain.

The algorithm is described in the paper but no implementation of it exists. Thus the goal of this paper is to create an implementation of this algorithm in Julia.

Some obstacles to the implementation of this algorithm were that while libraries existed for getting Hermite Normal Forms, they only existed for *row* Hermite Normal Form and not Column Hermite Normal forms, so it was necessary to create a function to do this and to prove its accuracy. This is detailed in section 3.1.3.

An additional goal of this paper is to possibly add the implemented algorithm into the Julia library StructuralIdentifiability [3] to make it accessible to scientists in the fields of Biology and Chemistry who are working with Ordinary Differential Equations that can make use of reparameterization to make them easier to solve.

3   Background

3.1   Hermite Normal Forms and Hermite Multipliers

The algorithm described in [5] extensively uses Hermite Normal Forms and what it calls "Hermite Multipliers". So this section will go over the definitions that are used in that paper and here in order to avoid potential errors over notation since different notations for Hermite Normal Forms exist from different authors.

3.1.1   Hermite Normal Forms (HNFs)

While different sources may have varying definitions of Hermite Normal Forms, the ones we will follow are from [5] as they are the ones used in the definition of the algorithm.

Definition 1 (*Column* Hermite Normal Form)   *(i) The first r columns are non-zero;*

  (ii) $h_{k,j} = 0$ for $k > i_j$;

  (iii) $0 \leq h_{i_j,k} < h_{i_j,j}$ when $j < k$.

Definition 2 (*Row* Hermite Normal Form)   *(i) The first r rows are non-zero;*

  (ii) $h_{i,k} = 0$ for $k < j_i$;

  (iii) $0 \leq h_{k,j_i} < h_{i,j_i}$ when $i < k$.

  Example. Take for example the matrix

$$\begin{pmatrix} 6 & 0 & -4 & 1 & 3 \\ 4 & 3 & 1 & -4 & 3 \end{pmatrix}.$$

The *row* Hermite Normal Form of this matrix is

$$\begin{pmatrix} 2 & 6 & 6 & -9 & 3 \\ 0 & 9 & 11 & -14 & 3 \end{pmatrix}.$$

The *column* Hermite Normal Form of this matrix is

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}.$$

While they may seem similar, it is important to be careful since both of these forms are used in the algorithm, thus in this paper we will try to always be specific about which Hermite Normal Form we are using.

3.1.2   Hermite Multipliers

Any integer matrix A can be transformed via integer row operations (respectively column) to obtain a unique row (respectively column) Hermite Normal Form matrix H. We can encode these row operations (respectively column) into a unimodular integer matrix V such that we have $V \cdot A = H$ (respectively $A \cdot V = H$). This matrix V is sometimes called the Hermite transform. In this paper we will be following [5] and calling in the *Hermite Multiplier* of A.

Example. Again let us take the matrix

$$A = \begin{pmatrix} 6 & 0 & -4 & 1 & 3 \\ 4 & 3 & 1 & -4 & 3 \end{pmatrix}.$$

Its *column* Hermite Normal Form is:

$$H = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}.$$

And a Hermite Multiplier is

$$V = \begin{pmatrix} 16 & 4 & 6 & 33 & -60 \\ -28 & -7 & -11 & -58 & 105 \\ 24 & 6 & 9 & 50 & -90 \\ 1 & 0 & 0 & 2 & -3 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

We can check that $A \cdot V = H$.

3.1.3   Algorithmically determining HNF and Hermite Multipliers

There exists algorithms that can computer Hermite Normal Forms which can be found for example in [2]. The steps and complexity of this algorithm is discussed there as well. However, for the scope of this paper we will not discuss these algorithms in detail.

Effectively, we will use these algorithms as a part of our algorithm. To do this we give this algorithm a specification. The function we care about is in the Julia library `Nemo.jl` [4] and is called `hnf_with_transform`. The specification of this function is as follows:

This means that in our algorithm, we already have a tool that allows us to calculate row Hermite Normal Form. What we need to do is to find a way to use this tool, for example by doing matrix operations on the input matrix A and on the result from using the function `hnf_with_transform` to be able to create a new function that taking input matrix A, returns the column hermite normal form of A. In the next section we will show how we are able to do this and prove our new method.

