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" [3], 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 [3]

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 [3] 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 [3] 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` [2] 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.

1. We set m to be the number of rows of matrix A and r its rank.
2. Reverse the rows of A.
3. Transpose A.
4. We set H and V to be the `hnf_with_transform` of our new A.
5. We transpose, reverse the rows then reverse the first r columns of our H matrix and the result is our column HNF of the input matrix A.
6. We transpose V and then reverse the first r columns. The result gives us the Hermite Multiplier of our input matrix A.

How this algorithm works:

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 Multipler. 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 $0 \leq | \leq X$.

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.

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 [3], so in this we will be putting together the code to make this algorithm.

4 Implementation

$$\begin{aligned} \frac{\mathrm{d}x}{\mathrm{d}t} &= a - kx + hx^2 y \\ \frac{\mathrm{d}y}{\mathrm{d}t} &= b - hx^2 y \end{aligned}$$

I explain the process to work on this from now on.

Step 1: Form the function F with the Laurent polynomials of the differential equations.

4.1 Example Implementation 1

4.2 Example Implementation 2

5 References

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

[2] 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.

[3] 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