École Polytechnique
CSE303
Instructor: Gleb Pogudin

Using graphs to find differential submodels

Lorenzo Tarricone
Stefan Vayl

Due Date: December 12, 2022

Abstract

Large models described by differential equations are actively used in life science and engineering (you can see an example in this article [4]). The size of the model often makes its analysis and usage computationally expensive. It has been observed that sometimes one can first find a self-contained submodel of smaller size, analyse it, and use the result in the analysis of the larger model. In this research we achieve that by first building a graph representation of the model and then traversing it in a rigorous way to find some base submodels. This process is followed by an exhaustive search for possible union of those submodels. This will yield all the remaining submodels not found in the previous step. We then turned our attention to the development of such algorithm in Julia, with the aim of adding it as an extension to the existing open source package "StructuralIdentifiability.jl" [5].

Purposes

Our work adds to a big library which aims to help pure scientists with their data analysis [2]. In particular it is designed to help them simplify the number of computations needed for simulations and shorten the time required for them to bring out their experiments. Our functions aim to split the initial model composed by differential equations into all possible submodels of smaller size. Such subsystems might be much less heavy to simulate in terms of computation and they could be very useful when it comes to work with a particular input or if the scientist wants to study a particular aspect of the model. Sometimes, it is also the case that when researchers build big abstract models they cannot get enough data to identify its parameters. Submodeling may help to address this problem by providing submodels that are completely solvable and that can sometimes simulate the system with enough accuracy.

Main goals

1. Get a deep understanding of differential submodels and the problem of "Structural Identifiability", that is the context of our research.
2. Develop an algorithm which we can use to find all the differential submodels of a given model.
3. Implement some working code with Python, that can be found here[3].
4. Optimise the code with the aim of reaching a running time linear in the size of the output.
5. Rewrite the code in Julia and merge our results with main project "Structural Identifiability".

1 Formal setting and Examples

We follow a simplified version of the formal setting provided by the "Structural Identifiability" problem. Here we have a system of differential equations Σ of the following form:

$$\begin{cases} \dot{x}(t) = f(x(t)) \\ y(t) = g(x(t)) \end{cases} \tag{1}$$

Where:

1. $x(t)$ are the states of our models that depend on time
2. $y(t)$ are the observations of our model, that is what we measure in our experiment
3. $\dot{x_i} = f_i(x)$ is the ith equation of the model

To simplify the notation we will now always write x_n instead of $x_n(t)$.

We now provide two examples of very simple models that we will stick to, in order to showcase how our algorithm works. These models have been chosen ad hoc because despite their simplicity, they have some interesting properties that we will discuss later in this paper.

$$\begin{cases} \dot{x_1} = ax_1 + x_2^2 \\ \dot{x_2} = bx_2 + cx_1x_2 \\ \dot{x_3} = x_1 + x_2 + x_3 \\ y_1 = x_1 \\ y_2 = x_3 \end{cases} \qquad \begin{cases} \dot{x_1} = ax_1 \\ \dot{x_2} = x_1 + bx_2^2 \\ \dot{x_3} = x_3 - cx_1x_2x_3 \\ y_1 = x_1 \\ y_2 = x_1 + x_2 \\ y_3 = x_3 \end{cases} \qquad \begin{cases} \dot{x_1} = x_1 \\ \dot{x_2} = x_2 \\ \dot{x_3} = x_3 \\ \dot{x_4} = x_4 \\ y_1 = x_1 \\ y_2 = x_2 \\ y_3 = x_3 \\ y_4 = x_4 \end{cases}$$

2 Introduction

Our first step was to define in a rigorous way what a submodel is. After some time of research we ended up with few main criteria that are summed up in the following definitions and trivial results (that's why a proof will not be provided):

Definition 2.1 *A **submodel** is a subset of the original model where all the states equations and observations included contain observations and states of which law describing their dynamics is present in the model. Formally we say*

that:
$\dot{x}_i(t) = f_i(x_K) \in$ *submodel* $\longrightarrow \dot{x}_k = f_k(x_{K'}) \in$ *submodel* $\forall k \in K$ *subset of indices of the states involved in the dynamics of equation* i
$y_i(t) = g_i(x_L) \in$ *submodel* $\longrightarrow \dot{x}_l = f_l(x_{L'}) \in$ *submodel* $\forall l \in L$ *subset of indices of the states involved in the description of the observation*

