Modia and Julia for grey box modeling

Frederic Bruder, Lars Mikelsons

Angaben zur Veröffentlichung / Publication details:

Bruder, Frederic, and Lars Mikelsons. 2021. "Modia and Julia for grey box modeling." In *Proceedings of 14th Modelica Conference 2021, Linköping, Sweden, September 20-24, 2021*, edited by Martin Sjölund, Lena Buffoni, Adrian Pop, and Lennart Ochel, 87–95. Linköping: Linköping University Electronic Press. https://doi.org/10.3384/ecp2118187.

Modia and Julia for Grey Box Modeling

Frederic Bruder[1] Lars Mikelsons[1]

[1] Chair of Mechatronics, Augsburg University, Germany, {frederic.bruder,lars.mikelsons}@uni-a.de

Abstract

During the process of modelling an existing dynamic physical system, it may be hard to capture some of the phenomena exactly on the basis of only textbook-equations. With measurement data from the real system, approximators like artificial neural networks can help improve the models. However, simulation and machine learning are usually done in different software applications. A unified environment for modeling, simulation and optimization would be highly valuable. We here present a framework within the Julia programming language that encompasses tools for acausal modeling, automatic differentiation rsp. sensitivity analysis involving solvers for differential equations. We use it to build and evaluate an easily interpretable model based on both physics and data.

Keywords: Grey Box Modeling, Hybrid Modeling, Scientific Machine Learning, Modia, Julia

1 Introduction

1.1 Usefulness of Acausal Modeling

Equation-based acausal modeling like in Modelica has considerable benefits for the model author when compared to signal-based modeling.

It allows them to structure model equations and variables hierarchically, which greatly promotes later reuse of the components. It lets the author focus on model topology rather than signal flow and its direction within the model. Connect equations instead of assignments permit signal flow in both directions so that the compiler can decide what the direction will be.

Well-established Modelica Compilers automatically improve the numerical behaviour of the models, e.g. by reducing the size of nonlinear systems with the help of tearing or by reducing the index of Differential-Algebraic-Equations (DAE).

In short: acausal modeling is highly useful because it eases the job for authors of white box (i.e. mechanistic physics-based) models.

1.2 Grey Box Modeling

In the case of GBM ('Grey Box Modeling' or '... Model'), the author combines approximators like ANN (artificial neural networks) with trusted white box model equations. The goal of this technique is to transfer physical knowledge into a model that can be improved with the help of machine learning. Others referred to this field as 'Scientific Machine Learning' (Rackauckas, Ma, et al. 2020), 'Hybrid Physics Guided Machine Learning' (Rai and Sahu 2020) or simply 'Hybrid Modeling' (Willard et al. 2020). There are different motivations to do this:

- Known white box models may fail to describe the dynamics of an existing system appropriately. In this case the insertion of ANN into a model may help to reduce model error w.r.t. ground truth data collected the real physical system. Imagine a situation in which you first model a physical system. Take, for example, a simple model to predict the temperature in a lake as in (Karpatne et al. 2018). Although you stuck to well-known mechanistic equations, you see that the prediction error is too high. Due to the nature of the errors, you suspect a systematic error, not a stochastic one. To improve your model, you have a few options: You could try and improve your model by altering the equations you used. You would have to think about your model assumptions and whether or not they hold. This may, after possibly a lot of work, provide you with new physical insights. If you are more interested in fast results or if your system is just too complicated to fully grasp, you may consider using a machine learning-assisted approach that your initial model as a starting point and optimizes the 'flexible' parts to improve the predictions.

- White box models may require more computational resources than their target platform offers. Performance-critical parts may then be exchanged by ANN in order to create more efficient surrogate models. Thus, this technique enables model authors to trade model accuracy for better performance. An example of this application can be found in (Ma et al. 2021).

There are some complications with well-established Modelica compilers when it comes to GBM:

- Small modifications of existing Modelica models may require substantial refactoring. For example, you may have to declare a few new (abstract) components if you just want to replace a single equation in a model.

- It may be hard for a user to interpret or edit the structure of causalized models. This is problematic when it is not yet clear where exactly the ANN shall be inserted into the model. ANN that are defined as mathematical functions typically have an 'input layer' that is nonlinearly transformed to an 'output layer'. Said transformation may be arbitrarily complex. During the causalization process of the model equations, signal flow may be 'reversed' w.r.t. the intended direction so that the ANN is bound to end up in a root finding loop. This might lead to subpar numerical performance. To prevent this situation, the model author may wish to decide where exactly the ANN is included **after** the causalization process. It is, however, not very convenient to deal with C-Code output from a Modelica compiler.

- Another issue is the ANN training process. ANN are typically optimized w.r.t. their training/hyper parameters in languages that promote fast prototyping (e.g. Python, Julia, R, ...) with the help of tools like automatic differentiation. The latter kind of tools often requires that the function to be differentiated be formulated in the same language as the prototyping language.

It would be much more convenient to have modeling, simulation, automatic differentiation / sensitivity analysis and optimization algorithms in a single unified environment suited for rapid prototyping.

1.3 Julia

Julia (Bezanson et al. 2012) is one of such languages that promote fast prototyping. Julia has become popular in the scientific community because of the high level of performance it can reach without the need to use precompiled libraries created in a different language. The latter workflow is common e.g. in Python. On top of that, an ecosystem of packages useful for scientific calculations has gathered around it. Moreover, Julia offers metaprogramming capabilities that make it possible to create DSL (domain-specific languages).

1.4 Modia / TinyModia for GBM

Modia (Elmqvist and Otter 2017) is such a DSL that can be used within Julia. Its authors describe it as a testbed for new features for the Modelica language that 'shall be both simpler and more powerful than Modelica 3.3'.