Algorithm 1: hnf_with_transform_column

Input Integer matrix A

Output *column* HNF matrix H and Hermite Multiplier V

(Step 1) Set m the number of rows of matrix A and r its rank.

(Step 2) Reverse the rows of A then transpose A.

(Step 3) Set H and V to be the *row* HNF and corresponding multiplier of A.

(Step 4) Transpose H, then reverse the rows of H, then reverse the first r columns of H.

(Step 5) Transpose V, then reverse the rows of V.

(Step 6) Return H our *column* Hermite Normal Form of the input A and V the corresponding Hermite Multiplier.

The `hnf_with_transform` function that the Nemo library provides us with works to get the *row* Hermite Normal Form of our input matrix A by doing row operations. This is due to the property of the Hermite Normal Form being found through a series of row operations on the input matrix A. Similarly, the *column* Hermite Normal can be found through series of column operations, so transposing A applying the `hnf_with_transform` and then transposing the result will allow us to effectively be using column operations (Row operations on the transpose are like column operations on the original matrix).

There are a few more details to this algorithm that are elaborated on in the next section.

3.1.4   Explanation and Proof of Algorithm

We start with a matrix $A \in \mathbb{Z}^{m \times n}$ of rank r. We start by reversing the rows of A, which is equivalent to multiplying A on the left my the $m \times m$ matrix. The reason we do this is that later on in the algorithm we will have to reverse the rows again in order to get the proper *column* Hermite Normal Form layout for the result matrix, so we have to flip the rows once in advance to cancel out the effect.

$$C_m = \begin{pmatrix} 0 & \cdots & 1 \\ \vdots & \cdot^{\cdot^{\cdot}} & \vdots \\ 1 & \cdots & 0 \end{pmatrix}$$

So we get the result of $C_m A$. We then take the transpose of this to get:

$$(C_m A)^\mathsf{T} = A^\mathsf{T} C_m$$

We then get the Hermite Normal Form and Hermite Multiplier of this, this is done through a series of row operations on the $A^\mathsf{T} C_m$. We use the Nemo function `hnf_with_transform(A)` to do this which, gives us H the *row* Hermite Normal Form of A and V a Hermite Multiplier. Thus applying this to $A^\mathsf{T} C_m$, we get H and V such that:

$$V \cdot (A^\mathsf{T} C_m) = H$$

Furthermore, since H is in *row* Hermite Normal Form, we have that it follows the definition, and thus as a matrix looks something like this:

$$H = \begin{pmatrix} * & \cdot & | & | & \cdot \\ 0 & 0 & * & | & \cdot \\ 0 & 0 & 0 & * & \cdot \\ 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \end{pmatrix}$$

Where non first r columns are 0, and the $|$ symbols show values that are $0 \leq | \leq *$ To follow conditions (ii) and (iii).

Now we need to change H so that it is in *column* Hermite Normal Form according to our definition and so that $A \cdot V = H$.

To do this our next step is to take the transpose of H. Taking this our new H^T looks like this.

$$H^\mathsf{T} = \begin{pmatrix} * & 0 & 0 & 0 & 0 \\ \cdot & 0 & 0 & 0 & 0 \\ - & * & 0 & 0 & 0 \\ - & - & * & \vdots & \vdots \\ \cdot & \cdot & \cdot & 0 & 0 \end{pmatrix}$$

We then bring flip the rows again giving us:

$$C_m H^\mathsf{T} = \begin{pmatrix} \cdot & \cdot & \cdot & 0 & 0 \\ - & - & * & 0 & 0 \\ - & * & 0 & 0 & 0 \\ \cdot & 0 & 0 & \vdots & \vdots \\ * & 0 & 0 & 0 & 0 \end{pmatrix}$$

The final step is to flip the first r columns of the resulting $C_m H^\mathsf{T}$ to get:

$$C_m H^\mathsf{T} C_r = \begin{pmatrix} \cdot & \cdot & \cdot & 0 & 0 \\ * & - & - & 0 & \vdots \\ 0 & * & - & 0 & 0 \\ 0 & 0 & \cdot & \vdots & \vdots \\ 0 & 0 & * & 0 & 0 \end{pmatrix}$$

This final result is in *column* Hermite Normal Form according to our definition, so thus we have our desired result.