Definition 2.2 *The dimension of a submodel* Σ' *is the cardinality of the set of states included in the submodel*

$$d(\Sigma') = |\{x_i : x_i \in \Sigma'\}|$$

Remark*: Notice that when we define the dimension of the submodels we don't take into account the number of observations/outputs of the model. The idea is that while states are the dynamic entity of the models that add complexity, observations are variables that allow us to have information on the system we are studying. Because of that we will never be worse-off by adding a "coherent" observation to the system and we will adopt the convention that every model returned will include the maximum number of observations for that model.*
(We define a coherent observation for a model an observation that depends just on states present in the submodel)

For ***example*** *given the an original model (left) and two valid submodels of the same dimension (center and right):*

$$\begin{cases} \dot{x_1} = ax_1 \\ \dot{x_2} = bx_2 + cx_1x_2 \\ \dot{x_3} = x_3 \\ y_1 = x_1 \\ y_2 = x_1 + x_2 \\ y_3 = x_3 \end{cases} \qquad \begin{cases} \dot{x_1} = ax_1 \\ \dot{x_2} = x_1 + bx_2^2 \\ y_2 = x_1 + x_2 \end{cases} \qquad \begin{cases} \dot{x_1} = ax_1 \\ \dot{x_2} = x_1 + bx_2^2 \\ y_1 = x_1 \\ y_2 = x_1 + x_2 \end{cases}$$

We will return just the submodel on the right, because it has one more "free observation" that it's always desirable to have

Observation 2.1 The whole model and the model containing no equations and no observations are submodels. We will say that a submodel is non-trivial if it's a submodel and it's not one of these two.

$$\Sigma, \emptyset \in submodels$$

Observation 2.2 The union of two submodels is a valid submodel of dimension at least as big as the bigger one of the two submodels and no bigger than the sum of the dimension of the two original models

$$d(\Sigma') + d(\Sigma) \geq d(\Sigma' \cup \Sigma'') \geq min[d(\Sigma'), d(\Sigma'')]$$

Observation 2.3 Given two submodels where one of the two is s subset of the others, the unions of these two submodels will correspond to the biggest of the two

$$\Sigma'' \subset \Sigma', d(\Sigma'') \leq d(\Sigma') \longrightarrow \Sigma'' \cup \Sigma' = \Sigma'$$

3 Graph representation

We designed and implemented an easy way of representing systems of differential equations Σ as a graph $G =< V, E >$. Each node is either one of the states $x_i's$ or one of the observations $y_i's$ and this allows us to create a partition of the nodes based on their types. We will therefore define $X := \{v \in V : v \text{ is a state}\}$ and $Y := \{w \in V : w \text{ is an observation}\}$. We say that there exists an edge from node n to node m if m appears in the equation of n, meaning that (depending on the fact that n is a state or an observation) m is either a state appearing in the law of n or m is a state appearing in the equation of the observation n. Notice that this method is neglecting any particular ratio, coefficient or nonlinear function applied on the states on the RHS of our equations. This allowed us to load any system of differential equations and construct a graph which will be used as input for the main algorithm.

4 Algorithm

This algorithm as previously mentioned takes as input the graph representation G of the system Σ and produces as output the list of all possible submodels. The main steps of the algorithm are the following:

1. For each observation in Y create a list, then run the algorithm DFS and add to the element found to that list.

2. For each returned list add those other observations in Y that have all their edges pointing to elements in the list.

3. Search for all the possible unions and if some new model is found add it as a new list

4. Return the lists created, each one will be a submodel.

The different versions of the algorithm listed below differ in how the search of the unions in the third step is implemented

4.1 First Version

The first version of the algorithm has a find union function that iterating over the dimension of the unions (pairwise, three elements, etc...) performs the following checks:

1. Check if the size is smaller or equal than the one of the complete model (nothing would be added at that point) and that the size of the union is strictly larger the size of the biggest of the two models (because this would be equal to check that we are not taking the union of a submodel and its (sub)submodel.

2. Check if the submodel that passed the first step was not already found in a previous step.

3. Add this new valid submodel to the list of submodel and return it.

This initial and naive approach is extremely expensive in terms of computations, because it makes a number of different checks on a list of items that could be potentially exponential in length with respect to the size of the input.