Modia offers features highly valuable to GBM since it solves some of the issues listed in 1.2. In our present research, we are, however, using TinyModia (Elmqvist and Otter 2021) v.0.7.2 that had some features not yet present Modia.

- Models are represented as hierarchical `NamedTuples`. They can be modified with a mechanism called 'merging', their `Array` fields can even be manipulated directly. So, after a model has been declared, it can be modified equation-wise. This allows for a very convenient GBM workflow: you can declare a base model and then derive different versions of grey box models where different sets of equations are modified.

  Whereas in Modelica, one would typically have to define new (replaceable) component `Models` with a different set of equations and then derive simulation `Models` as a combination of those new components.

- During model 'instantiation', a Julia method (`getDerivatives!`) that calculates the state derivatives of the model is generated with the help of metaprogramming. Due to the open nature of TinyModia, the AST (abstract syntax tree) that, when evaluated, defines this new method, can be altered as well.

  Modelica Compilers, on the other hand, do not expose the AST of the simulation code to Julia. While they output source code of the simulation model, working with this representation would be less convenient: one would have to modify the output C-Code manually, without Julia's metaprogramming functionality.

- With a modifiable method `getDerivatives!`, the model author can take advantage of packages other than TinyModia to simulate the model. With this workflow, it even becomes possible to perform AD through solvers using the scientific packages available for Julia.

1.5 Structure of this Paper

This paper is structured as follows. In section 2, we describe a planar slip-based vehicle model which we modified to fit to our tests. Using this model as an example, we elaborate how models can be turned into GBM by replacing a set of model equation terms by trainable neural networks. In section 3, we proceed by detailling the time horizon-based training scheme that we used to train said neural networks. We then describe how we designed our tests with the modified and optimized vehicle model in section 4 and discuss the results. After that in 5, we describe the framework of tools we used to set up a GBM. In addition, we describe possible features for TinyModia/Modia that would have made our workflow a lot easier. Next, we name some of our future research perspectives involving the mentioned tool set and the Grey Box methodology. We then finish with a conclusion.

The main contribution of this paper is the detailed framework of tools that we use for Grey Box Modelling. We describe how to set up a model and enhance it with artificial neural networks. We sketch a training loop that is capable of optimizing such grey box models. We make suggestions for new features that would have made this process a lot more straight-forward.

2 GBM of a Single Track Vehicle

2.1 Original Model and our Modifications

We applied the ideas explained above to an NLSTM (non-linear single track model). It makes use a well-known slip-based tire model as in (Pacejka 2005). The original model without suspension dynamics from (Velenis, Frazzoli, and Tsiotras 2009), on which our model is based, focusses on stability of cornering maneuvers, whereas we needed a more general purpose model. So we chose an alternative set of dynamic states $\boldsymbol{u}$ for our NLSTM:

$$\boldsymbol{u} = \begin{bmatrix} \dot{x} & \dot{y} & \psi & \dot{\psi} & x & y & \omega_F & \omega_R \end{bmatrix}^T \tag{1}$$

x and y denote the location of the center of mass of the vehicle. ψ measures the yaw angle / heading direction of the vehicle, ω_F and ω_R describe the rotational velocity of the front and rear wheel. Furthermore, we modified the original slip-based friction model to be less numerically stiff. For this, each occurrence of the term on the RHS (right-hand side) of

$$\texttt{hard\_pole}(x) = |x|^{-1} \tag{2}$$

was replaced by the RHS of

$$\texttt{smooth\_pole}(x) = \begin{Bmatrix} |x|^{-1} & \text{if } |x| > x_i \\ -\frac{1}{2}|x_i|^{-3}x^2 + \frac{3}{2}|x_i|^{-1} & \text{else} \end{Bmatrix} \tag{3}$$

with $x_i = 10^{-3}$. $\texttt{smooth\_pole}(x)$, as the name suggests, is a symmetric continuously differentiable function that replaces the actual 'pole region' of $\texttt{hard\_pole}(x)$ for $x \in [-x_i, x_i]$ by an inverted parabola tangent to the original function. This modification effectively prevents divisions by zero.

To have velocity-dependent friction in the model, a **linear** air drag force $\boldsymbol{f}_{drag}$ was applied to the center or mass of the vehicle:

$$\boldsymbol{f}_{drag}(\boldsymbol{x}) = w \cdot \begin{bmatrix} \dot{x} & \dot{y} \end{bmatrix}^T \tag{4}$$

Air drag parameter w was set to 30 Ns/m.

Apart from these three modifications, our RM (reference model) and the original model share the same equations and parameters. Note however, that x_i was set arbitrarily. In a real application, it may be helpful to set this parameter, introduced to enhance numerical performance, based on collected data.

The model inputs are the same as those of the original model:

$$\boldsymbol{i}(t) = \begin{bmatrix} \delta(t) & T_F(t) & T_R(t) \end{bmatrix}^T \tag{5}$$

where δ is the steering angle, T_F and T_R are the engine torques acting on the front rsp. the rear wheel.

2.2 Replaced Equations

In the original model, normal forces acting between the ground and the wheels are denoted by f_{Rz} (rear wheel) and f_{Fz} (front wheel). Based on the original model equations, the following equations hold:

$$f_{Rz} = \frac{mgA}{A - h\mu_{Rx} + l_R} \tag{6}$$

$$f_{Fz} = mg - f_{Rz} \tag{7}$$

$$A = h(\mu_{Fx}cos(\delta) - \mu_{Fy}sin(\delta)) + l_F \tag{8}$$

m, g, h, l_R and l_F are fixed model parameters. A is a convenience variable introduced for readability. The remaining symbols in Table 1 are model variables which can change during simulation.