From the linear algebra perspective, what we did holds up. From what we had earlier:

$$V \cdot (A^\mathsf{T} C_m) = H$$
$$(V \cdot (A^\mathsf{T}) C_m)^\mathsf{T} = H^\mathsf{T}$$
$$C_m A \cdot V^\mathsf{T} = H^\mathsf{T}$$
$$A \cdot V^\mathsf{T} C_r = C_m H^\mathsf{T} C_r$$

We note that in our algorithm applied transpose and multiplication on the right by C_r so our resulting V is $V^\mathsf{T} C_r$ from the original V we had from our `hnf_with_transform(A)` function.

It is very clear to see how it uses the steps of our algorithm to get the *column* Hermite Normal Form.

3.1.5   Normal Hermite Multiplier

As explained earlier, the Hermite Normal Form of an integer matrix is unique, however the Hermite Multiplier itself is not. In section 2.2 of [5], they describe a way to calculate the *row* Hermite Normal Form Normal Hermite Multiplier through making a change to the input matrix A. Proposition 2.3 in the paper defines the Normal Hermite Multiplier and provides the algorithm.

This algorithm was implemented in Julia for this project. The implementation is as follows.

Algorithm 2: hnf_with_normal_transform_column

Input Integer matrix A

Output *column* HNF matrix H and *Normal* Hermite Multiplier V

(Step 1) Set H to be the *column* Hermite Normal Form of A.

(Step 2) Set r and n to be the number of rows and columns of A respectively.

(Step 3) Set B to be A with an $n \times n$ identity matrix on top, aka $B = \begin{bmatrix} I_n \\ A \end{bmatrix}$.

(Step 4) Set H^* and *row* Hermite Normal Form of B.

(Step 5) Set V_i to be the last r columns of H^* with the bottom r rows removed.

(Step 6) Set V_n to be the first $n - r$ columns of with the bottom r rows removed.

(Step 7) Set $V = \begin{bmatrix} V_i & V_n \end{bmatrix}$.

(Step 8) Return H and V.

---

4 Algorithm

Now that we have a way to calculate the *column* Hermite Normal Forms and *column* Normal Hermite Transforms, we finally have all the tools to implement the algorithm to simplify the ODE equations. The proofs of the algorithm are explained in [5] so this paper is not going to go over them. However, we will be going over each part of the algorithm with examples to demonstrate how it comes together in order to reparameterize ODEs.

The clearest way to illustrate this algorithm is with a diagram. The diagram below illustrates the steps of the algorithm that will be elaborated in the next section with examples.

```
┌─────────────────────────────────────────────────────────┐
│                       Input: ODE                        │
└─────────────────────────────────────────────────────────┘
                              │
                              ▼
┌─────────────────────────────────────────────────────────┐
│                      ODE Matrix K                       │
└─────────────────────────────────────────────────────────┘
                              │
                              ▼
┌─────────────────────────────────────────────────────────┐
│        H is row HNF of K and U is the corresponding     │
│                    Hermite Transform                    │
└─────────────────────────────────────────────────────────┘
                              │
                              ▼
┌─────────────────────────────────────────────────────────┐
│    r is the number of zero rows in H, A is the last r rows │
│                         of U                            │
└─────────────────────────────────────────────────────────┘
                              │
                              ▼
┌─────────────────────────────────────────────────────────┐
│   V is the normal column HNF transform of A and W is    │
│                      inverse of V                       │
└─────────────────────────────────────────────────────────┘
                              │
                              ▼
┌─────────────────────────────────────────────────────────┐
│   Vn is the last n-r columns of V and Wd is the n-r last │
│                        rows of W                        │
└─────────────────────────────────────────────────────────┘
                              │
                              ▼
┌─────────────────────────────────────────────────────────┐
│        (y1, y2, ... y{n-r}) are the new parameters      │
└─────────────────────────────────────────────────────────┘
                              │
                              ▼
┌─────────────────────────────────────────────────────────┐
│            Proposition 6.2 From Hubert and Labahn       │
└─────────────────────────────────────────────────────────┘
                              │
                              ▼
┌─────────────────────────────────────────────────────────┐
│   Output: Reparameterized ODE, Dict from new to old     │
│                       variables                         │
└─────────────────────────────────────────────────────────┘
```

5   Implementation

The following section explains how this algorithm was implemented in practice with an example.