4.2 Second Version

The limitations of the first algorithm brought us to adopt a new approach in finding the unions. This time instead of checking some properties of the given submodels at every iteration, we take all the possible unions using an inductive procedure on the number of elements that come out from the step 2 of the algorithm. We will add these elements to the original list of submodels just if these are not already present in the original list (meaning that one submodel was fully contained into another).

5 Results

The proposed algorithm gave back the correct answer for all the three examples showed above.
The first example was the easiest one, given that the only nontrivial submodel that needs to be returned is the one containing the first two states and the first observation :

$$\begin{cases} \dot{x}_1 = ax_1 + x_2^2 \\ \dot{x}_2 = bx_2 + cx_1x_2 \\ y_1 = x_1 \end{cases} \tag{2}$$

Our algorithm gave the following output:

```
Vector{ODE{fmpq_mpoly}}:
x2'(t) = c*x1(t)*x2(t) + b*x2(t)
x1'(t) = x2(t)^2 + a*x1(t)
y1(t) = x1(t)
```

In the second example we had a nested structure that we wanted to retrieve and our algorithm correctly finds that the two non-trivial submodels:

$$\begin{cases} \dot{x_1} = ax_1 \\ y_1 = x_1 \end{cases} \qquad \begin{cases} \dot{x_1} = ax_1 \\ \dot{x_2} = x_1 + bx_2^2 \\ y_1 = x_1 \\ y_2 = x_1 + x_2 \end{cases}$$

And again our algorithm gave the following output:

```
2-element Vector{ODE{fmpq_mpoly}}:
x2'(t) = b*x2(t)^2 + x1(t)
x1'(t) = a*x1(t)
y1(t) = x1(t)
y2(t) = x1(t) + x2(t)

x1'(t) = a*x1(t)
y1(t) = x1(t)
```

In the third and last example we wanted the algorithm to be able to find all the possible combinations of states given that the system is composed of "singletons" models. The correct is the following:

$$\begin{cases} \dot{x_1} = x_1 \\ y_1 = x_1 \end{cases} \quad \begin{cases} \dot{x_2} = x_2 \\ y_2 = x_2 \end{cases} \quad \begin{cases} \dot{x_3} = x_3 \\ y_3 = x_3 \end{cases} \quad \begin{cases} \dot{x_4} = x_4 \\ y_4 = x_4 \end{cases}$$

$$\begin{cases} \dot{x_1} = x_1 \\ \dot{x_2} = x_2 \\ y_1 = x_1 \\ y_2 = x_2 \end{cases} \quad \begin{cases} \dot{x_1} = x_1 \\ \dot{x_3} = x_3 \\ y_1 = x_1 \\ y_3 = x_3 \end{cases} \quad \begin{cases} \dot{x_1} = x_1 \\ \dot{x_4} = x_4 \\ y_1 = x_1 \\ y_4 = x_4 \end{cases} \quad \begin{cases} \dot{x_2} = x_2 \\ \dot{x_3} = x_3 \\ y_2 = x_2 \\ y_3 = x_3 \end{cases}$$

$$\begin{cases} \dot{x_2} = x_2 \\ \dot{x_4} = x_4 \\ y_2 = x_2 \\ y_4 = x_4 \end{cases} \quad \begin{cases} \dot{x_3} = x_3 \\ \dot{x_4} = x_4 \\ y_3 = x_3 \\ y_4 = x_4 \end{cases} \quad \begin{cases} \dot{x_1} = x_1 \\ \dot{x_2} = x_2 \\ \dot{x_3} = x_3 \\ y_1 = x_1 \\ y_2 = x_2 \\ y_3 = x_3 \end{cases} \quad \begin{cases} \dot{x_1} = x_1 \\ \dot{x_3} = x_3 \\ \dot{x_4} = x_4 \\ y_1 = x_1 \\ y_3 = x_3 \\ y_4 = x_4 \end{cases}$$

$$\begin{cases} \dot{x_2} = x_2 \\ \dot{x_3} = x_3 \\ \dot{x_4} = x_4 \\ y_2 = x_2 \\ y_3 = x_3 \\ y_4 = x_4 \end{cases} \qquad \begin{cases} \dot{x_1} = x_1 \\ \dot{x_2} = x_2 \\ \dot{x_4} = x_4 \\ y_1 = x_1 \\ y_2 = x_2 \\ y_4 = x_4 \end{cases}$$