Table 1. remaining model variables in Equations 8 to 7

Symbol(s)	*Meaning*
δ	steering angle
μ_{Fx}, μ_{Fy}	friction quantities (front wheel)
μ_{Rx}	friction quantity (rear wheel)

Equations 8 to 7 can be restructured as:

$$\begin{bmatrix} f_{Rz} \\ f_{Fz} \end{bmatrix} = \boldsymbol{f}_{Xz}(h, \mu_{Fx}, \delta, \mu_{Fy}, l_F, m, g, \mu_{Rx}, l_R) = \boldsymbol{f}_{Xz}(\boldsymbol{v}) \tag{9}$$

$\boldsymbol{v}$ is a convenience vector holding all the function arguments. Like this, $\boldsymbol{f}_{Xz}$ calculates both f_{Rz} and f_{Fz} from arguments $\boldsymbol{v}$.

In order to turn the original model into a flexible GBM, function $\boldsymbol{f}_{Xz}$ is replaced by an ANN named $\hat{\boldsymbol{f}}_{Xz}(\boldsymbol{\theta}, \boldsymbol{v})$. It depends on $\boldsymbol{\theta}$, a set of training parameters and $\boldsymbol{v}$. It was built from fully connected layers as depicted in Table 2. This ANN was defined with the help of `Flux.jl`

Table 2. layers of ANN $\hat{\boldsymbol{f}}_{Xz}$

layer	*# inputs*	*# outputs*	*activation*
input	9	10	Relu
hidden	10	10	Relu
output	10	2	Identity

(Michael Innes et al. 2018) and is initialized using random numbers drawn from a `glorot_uniform` distribution.

Note that the ANN takes as many inputs as the pair of equations it replaces. This would not be neccessary, one could exclude the fixed parameters and make the input space 4-dimensional instead of 9-dimensional. The input space is kept as large as that of $\boldsymbol{f}_{Xz}$ so that the knowledge about which model variables change and which do not is not required.

3 Model Training

3.1 Loss Function

The method into which the ANN was inserted specifies the RHS of an ODE (ordinary differential equation). So, `getDerivatives!` calculates $\dot{\hat{\boldsymbol{u}}}(\hat{\boldsymbol{u}}(t), \boldsymbol{\theta}, t)$.

In order to train the parameters of the ANN, a loss function $l(\boldsymbol{\theta})$ needs to be formulated. For this test, we chose an MSE-based (mean squared error) function:

$$l_{MSE}(\boldsymbol{\theta}, C, S) = \frac{1}{|C| \cdot |S|} \sum_{c \in C} \sum_{s \in S} (\hat{u}_s(\boldsymbol{\theta}, t = c) - u_s(t = c))^2 \tag{10}$$

l_{MSE} compares a single solution $\hat{\boldsymbol{u}}(\boldsymbol{\theta},t)$ of the ODE to a ground-truth solution $\boldsymbol{u}(t)$.

Normally, when ground-truth data is collected using sensors from real physical systems, $\boldsymbol{u}(t)$ is only known at *specific points in time*. We call these *checkpoints* and C denotes the set of checkpoints relevant to l_{MSE}.

S, on the other hand, is a set of states relevant to l_{MSE}. Typically, ground truth measurements cannot capture all of the dynamic states that would be neccessary to fully specify the state of an arbitrary model of the real system. Our system state in Equation 1 has 8 entries. If you can only measure, say, the planar position of the vehicle, S would hold only 2 elements. $u_s(t)$ rsp. $\hat{u}_s(t)$ denote the trajectories of single elements of the system states in time.

3.2 Training Procedure

When we first tried to minimize l_{MSE} with the full set of available checkpoints, we would end up with with very bad local minima. The solution trajectory to the model ODE would look like an underfitting approximator: instead of following the checkpoint trajectory closely, it would resemble a smoothed cutting-corners version of the checkpoint trajectory. This is due to the formulation of our loss function: it compares the model ODE trajectory against all of the measurements at the checkpoint times at the same time and may reach local minima with the last few checkpoints, whereas earlier checkpoints are far missed.

However, this is not what we wanted. If early model predictions are very far off, we do not care about whether the model later succeeds in hitting checkpoints again. We would rather expect later checkpoints to be missed even further. Unless we are dealing with systems that evolve to a stable equilibrium.

To overcome this issue, a **growing horizon** training scheme has been implemented. It adds an outer loop to our training loop: we start with horizon $C_0 = [t_0, t_1, ..., t_{init}]$ and apply our inner training loop to optimize $\boldsymbol{\theta}$. The inner loop then keeps iterating until we hit a breaking condition involving distance metric d_{max}:

$$d_{max}(\boldsymbol{\theta}, C_n, S) = \max_{c \in C_n, s \in S} |\hat{u}_s(\boldsymbol{\theta}, t = c) - u_s(t = c)| \quad (11)$$

If d_{max} falls below threshold d_t, we save our last training parameter set $\boldsymbol{\theta}^{0*}$, grow the training horizon to $C_1 = [t_0, t_1, ..., t_{init}, t_{init+1}]$ and then further optimize $\boldsymbol{\theta}^{0*}$ over that new horizon. We repeat this process until our inner loop has optimized $\boldsymbol{\theta}$ over the final horizon $C_{final} = C_{|C|-init-1}$. This training scheme heavily improved the quality of our model predictions on the training set.

One may wonder why we constructed d_{max}, an arbitrary distance metric, to be compared against a threshold instead of governing l_{MSE} instead. Both metrics evaluate to 0 for a perfect $\boldsymbol{\theta}$. However, $d_{max} < d_t$ is a very intuitive condition: none of the trajectories of the states in S may deviate further than d_t off the recorded time series data at the checkpoint times. If l_{MSE} was compared to a fixed threshold instead, the condition would become more and more permissive towards single outliers the larger the horizon grows.

