Enabling Adaptive Methods for Type One Diabetes Models

Ian Ballinger
Nonlinear Control Final Report, Spring 2025
Massachusetts Institute of Technology
Mechanical Engineering | Traverso Lab

***Abstract*—This paper will prescribe procedures for transpiling high-dimensional sets of differential equations (on the order of 100 state variables), written for human academic audiences, into code compatible with modern nonlinear Continuous Time (CT) solvers. Additionally, this paper will cover the steps necessary for implementing an adaptive controller for these kinds of high dimensional systems.**

***Index Terms*—Control Systems, Continuous Glucose Monitors, Nonlinear Estimation, Nonlinear Control, Artifical Pancreas Systems, Glycemic Control, Diabetes Control, Personalized Medicine**

I. Introduction

The field of personalized medicine is poised to revolutionize healthcare as we know it, and nonlinear adaptive algorithms will be an indispensable part of that revolution. It is common knowledge that the Internet of Things is still in its infancy, with FDA approvals for integrated medical devices lagging even further, yet there already exist communities dedicated to "DIY" closed-loop control of their chronic illnesses [1]. In the cited example (the OpenAPS ecosystem), closed loop "Artificial Pancreas" systems have already completed on the order of 100-million deployment-hours with a vanishingly small market share compared to the total population of Type-one Diabetes (T1D) patients. Adaptive algorithms have yet to emerge in this space due to their complexity and the need for each individual to take responsibility for their own implementation detail, again on account of the lagging FDA clearances for commercial products [2]. This paper aims to introduce a path towards adaptive control strategies for personalized medicine applications in this space. This paper will not present the critical details of inter-operating within the OpenAPS ecosystem in the interest of patient safety.

II. Computational Diabetes Models and Design Safety

The "University of Virginia / Padova (UVA/Padova) models" of T1D are FDA approved for pre-clinical control system validation [3]–[5]. This means these models are suitable for conducting control system design safety tests before human testing begins, and therefore these models are suitable to be used as the best guess of known-but-uncertain dynamics in an adaptive control algorithm. The overarching structure of the UVA/Padova models can be seen in **Fig. 1** . The model contains 28 different subsystems, each representing different biological processes in the human glucose homeostatic regulation system, and comprises approximately 86 differential equations, in addition to other modeling overhead.

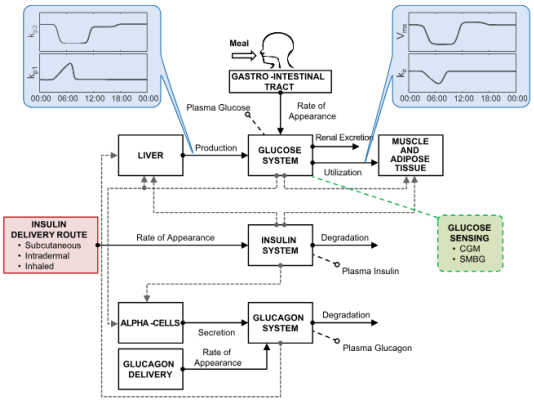

Fig. 1: The UVA/Padova S2017 model structure, from [3]

The strategy for developing a control system for this system is simple, but labor intensive. First, and most laboriously, the model must be converted to a form compatible with computational modeling tools. Afterwards, the translated code must be algorithmically decomposed to acquire the matrices and variables necessary for adaptive control implementations, namely $Y(\dot{x}(t), x(t))$.

ModelingToolkit.jl (MTK) was chosen as the computational modeling tooling on account of it's native compatibility with many nonlinear systems and control libraries in the Julia language ecosystem [6], [7]. MTK is a mixed symbolic and numerical modeling ecosystem, so it is feasible to mix systems of differing implementations in the future, for example in the case of testing discretized control laws in future work.

A. Transpiling Differential Equaitons into Julia Code

The procedure to transpile large systems of human readable differential equations to Julia is elaborated below, with examples listed per step. You may also view a full implementation of the UVA/Padova model as it relates to adaptive algorithms in the Appendices. Let us begin with a sketch of the overarching program structure:

```
## 1: dependency management
    ...
## 2: Variable and Parameter Pre-declaration
    ...
## 3: Derivative equations
    ...
## 4: Model Compilation
    ...
## 5: Using the model
    ...
```

1) Dependencies and Setup Operations: The following is a minimal set of boilerplate to get started with model transpilation in support of adaptive control implementations.

```
using ModelingToolkit
using DifferentialEquations #MTK implementation detail
using NonlinearSolve #MTK implementation detail
using LinearAlgebra
using Random
using DataInterpolations #for feeding models with
    timeseries data
using Symbolics
using SymbolicUtils
using Plots

# This is a "Nothing-up-my-sleeve number"
ESTIMATE_UNKNOWN = 0.5 #chosen to be a small, nonzero,
    positive number
#note: a choice of 0.1 results in numerical instability
    for solutions to the virtual system this is used to
    initialize. 0.5 seems to be a good choice.

@independent_variables t
D = Differential(t)


pmap = [] #parameter map, which may be recursive
x_eqns = []
x = []
x0 = []
p_tv = [] #specifically for time varying parameters
p_tv0 = []
p0 = []
p = [] #for all constant parameters
```