And this is the result produced by our algorithm (we will not show all of it to save space):

```
15-element Vector{ODE{fmpq_mpoly}}:
x2'(t) = x2(t)
y2(t) = x2(t)

x3'(t) = x3(t)
y3(t) = x3(t)

x2'(t) = x2(t)
x3'(t) = x3(t)
y2(t) = x2(t)
y3(t) = x3(t)

x4'(t) = x4(t)
y4(t) = x4(t)

...
```

6 Formatting the Output

6.1 Integration into the Library

Despite initial implementations being done in Python, one needs to obtain a working Julia version integrated with structure and specifics of the "StructuralIdentifiabilty.jl" library, part of which the project eventually is supposed to become.
This process can be divided into two main parts:

1. Direct translation Python code into Julia
2. Integration to the library

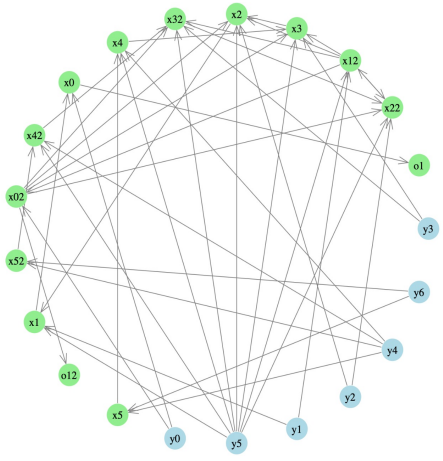
Let's look more detailed into the second aspect. In the Python version there is a script which allows the user to use as an input a .txt file where one can provide

an initial system of equations. In the meantime, in Julia there is data structure of ODE models defined in the core of the library which allows to access and manipulate equations and variables. The important feature of this structure is the way of how equations are being defined as it uses the Nemo library [6] module and creates PolynomialRing for each of them for better efficiency in memory usage. So, to achieve integration one has to proceed with following steps:

1. Adapt already existing code to the restrictions on the input and the output of the library.
2. Track and preserve all the variables being involved through all of the processes without changes.
3. Construct an ODE object from raw submodels which are just lists of variables being involved.
4. Change parent PolynomialRing for each equation of the submodel.
5. Merge code with other modules of the library.

7 Plot function

To give the user/scientist that is using the package a better and more intuitive understanding of the modelling problem, we implemented also a very simple function that allows the user, starting from existing packages for plotting [1], to plot his original system of ODEs as a directed graph (as described in this report). An example of a real and more complex system of ODEs that we analysed is the following:



8 Limitations and Possible Developments

The biggest limitation in the development of new algorithms is that we have a lower bound in efficiency given by the fact that the function to find the unions will have to return an exponential number of submodel in the worst case scenario (that is the example 3 provided above, where all the states are not communicating between each other and are therefore all independent submodels of dimension one).

To have a more efficient result one could try to use the exploration procedure of the original graph at point 2 of the algorithm to create a sort of "meta-graph", that is a graph representing all the dependencies in terms of inclusions between the variables. This will in turn provide some information to the search union function at point 3, that will "spare" some unions. This is because of the result provided by observation 2.3 that gives us exactly the result of a union of such types of subsets.

References

[1] Library of functions that we used to plot. https://github.com/JuliaGraphs/GraphPlot.jl.

[2] Our fork for the repository of structural identifiability. https://github.com/StefanVaylBX2023/StructuralIdentifiability.jl.

[3] Our working repository. https://github.com/StefanVaylBX2023/CSE303.

[4] Guillaume Ballif, Frédérique Clément, and Romain Yvinec. Nonlinear compartmental modeling to monitor ovarian follicle population dynamics on the whole lifespan. working paper or preprint, July 2022.

[5] Ruiwen Dong, Christian Goodbrake, Heather A Harrington, and Gleb Pogudin. Differential elimination for dynamical models via projections with applications to structural identifiability. *arXiv preprint arXiv:2111.00991*, 2021.

[6] Claus Fieker, William Hart, Tommy Hofmann, and Fredrik Johansson. Nemo/Hecke: Computer Algebra and Number Theory Packages for the Julia Programming Language. In *ISSAC '17 - International Symposium on Symbolic and Algebraic Computation*, Kaiserslautern, Germany, July 2017.