4 Evaluation

4.1 Generation of Training Data

For lack of a real vehicle on which to collect data, we made use of the original vehicle model to generate training data for our GBM. Our data generation is limited to open-loop DLC (double lane changes) that the vehicle performs after its initial state has been set to a straight slip-free state:

$$\boldsymbol{u}_0 = 30 \cdot [1, 0, 0, 0, 0, 0, r_F^{-1}, r_R^{-1}]^T \quad (12)$$

Figure 1 and Table 3 illustrate how model input δ(t) is varied over time. During all DLC maneuvers, model

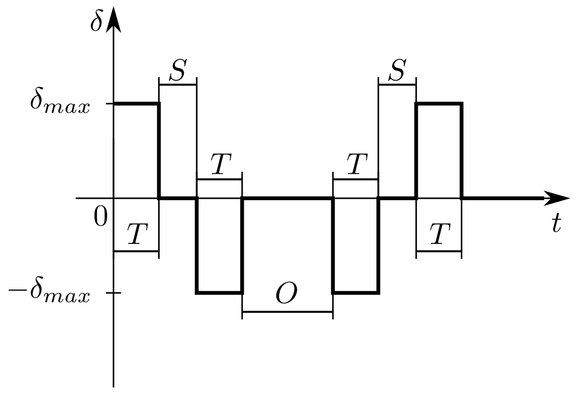

Figure 1. δ(t) during a DLC maneuver

Table 3. DLC parameters

Parameter	*Meaning*
δ_{max}	maximum absolute steering angle
T	**turn** time
S	lane **switch** time
O	time on the **oncoming** lane

inputs T_R and T_F are kept constant over time. T_R is fixed at 0, whereas the value of T_F is a maneuver parameter.

Thus, a single DLC maneuver of ours is parametrized using a set of five parameters:

$$\boldsymbol{p}_{DLC} = [\delta_{max}, T, S, O, T_F] \quad (13)$$

4.2 Results

In the following, the training dataset was limited **to only one** trajectory that was produced with the DLC parametrization shown in Equation 14.

$$\boldsymbol{p}_{DLC}^{train} = \boldsymbol{p}_{DLC}^{1} = [\frac{\pi}{8}\text{rad}, 1\text{s}, 1\text{s}, 2\text{s}, 1000\text{Nm}] \quad (14)$$

The final training horizon was set to an equidistant grid:

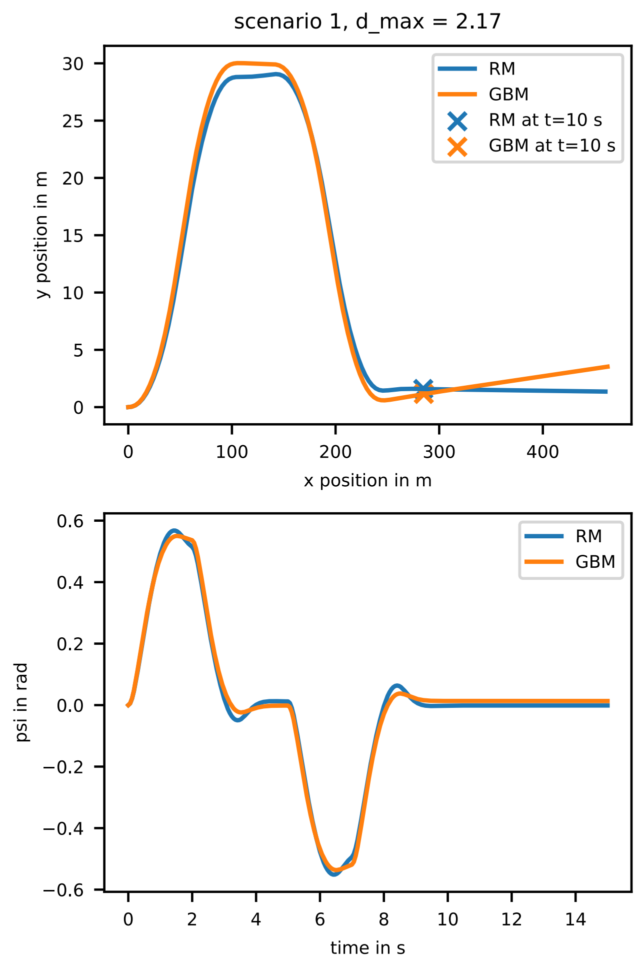

Figure 2. DLC on the training maneuver parametrized by $\boldsymbol{p}_{DLC}^{train}$

$$C_{final} = \frac{10}{32}\text{s} \cdot [1, 2, ..., 32] \tag{15}$$

C_{init}, on the other hand, held only the first three entries of C_{final}. S, the set of compared system states, was set to

$$S := \{x, y\} \tag{16}$$

so that the optimization algorithm could only optimize the ANN parameters on the basis of vehicle location data. It was **not** provided information about the vehicle yaw angle ψ. The fact that the normal load forces are very influential effects within the model and that two (scalar) functions are trained at once on only one trajectory makes this training task particularly hard.

Figure 2 displays how the GBM (after training) and the RM behave during the single training maneuver. The total simulation time was set to 15s. The top subplot of Figure 2 shows how both RM and GBM moved on the x-y plane. *Note that the axes are not scaled equally.* Both models start in the origin of the coordinate system (as specified in Equation 12). The crosses mark where both models were located at $t = 10$s, i.e. the last time at which states in S were compared to train the ANN. It can be seen how the GBM follows the track of the RM closely at first. Larger differences occur mostly after the 10s-mark. This is due to the imperfect prediction of ψ, which can be seen in the bottom plot. After the completion of the DLC, a noticeable difference remains between the predicted ψ of the GBM and that of the RM.