5.1   Transformation of ODEs into Matrix Form

The initial phase in the reparameterization of ordinary differential equations (ODEs) pivots on the foundational step of converting our system of ODEs into a structured format amenable to manipulation and analysis. This process begins by reformatting the system into a matrix representation, which facilitates the application of algebraic and computational techniques for further examination and transformation.

   In the context of our implementation, we leverage the 'StructuralIdentifiability' [3] package in Julia, a tool for symbolic computation, made for for analyzing the identifiability of parameters within systems described by ODEs. This package offers a structured approach to defining ODEs, incorporating essential elements such as variables `x_vars`, parameters, and the independent variable `t`. This structured definition not only standardizes the representation of differential equations but also simplifies the extraction and manipulation of their components.

   The transformation process is as follows:

1. **Definition of the ODE System:** Utilizing the `@ODEmodel` macro provided by the 'StructuralIdentifiability' package, we can easily define in Julia a system of ODEs. We can access the equations of this ODE through the `x_equations` field and the parameters through the `x_vars` and `parameters`. In many real life experimental scenarios, the user may also want to designate certain parameters as measured quantities, which in the ODE are designated as output values in the field `y_vars`.

2. **Extraction and Parsing of Equations:** With the system defined, the next step involves extracting each equation and parsing them to isolate the numerator and denominator. This step is crucial for handling fractional equations, which are common in biological and physical systems modeling. The parsing process ensures that each equation is represented in a fractional form (including non-fractional ones by treating the denominator as just 1), facilitating uniform treatment in later stages. We then use a custom function `vector_to_matrix` in order to get the result in the Nemo integer matrices that are easier to work with than the lists of vectors that we get from the equation.

3. **Normalizing** Following the algorithm [5], we now have to normalize each equations. This means that we assure that at least one term in the denominator is a constant. This is equivalent to one of the rows in the denominator matrix being zero. So the way we implement this in our algorithm is to take whatever the first row in our denominator is and to subtract it from all the other rows in the numerator and the denominator. This is analogous to dividing by that term in both the numerator and the denominator, which does not change the value of the polynomial.

4. **Padding of Matrix** Due to the independent variable `t` not being extracted, we need to account for it by adding a new column at the end of our resulting matrix. This can be seen in the algorithm in [5] as the matrices K always have a row for the independent variable (usually `t`).

5. **Concatenation** We combine the resulting matrices into one matrix that is our resulting matrix. We then transpose the result to get a matrix following the convention followed in [5].

   We can illustrate this process with this example.
   Let us start with the Predator Prey Model:

$$
\frac{\mathrm{d}n}{\mathrm{d}t} = n(r(1 - \frac{n}{K} - k\frac{p}{n+d})),
$$
$$
\frac{\mathrm{d}p}{\mathrm{d}t} = sp(1 - h\frac{p}{n})
$$

In our 'StructuralIdentifiability' @ODEmodel struct, we will be interested in three fields. x_vars, parameters and x_equations. In this example x_vars is the vector containing n(t) and p(t), parameters is the vector containing K, d, h, k, r, s, x_equations is a dictionary mapping x_vars to the corresponding equation for it in the ODE.

With that in mind we start by extracting a list of the equations. In this example giving us a list like this:

```
2−element  Vector{AbstractAlgebra.Generic.FracFieldElem{QQMPolyRingElem}}:
(−n(t)^3*r + n(t)^2*K*r − n(t)^2*d*r −
n(t)*p(t)*K*k*r + n(t)*K*d*r)//(n(t)*K + K*d)
(n(t)*p(t)*s − p(t)^2*h*s)//n(t)
```

The comes the 'parsing' process, where we turn the equations into tuples of numerator and denominator matrices. We handle non fractional equations by turning them into fractional equations by setting the denominator to 1. We then concert the result into Nemo ZZMatrices that will be easier to work with. At this stage we are left with a series of matrices. In our example we get

```
2−element  Vector{Tuple{ZZMatrix, ZZMatrix}}:
([0 0 0 0 1 0 3 0; 1 0 0 0 1 0 2 0; 0 1 0 0 1 0 2 0; 1 0 0 1 1 0 1 1; 1 1 0 0 1 0
([0 0 0 0 0 1 1 1; 0 0 1 0 0 1 0 2], [0 0 0 0 0 0 1 0])
```

The order of the variables here is the parameters field elements followed by x_vars field elements.

