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 Introduction

In their paper "Scaling Invariants and Symmetry Reductions of Dynamical Systems" [4], they discuss the theory behind the algorithm and how it can be implemented.

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

In order to do this, I implemented in Julia, the necessary functions, such as getting the Hermite Multipliers in both row and column form. Most of the references for this. These include the functions [4]

Furthermore, I was (hopefully) able to integrate this algorithm that I had implemented into a more overarching Julia library "StructuralIdentifiability.jl"

Gleb: some points to keep in mind

1. mention relations to the Buckingham Pi theorem and dimension analysis

2 Background

2.1 Hermite Normal Forms and Hermite Multipliers

2.1.1 Hermite Normal Forms (HNFs)

While different sources may have varying definitions of Hermite Normal Forms, the ones we will follow are from [4] 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.

2.1.2 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 [4] 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 $A \cdot V = H$.

2.1.3 Algorithmically determining HNF and Hermite Multipliers

There exists algorithms that can computer Hermite Normal Forms which can be found for example in [1]. 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` [3] 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 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.

2.1.4 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.

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

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

$$(C_m A)^\intercal = A^\intercal C_m$$

We then get the Hermite Normal Form and Hermite Multiplier of this, this is done through a series of row operations on the $A^\intercal C_m$. 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 $A^\intercal C_m$, we get H and V such that:

$$V \cdot (A^\intercal C_m) = H$$

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 $H^\intercal$ looks like this.

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

We then bring flip the rows again giving us:

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

The final step is to flip the first r columns of the resulting $C_m H^\intercal$ to get:

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

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:

$$\begin{aligned} V \cdot (A^\intercal C_m) &= H \\ (V \cdot (A^\intercal) C_m)^\intercal &= H^\intercal \\ C_m A \cdot V^\intercal &= H^\intercal \\ A \cdot V^\intercal C_r &= C_m H^\intercal C_r \end{aligned}$$

We note that in our algorithm applied transpose and multiplication on the right by C_r so our resulting V is $V^\intercal C_r$ 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.

2.1.5 Normal Hermite Multiplier

As explained earlier, the Hermite Normal Form of an integer matrix is unique, however the Hermite Multiplier itself is not, thus for any matrix

3 Algorithm

Now that we have a way to calculate the *column* Hermite Normal Form, we can implement the algorithm to simplify the ODE equations. The proofs of the algorithm are explained in [4], so in this we will be going over step by step each individual part of the algorithm all together to demonstrate how it is implemented in `reparametrizing-odes`.

4 Implementation

4.1 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' [2] 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 [4], 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 [4] 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 [4].

We can illustrate this process with this example.

Let us start with the Predator Prey Model:

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

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 + 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}}:
([3 0 0 0 0 0 1 0; 2 0 1 0 0 0 1 0; 2 0 0 1 0 0 1 0;
1 1 1 0 0 1 0 0; 1 0 1 1 0 0 1 0], [1 0 1 0 0 0 0 0; 0 0 1 1 0 0 0 0])
([1 1 0 0 0 0 0 1; 0 2 0 0 1 0 0 1], [1 0 0 0 0 0 0 0])
```

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

We then have to take into account the $\frac{d(x)}{dt}$ 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 [4].

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

Following this from the predator prey model we get:

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

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

4.2 Example Implementation 1

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

$$\frac{dn}{dt} = rn\left(1-\frac{n}{k}\right).$$

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 straightforwardly 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:

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

Using Proposition 5.1 from [4] 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 [3], which gives us the resulting *row* Hermite Normal Form of K:

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

And a corresponding Hermite Multiplier:

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

Proposition 5.1 tells us that we look at the number of zero rows of H, which we call r, 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.

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

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 and its inverse are:

$$V = \begin{bmatrix} -1 & 0 & 1 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{and} \quad W = \begin{bmatrix} -1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

4.3 Example Implementation 2

5 References

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

[2] 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.

[3] 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.

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

A Appendix