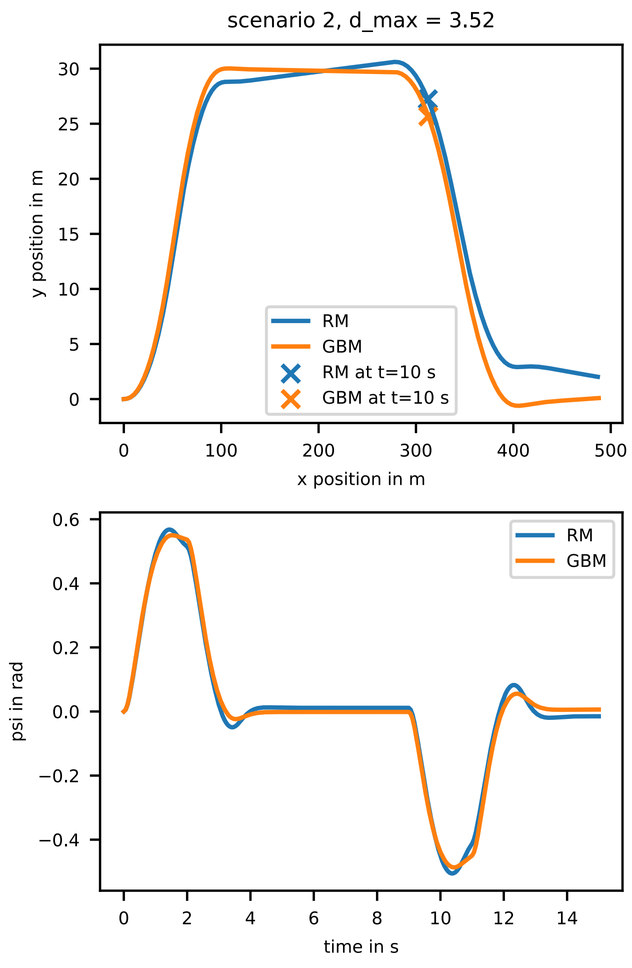

Figure 3. DLC on the training maneuver parametrized by $\boldsymbol{p}_{DLC}^{2}$

A first validation maneuver is specified in Equation 17. This scenario differs from the training scenario by the time spent on the oncoming lane. It is now three times as long.

$$\boldsymbol{p}_{DLC}^{2} = [\frac{\pi}{8}\text{rad}, 1\text{s}, 1\text{s}, 6\text{s}, 1000\text{Nm}] \tag{17}$$

The results can be seen in Figure 3. This time, the largest deviations can be seen after the vehicle returns to the initial lane. Once again, the final ψ of the RM and the GBM are different so that over time, the tracks will keep drifting further apart.

Another interesting maneuver is defined in Equation 18.

$$\boldsymbol{p}_{DLC}^{3} = [\frac{\pi}{8}\text{rad}, 5\text{s}, 1\text{s}, 2\text{s}, 1000\text{Nm}] \tag{18}$$

Compared to $\boldsymbol{p}_{DLC}^{1}$, it only differs by the turning time, which here is 5 times as long. This parametrization no longer looks like a DLC maneuver, but it still demonstrates how the tracks shown in Figure 4 drift apart over time.

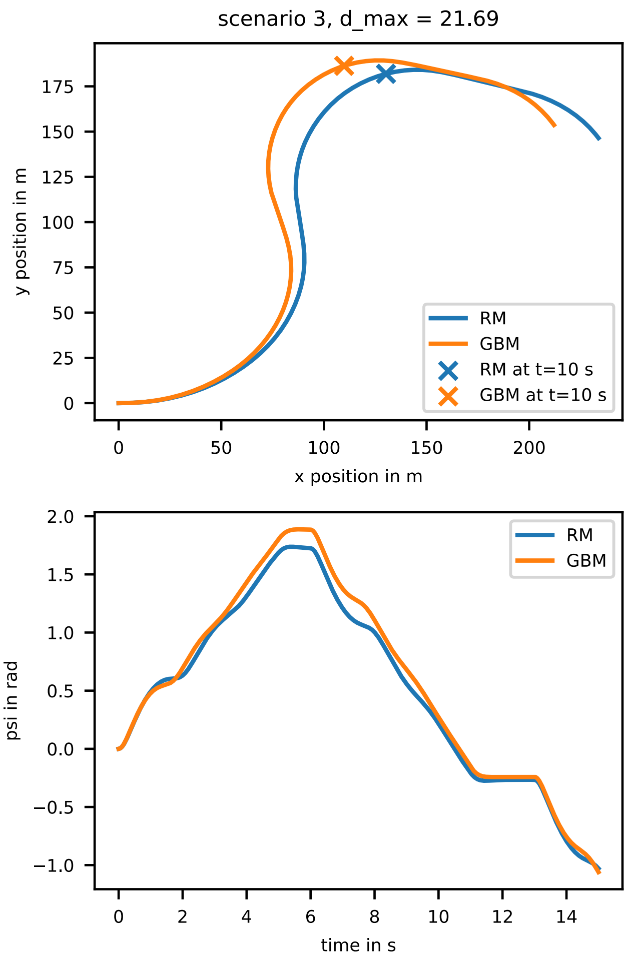

Figure 4. DLC on the training maneuver parametrized by $\boldsymbol{p}^3_{DLC}$

The results are heavily different for the last maneuver considered here in Equation 19.

$$\boldsymbol{p}^4_{DLC} = [\frac{\pi}{16}\text{rad}, 1\text{s}, 1\text{s}, 2\text{s}, 1000\text{Nm}] \tag{19}$$

The trajectories can be seen in Figure 5. Deviations quickly build up even before changing lanes for the second time.

4.3 Discussion

Comparing (projections of) trajectories in state space in an objective manner is non-trivial. For this reason, we will resort to arguing qualitatively. Maneuver $\boldsymbol{p}^1_{DLC}$, $\boldsymbol{p}^2_{DLC}$ and $\boldsymbol{p}^3_{DLC}$ have in common that the tracks of both the RM and the GBM share strong similarities. Deviations become clearly visible as simulation time runs, but the behaviour seems to be comparable. Furthermore, the corresponding trajectories of ψ are encouraging since they look even more similar, although the GBM was never explicitly trained to fit its predictions along that axis of state space.