We then have to take into account the $\frac{\mathrm{d}(x)}{\mathrm{d}t}$ for all the x_vars and add the independent variable t to our resulting matrix so we have a function that takes the 'parsed' equations and returns the new resulting equations. It also takes care of normalizing the denominator to follow the specifications of the algorithm in [5].

We normalize the denominator by subtracting the first row of the denominator to all the rows of the numerator and denominator, thus ensuring that the numerator has at least a single non-variable term and thus being normalized.

Finally, with the output being a few separate matrices, the last step is to concatenate them together to achieve our desired ODE matrix K.

Following this from the predator prey model we get:

$$K = \begin{pmatrix} -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & -1 & -1 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \end{pmatrix}.$$

Which is the matrix we need to move on to the next part of the algorithm.

5.2   Getting Scaling Symmetries

Getting the scaling symmetries is the next part of the algorithm. This step is simple to implement in Julia since we have already implemented the function for getting *row* and *column* Hermite Normal Forms and their transforms.

First we take the *row* Hermite Normal Form and Hermite Transform of K. In the example of the Predator-Prey model we get using the code:

```
H, U = hnf_with_transform(K)
```

That:

$$H = \begin{pmatrix} 1 & 0 & 0 & 0 & -1 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

And that:

$$U = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & -1 & 0 & -1 & 0 & -1 & -1 & -1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & -1 \end{pmatrix}.$$

We see here that H has 3 zero rows so this means that $r = 3$. Thus we set A to be the last 3 rows of U and we get that:

$$A = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & -1 \end{pmatrix}.$$

Here A is our scaling symmetry matrix.

5.3   Getting our Normal Hermite Transforms

This step is simple. We set V to be the Hermite Normal Transform of A and W to be the inverse of V. The code is simply as such:

```
_, V = hnf_with_normal_transform_column(A)
```

We get that:

$$V = \begin{pmatrix} 1 & 0 & 0 & -1 & 0 & 0 & -1 & -1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

And:

$$W = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

We then care about V_n and $W_\mathfrak{d}$ which are the last $n - r$ columns and rows respectively of V and W. So in this case:

$$V_n = \begin{pmatrix} -1 & 0 & 0 & -1 & -1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

And:

$$W_n = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

5.4   Putting Everything Together

The final part of the algorithm is to use section 6.2 of [5]. What we do here in Julia is to create a new destination PolynomialRing for the new reparametrized ODE. We focus on equation (16) of the paper which is:

$$\frac{dy}{dt} = y \star F(y^{W_\mathfrak{d}}) \cdot V_n.$$

In order to use this equation we need to get the $\mathbb{R}^n \to \mathbb{R}^n$ function F. To do this we use the definition that

$\frac{dz}{dt} = z \star F(z)$, where z is a vector of all the variables in the ODE to see find that with:

$$\frac{dz}{dt} = \begin{pmatrix} \frac{dK}{dt} \\ \frac{dd}{dt} \\ \frac{dh}{dt} \\ \frac{dk}{dt} \\ \frac{dr}{dt} \\ \frac{ds}{dt} \\ \frac{dn}{dt} \\ \frac{dp}{dt} \\ \frac{dt}{dt} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ n(r(1 - \frac{n}{K} - k\frac{p}{n+d})) \\ sp(1 - h\frac{p}{n}) \\ 1 \end{pmatrix}.$$

So

$$F(z) = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ r(1 - \frac{n}{K} - k\frac{p}{n+d}) \\ s(1 - h\frac{p}{n}) \\ \frac{1}{t} \end{pmatrix}$$

With us knowing what F is, all that is left to do is to put all of this into Julia code and we are left with a function that takes in the ode, V_n and W_0 and that outputs the new parameterizations of the ode.

```
prop_6_2(ode, V_n, W_d)
```