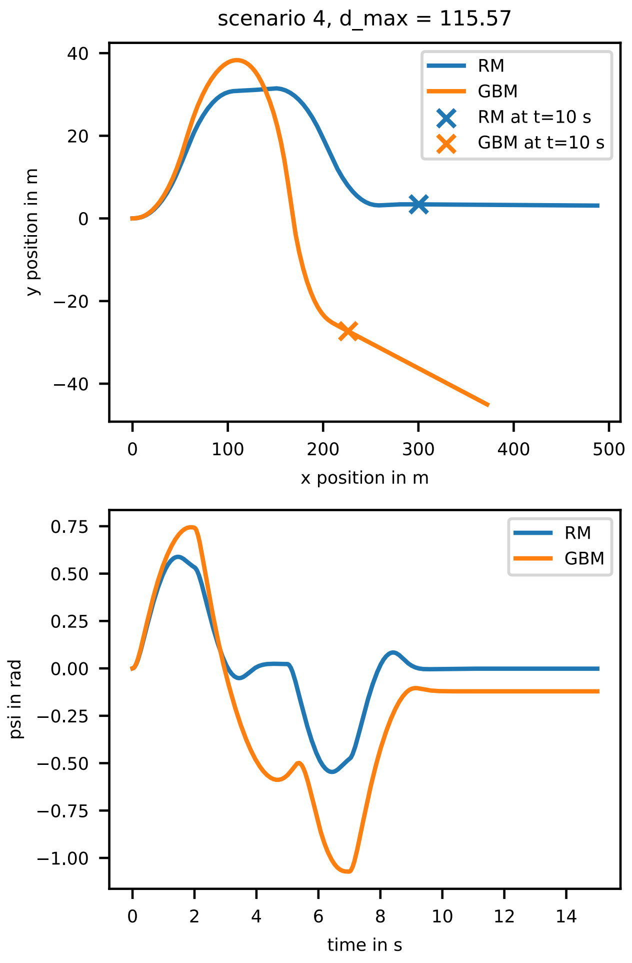

Figure 5. DLC on the training maneuver parametrized by $\boldsymbol{p}^4_{DLC}$

Not so in the case of $\boldsymbol{p}^4_{DLC}$, however. The GBM misses the track of the RM by far. This mismatch is further displayed by the corresponding trajectories of ψ.

The reason for the different results is probably the nature of the differences between the maneuver parameters. The first three maneuver parameter sets only differ in **timing** parameters (T, S, O), not signal level parameters (δ). The ANN is evaluated many times during a single maneuver simulation. Friction quantities μ_{Fx}, μ_{Fy} and μ_{Rx} are not directly influenced by the parameter sets and can vary during the simulations. This is not the case for δ. It is directly passed through to the input layer of the ANN. For this reason, during the last maneuver, the ANN was presented with inputs that it just could not have seen throughout the single training maneuver.

Although differences between GBM predictions and the data are visible, it was shown that GBM has the potential to enhance/complete white box models with very sparse data. Facilities to enhance models in this manner should become features of acausal modeling tools like Modia/TinyModia.

4.4 Remarks

l_{MSE} evaluates how well the GBM follows the track of the RM. Even if it reports a very low loss, can we ever expect our ANN to calculate 'the correct' normal forces? Or can a potentially large space of functions lead to low losses?

In order to answer this question for our trained model, we compared $\boldsymbol{f}_{X_Z}$ and $\hat{\boldsymbol{f}}_{X_Z}$ over the GBM trajectory simulated from maneuver $\boldsymbol{p}^2_{DLC}$. The results are shown in Figure 6. Note that this step cannot usually be done in practice if the underlying system equations are unknown.

As we can see, $\boldsymbol{f}_{X_Z}$ and $\hat{\boldsymbol{f}}_{X_Z}$ do not behave as similarly as we would like them to. Due to Equation 7, the components of $\boldsymbol{f}_{X_Z}$ always add up to a constant value: the reaction force that neutralises the gravitational force acting on the vehicle CoM. When we replaced the set of equations by the ANN, we explicitly neglected this invariance. And as we can see, our optimization technique did not recover said invariance from the single training trajectory.

5 Framework of Tools

5.1 GBM

The model that was described in 2.1 has been declared with the help of TinyModia. The model inputs are passed to the vehicle model using a separate 'Driver' model that sets the inputs.

The actual replacement of Equations 8 to 7 was done by modifying the AST of the model directly before the model instantiation process mentioned in subsection 1.4: $\boldsymbol{f}_{X_Z}$ was replaced by $\hat{\boldsymbol{f}}_{X_Z}$. During said process, the `getDerivatives!` method was written to a file and later modified *by hand* as detailed in the next subsection. With the second feature in 5.3, this process would have been easy to automate as well.

5.2 Training Procedure

As a part of a gradient descent based optimization of $l_{MSE}(\boldsymbol{\theta})$, gradients/sensitivities of $\hat{\boldsymbol{u}}(t)$ w.r.t. $\boldsymbol{\theta}$ are required. For this task, we relied on `Zygote.jl` (Mike Innes et al. 2019) and `DifferentialEquations.jl` (Rackauckas and Nie 2017).

`DifferentialEquations`'s `solve` was used with solver `Tsit5()` and default options to simulate the GBM in order to yield $\hat{\boldsymbol{u}}(t)$. To differentiate $\hat{\boldsymbol{u}}(t)$ at specific points w.r.t. $\boldsymbol{\theta}$, `Zygote` needs information on how to calculate sensitivities of ODE solutions. This gap is filled using package `DiffEqSensitivity.jl`. In the standard setting, which we used, an approach based on adjoint sensitivities is used.

`DifferentialEquations`'s `solve` requires the user to specify an array of parameters with respect to which the ODE solution can then be differentiated.

At the time of writing this, this detail may be inconvenient to users. It is the reason why the results returned from TinyModia's `simulate!` calls can not be differentiated w.r.t. model parameters.

We worked around this issue by further modifying the `getDerivatives!` function mentioned in subsection 5.1. We made $\boldsymbol{\theta}$ a function parameter as well and then called `DifferentialEquations`'s `solve` ourselves to enable differentiation.

5.3 Requested Features in Modia/TinyModia

To streamline the GBM workflow, we would like to see the following features in Modia/TinyModia:

- **Compatibility of `simulate!` with `Zygote.jl`**
  The need for a lot of work currently neccessary in order to generate GBM would be removed if Modia/TinyModia simulation results could be differentiated with respect to model parameters. Since `simulate!` internally uses `DifferentialEquations`'s `solve` that generally offers this functionality, we believe that the main work to be done here resides in the creation of a serialized version of all model parameters (i.e. in a single Array passed to `solve`).

- **The ability to obtain the AST of `getDerivatives!` conveniently after instantiation**
  It should not be neccessary to print the code of `getDerivatives!` to file from a debug log. Instead, the AST should be returned as an expression if the user requests this. This feature would be very handy for the more experimental case in which the user does not know beforehand where to best place ANN.

- **The ability to mark equations for later replacement by ANN-enhanced equations**
  We modified assignments manually with ANN in the generated simulation code.
  Normally, users of software intended for acausal modeling and subsequent simulation do not interact with intermediate representations of the model equations or the generated simulation code. They may have a slight idea of which of the model components in their model is "faulty" and leads to errors observed when simulation results are compared to real-world measurements. Take air drag as an example. Drag is a complex phenomenon and it is unlikely that a model as simple as ours in Equation 4 is sufficient to make good predictions in a real-world scenario with very low and high vehicle velocities. We are certain about which are the "faulty" model equations but we would not know how to improve them without extensive knowledge about aerodynamics.

  However in order to insert ANN into those "faulty" equations like we did in this paper, a user would have to modify the simulation code by hand like we did. They may have trouble finding the exact lines of code produced from their "faulty" component equations.

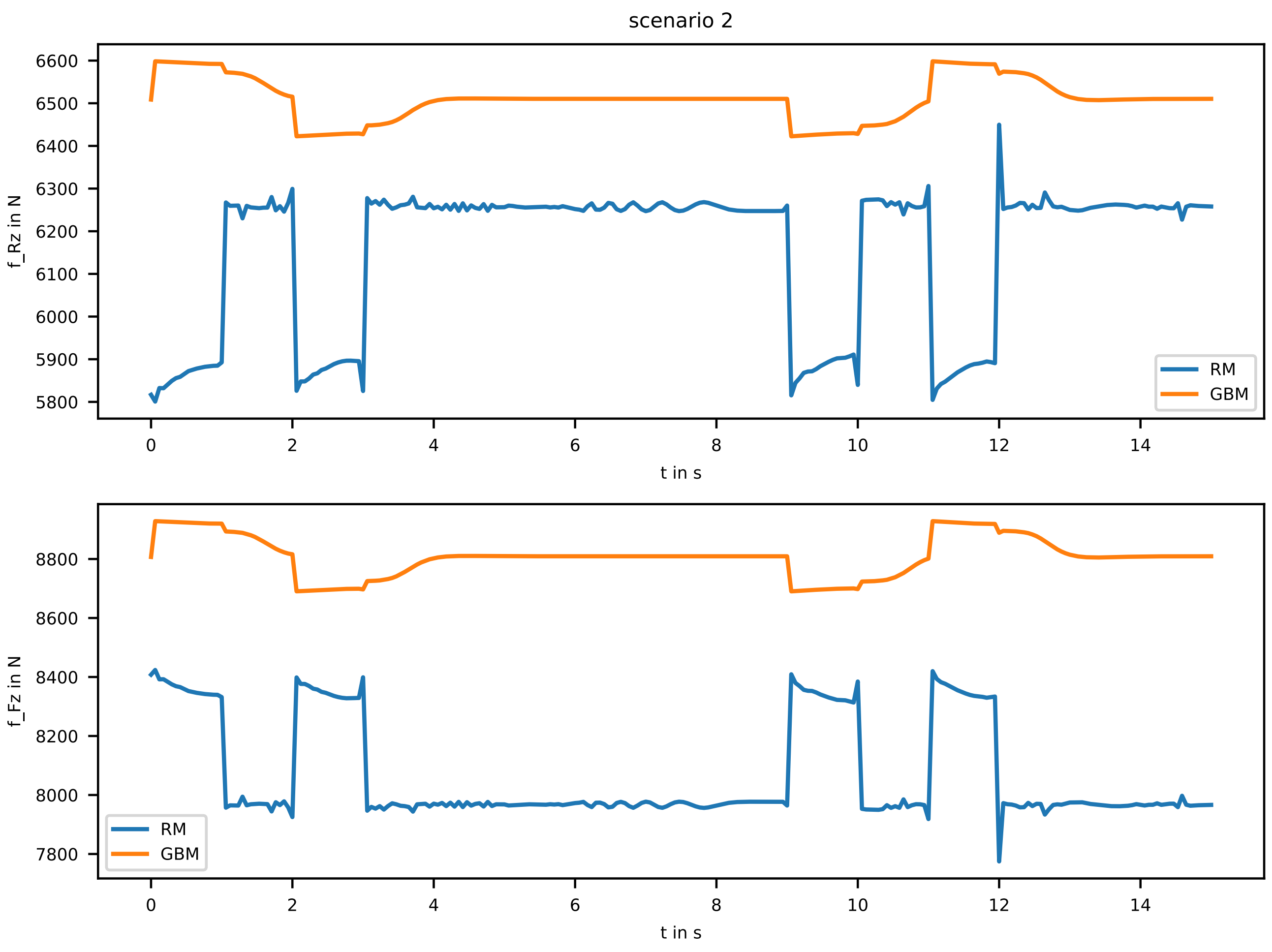

Figure 6. Comparison of the normal forces. The RM internally uses $\boldsymbol{f}_{X_z}$ to calculate normal forces f_{Rz} and f_{Fz}, the GBM uses $\hat{\boldsymbol{f}}_{Xz}$.