Running this code gives us the new reparameterization of the ODE which is:

$$\frac{dy_1}{dt} = 0$$

$$\frac{dy_2}{dt} = 0$$

$$\frac{dy_3}{dt} = 0$$

$$\frac{dy_4}{dt} = \frac{-y_1 y_4^2 + y_1 y_4 - y_2 y_4 y_6 - y_4^3 + y_4^2}{y_1 + y_4}$$

$$\frac{dy_5}{dt} = \frac{y_3 y_4 y_5 - y_3 y_5^2}{y_4}$$

$$\frac{dy_6}{dt} = 1$$

15

The code also gives us a mapping from the old new parameters using the relation that $y(t) = z(t)^{V_n}$. In this case the result is that:

$$y_1 \mapsto \frac{d}{k} \qquad y_2 \mapsto \frac{k}{h} \qquad y_3 \mapsto \frac{s}{r} \qquad y_4 \mapsto \frac{n}{K} \qquad y_5 \mapsto \frac{hp}{K} \qquad y_6 \mapsto rt$$

6 Full Example

This section gives another example with the full workthrough of the algorithm.

Let us say that we are working with the Verhulst Model of Logistic growth, defined by the differential equation given by:

$$\frac{\mathrm{d}n}{\mathrm{d}t} = rn\left(1 - \frac{n}{k}\right).$$

We start by obtaining the Luarent Polynomial from this equation. By multiplying the right side by t and $\frac{1}{n}$ we get that the Laurent Polynomial of this equation is:

$$rt - rk^{-1}n.$$

We can turn this polynomial into Julia code very straightfforwardly with the code

```
ode_verhulst = @ODEmodel(
    n'(t) = r * n * (1 - n / k),
)
```

We then following the use of the function that we implemented and previously explained `ode_to_matrix`. This function allows us to extract a matrix that describes the equation. We get the result, with variable order `k`, `r`, `n`, `t`, to be:

$$\begin{pmatrix} -1 & 0 \\ 1 & 0 \\ 1 & 1 \\ 1 & 1 \end{pmatrix}$$

Using Proposition 5.1 from [5] we can use this matrix to determine the matrix for the scaling symmetries of our system. We recall that this means we start by finding the *row* Hermite Normal Form of K and a corresponding Hermite Multiplier.

To do this we use the `hnf_with_transform` from Nemo [4], which gives us the resulting *row* Hermite Normal Form of K:

$$H = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}$$

And a corresponding Hermite Multiplier:

$$U = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{pmatrix}$$

Proposition 5.1 tells us that we look at the number of zero rows of H, which we call r. In this case $r = 2$. then taking the last r rows of U gives us the matrix A which gives us a matrix that describes the scaling symmetries of the system.

$$A = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{pmatrix}$$

The final step here is to get the *column* Hermite Normal Form of A, which is used to determine the new system. Using the new `hnf_with_normal_transform_column` function that we implemented earlier, we can find that our Normal Hermite Multiplier is:

$$V = \begin{pmatrix} 1 & 0 & -1 & 0 \\ 0 & -1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

We then can easily find its inverse W to be V^{-1}, so:

$$W = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Now that we have these matrices, what is left to do is to follow the last steps of the algorithm to get our new ODE model.

The last steps of the algorithm require us to split V into V_i and V_n. Again following the algorithm in [5], V_i is the first r columns of V and V_n is the last $n - r$ columns of V. Since $r = 2$ in our example, we get that:

$$V_i = \begin{pmatrix} 1 & 0 \\ 0 & -1 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}$$

and that:

$$V_n = \begin{pmatrix} -1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

Similarly, W is split into W_u and $W_{\eth}$, where W_u is the first r rows of W and $W_{\eth}$ is the last $n - r$ rows of W. In this example we therefore have that:

$$W_u = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{pmatrix}.$$

and that:

$$W_{\eth} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

The matrices we care about here are V_n and $W_{\eth}$. We will then use Proposition 6.2 from [5] in order to get our result. This proposition gives us the formula, that our new reparametrized ODE will be:

$$\frac{dy}{dt} \star F(y^{W_n}) \cdot V_n$$

First we have to define what F defines here. We have that $\frac{dz}{dt} = z \star F(z)$, where z is a vector of all the variables in the ODE, so in this example $z = \begin{pmatrix} k & r & n & t \end{pmatrix}$. So:

$$\frac{dz}{dt} = \begin{pmatrix} \dfrac{dk}{dt} & \dfrac{dr}{dt} & \dfrac{dn}{dt} & \dfrac{dt}{dt} \end{pmatrix}$$
$$= \begin{pmatrix} 0 & 0 & \frac{rnk - rn^2}{k} & 1 \end{pmatrix}$$