Even then they are restricted to the causalization created by the model transformation algorithm. After all, the user might want to include new variables in those equations.

We would like to have a feature that lets the user mark equations they are uncertain about with a tag and possibly additional desired involved model variables during the modeling process. The causalization algorithm would then work as usual and determine a sequence of calculation and possibly algebraic loops whilst keeping track of the user-provided equation tags. The user would then be informed about where and how their equations are used in the final calculation graph. Whether they are part of an algebraic loop, what their (or the corresponding loop's) inputs are and which model variable they were matched to.

With this information, the user can create an ANN with the right input/output size. The user would be very free to design what happens in the ANN: It could just be a densely connected layers with appropriate activation functions. Or it could take the original "faulty" equation as a basis and just add such a network so that the latter only has to learn a difference.

After the user has provided an ANN for each of the marked equations, a code generating algorithm would produce the actual simulation code for the model. This then leaves the user with an ANN-infused model that can be trained on an arbitrary data set.

6 Future Research

We are planning to experiment with different training schemes that operate on a set of trajectories instead of just a single training maneuvers.

Moreover, it may be beneficial to alter the way we build time horizons over which we compare trajectories. Instead of growing a single horizon to full length, one could slice a single trajectory into several segments and calculate gradients in parallel.

Another issue that needs to be addressed is the normalisation/transformation of the inputs that reach the ANN parts of the model. Some neural network architectures/activation functions were designed assuming that inputs to the input layer follow specific distributions. We did not account for this yet because it is an inherent property of our model structure. Before we simulate our model, we do not yet know what the inputs to the ANN will be. After all, they depend on the solution to the ODE. It should be possible to construct an algorithm that adapts a prescaler to keep inputs over training trajectories inside specific bounds. Future research will have to show whether this is feasible or whether this situation can be solved differently e.g. by making use of different activation functions.

For our loss function, we relied on a very simple MSE-based formulation. Other (differentiable) metrics such as soft dynamic time warping (Cuturi and Blondel 2017) exist in order to compare time series data. We will evaluate whether these alternative metrics are beneficial to our training results.

Furthermore, we will examine whether invertible neural networks as in (Ardizzone et al. 2019) can help overcome the issue of ANN ending up in inefficient root-finding loops.

7 Conclusion

We demonstrated that GBM can be trained with very sparse data to yield remarkable results. Furthermore, we detailed how to achieve this with free software and made suggestions for features that would make this process a lot simpler for the user.

References

Ardizzone, Lynton et al. (2019). *Analyzing Inverse Problems with Invertible Neural Networks*. arXiv: 1808.04730 [cs.LG].

Bezanson, Jeff et al. (2012). “Julia: A fast dynamic language for technical computing”. In: *arXiv preprint arXiv:1209.5145*.

Cuturi, Marco and Mathieu Blondel (2017). “Soft-dtw: a differentiable loss function for time-series”. In: *International Conference on Machine Learning*. PMLR, pp. 894–903.

Elmqvist, Hilding and Martin Otter (2017). “Innovations for future Modelica”. In: *Proceedings of 12th International Modelica Conference*. Linköping University Electronic Press.

Elmqvist, Hilding and Martin Otter (2021). *TinyModia*. https://github.com/ModiaSim/TinyModia.jl. Accessed: 2021-05-07.

Innes, Michael et al. (2018). “Fashionable Modelling with Flux”. In: *CoRR* abs/1811.01457. arXiv: 1811.01457. URL: https://arxiv.org/abs/1811.01457.

Innes, Mike et al. (2019). “A differentiable programming system to bridge machine learning and scientific computing”. In: *arXiv preprint arXiv:1907.07587*.

Karpatne, Anuj et al. (2018). *Physics-guided Neural Networks (PGNN): An Application in Lake Temperature Modeling*. arXiv: 1710.11431 [cs.LG].

Ma, Yingbo et al. (2021). *ModelingToolkit: A Composable Graph Transformation System For Equation-Based Modeling*. arXiv: 2103.05244 [cs.MS].

Pacejka, Hans (2005). *Tire and vehicle dynamics*. Elsevier.

Rackauckas, Christopher, Yingbo Ma, et al. (2020). *Universal Differential Equations for Scientific Machine Learning*. arXiv: 2001.04385 [cs.LG].

Rackauckas, Christopher and Qing Nie (2017). “Differentialequations.jl–a performant and feature-rich ecosystem for solving differential equations in julia”. In: *Journal of Open Research Software* 5.1.

Rai, R. and C. K. Sahu (2020). “Driven by Data or Derived Through Physics? A Review of Hybrid Physics Guided Machine Learning Techniques With Cyber-Physical System (CPS) Focus”. In: *IEEE Access* 8, pp. 71050–71073.

Velenis, Efstathios, Emilio Frazzoli, and Panagiotis Tsiotras (2009). “On steady-state cornering equilibria for wheeled vehicles with drift”. In: *Proceedings of the 48h IEEE Conference on Decision and Control (CDC) held jointly with 2009 28th Chinese Control Conference*. IEEE, pp. 3545–3550.

Willard, Jared et al. (2020). *Integrating Physics-Based Modeling with Machine Learning: A Survey*. arXiv: 2003.04919 [physics.comp-ph].