Therefore, we have that:

$$F(z) = \begin{pmatrix} 0 & 0 & \frac{rk-rn}{k} & \frac{1}{t} \end{pmatrix}$$

We have that our new parameters, which are in the vector y are going to be $n - r$ in number. Since in this example $n = 4$ and $r = 2$. We have that:

$$y = \begin{pmatrix} y_1 & y_2 \end{pmatrix}$$

We then take $y^{W_\mathfrak{d}}$. The definition of how to take a vector of polynomials to the power of a matrix is explained in section 3.1. So we have that:

$$\begin{aligned}
y^{W_\mathfrak{d}} &= \begin{pmatrix} y_1 & y_2 \end{pmatrix}^{\begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}} \\
&= \begin{pmatrix} y_1^0 y_2^0 & y_1^0 y_2^0 & y_1^1 y_2^0 & y_1^0 y_2^1 \end{pmatrix} \\
&= \begin{pmatrix} 1 & 1 & y_1 & y_2 \end{pmatrix}
\end{aligned}$$

We then have to take $F(y^{W_\mathfrak{d}})$. Since our variable order is n, k, r, t,

$$\begin{aligned}
F(y^{W_\mathfrak{d}}) &= F(\begin{pmatrix} 1 & 1 & y_1 & y_2 \end{pmatrix}) \\
&= \begin{pmatrix} 0 & 0 & 1 - y_1 & \frac{1}{y_2} \end{pmatrix}
\end{aligned}$$

Now, $F(y^{W_\mathfrak{d}}) \cdot V_\mathfrak{n}$:

$$F(y^{W_\mathfrak{d}}) \cdot V_\mathfrak{n} = \begin{pmatrix} 1 - y_1 & \frac{1}{y_2} \end{pmatrix}$$

Then the final result we get that

$$\begin{aligned}
\frac{dy}{dt} &= y \star (F(y^{W_\mathfrak{d}}) \cdot V_\mathfrak{n}) \\
&= \begin{pmatrix} y_1 - y_1^2 & 1 \end{pmatrix}
\end{aligned}$$

With this we can conclude and get our final new reparametrized differential equation to be:

$$\begin{aligned}
\frac{dy_1}{dt} &= y_1 - y_1^2 \\
\frac{dy_2}{dt} &= 1
\end{aligned}$$

Which as we can tell is much simpler than the equation we had in the beginning.

Again we get the new parameters in terms of the new parameters as a result as well here giving us that:

$$y_1 \mapsto \frac{n}{k} \qquad y_2 \mapsto rt$$

7 Conclusion

In this thesis, we implemented a novel methodology for the reparameterization of ordinary differential equation (ODE) models by using the scaling invariants and symmetry reductions. The approach, based on the algorithm by Hubert and Labahn [5], has been implemented through the Julia programming language.

8 Potential Improvements

The main flaw of the algorithm as it is implemented right now is that the user has no control over which parameters can stay fixed. To do this this algorithm would have to consider the output variables. Currently the code returns an error in this situation.

This flaw makes it so that the utility of this algorithm at present is limited.

9 References

[1] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[2] H. Cohen. *A Course in Computational Algebraic Number Theory*. Graduate Texts in Mathematics. Springer Berlin Heidelberg, 2013.

[3] R. Dong, C. Goodbrake, H. Harrington, and Pogudin G. Differential elimination for dynamical models via projections with applications to structural identifiability. *SIAM Journal on Applied Algebra and Geometry*, 7(1):194–235, 2023.

[4] Claus Fieker, William Hart, Tommy Hofmann, and Fredrik Johansson. Nemo/hecke: Computer algebra and number theory packages for the julia programming language. In *Proceedings of the 2017 ACM on International Symposium on Symbolic and Algebraic Computation*, ISSAC '17, pages 157–164, New York, NY, USA, 2017. ACM.

[5] Evelyne Hubert and George Labahn. Scaling invariants and symmetry reduction of dynamical systems. *Foundations of Computational Mathematics*, 13(4):479–516, Aug 2013.