2) variable predeclarations: When programming, variables must generally be declared before they are used (in this case, when writing the differential equations of many connected subsystems). There are generally two cases of variables here, being differential variables with equality constraints on their derivatives, and "helper" variables which have no such derivative constraints. It is possible to define symbolic handles for these types in the same way, but it is recommended you handle them differently as in the following example subsystem.

```
#glucose rate of appearance
@variables Q_sto(t) Q_sto1(t) Q_sto2(t) Q_gut(t)
@parameters f BW k_max k_min k_abs c α β Q_sto0
pmap = vcat(pmap, [
    f      => 0.9   #units unknown
    BW     => 100.0 #kg
    k_max  => 0.0558#units unknown
    k_min  => 0.0080#units unknown
    k_abs  => 0.057 #units unknown
    c      => 0.010 #units unknown
    α      => 0.82  #units unknown
    β      => 0.82  #units unknown
    Q_sto0 => ESTIMATE_UNKNOWN ]) #units unknown

# gastric emptying rate, clinically significant
@inline K_EMPT_IMPL(Q_STO, DOSE_) = (
    k_min + (((k_max - k_min) / 2.0) *
        (tanh(α * (Q_STO - β * DOSE_))
        - tanh(β *(Q_STO - c * DOSE_))))
) #names mangled to ensure no scoping errors
@inline RA_MEAL_IMPL(Q_GUT) = (
    (f * k_abs * Q_GUT) / BW
) #names mangled to ensure no scoping errors
x  = vcat(x,  [ Q_sto, Q_sto1, Q_sto2, Q_gut])
x0 = vcat(x0, [Q_sto0,    0.0,    0.0,   0.0])
a_true=vcat(a_true, [k_max, k_min, k_abs, c, α, β, Q_sto0
    ]);
```

Compare this to **Fig. 2**, being the human-readable differential equations for this subsystem from the UVA/Padova publication (version "S2017") [3]. Observe that in this pre-declaration stage, no derivative equations are yet defined, and initial conditions for all variables are declared. Parameter values are also initialized regardless of if they are known. In the case where a quantity of interest is not a differential variable, it is declared as an inline function so the dimensionality of the system of equations may be explicitly reduced where possible. These quantities may use both static and time varying parameters as usual. A final note is that this style of pre-declaration preserves locality of reference, supporting code maintainability, even when the systems in question are large.

Glucose Rate of Appearance

$$\begin{cases} Q_{sto}(t) = Q_{sto1}(t) + Q_{sto2}(t) \\ Q_{sto}(0) = 0 \\ \dot{Q}_{sto1}(t) = -k_{max} \cdot Q_{sto1}(t) + Dose \cdot \delta(t) \\ Q_{sto1}(0) = 0 \\ \dot{Q}_{sto2}(t) = -k_{empt}(Q_{sto}) \cdot Q_{sto2}(t) + k_{max} \cdot Q_{sto1}(t) \\ Q_{sto2}(0) = 0 \\ \dot{Q}_{gut}(t) = -k_{abs} \cdot Q_{gut}(t) + k_{empt}(Q_{sto}) \cdot Q_{sto2}(t) \\ Q_{gut}(0) = 0 \\ Ra_{meal}(t) = \dfrac{f \cdot k_{abs} \cdot Q_{gut}(t)}{BW} \\ Ra_{meal}(0) = 0 \end{cases}$$

with

$$k_{empt}(Q_{sto}) = k_{min} + \frac{k_{max} - k_{min}}{2} \cdot \left\{ \tanh\left[\alpha\left(Q_{sto} - \beta \cdot Dose\right)\right] - \tanh\left[\beta\left(Q_{sto} - c \cdot Dose\right)\right] + 2 \right\}$$

Fig. 2: The Glucose rate of Appearance subsystem in differential equation form. Note the presence of auxiliary helper functions [$Ra_{meal}(..)$, $k_{empt}(..)$] without derivative constraints.

3) Differential Equations and Model Compilation: Now that each of the variables are initialized, the model dynamics may be stated. It is at this stage that constraints on system states may be enforced as well, for example the concentration of blood glucose and other metabolytes is always greater than zero.

```
@inline EQ_CLAMP_D_VGTE0(variable, deriv_expr) = (
    #the >= is important so if we hit zero we can still
         move away instead of getting trapped.
    D(variable) ~ ifelse(variable >= 0.0, deriv_expr, 0.0
        )
)

x_eqns = [
    #glucose subsystem
    EQ_CLAMP_D_VGTE0(G_p,
        EGP_IMPL(k_p1, G_p, k_p3, X_L, X_H) +
             RA_MEAL_IMPL(Q_gut)
        - U_ii - E - (k_1 * G_p) + (k_2 * G_t)
    )
    EQ_CLAMP_D_VGTE0(G_t,
        (-1.0 * U_id) + (k_1 * G_p) - (k_2 * G_t)
    )
    G ~ (G_p / V_g)
    ... #other subsystem equations
]

... #other system equation blocks if needed

control_eqns = [ #for now, no control. run the system
     open loop to test.
    EQ_CLAMP_D_VGTE0(Dose,
        ifelse(t<1.0, 0.0,
        ifelse(t<5, 1000.0,
        ifelse(t<9, -1000,
        0.0
    ))))
    D(u_sc) ~ 0.0
    EQ_CLAMP_D_VGTE0(u_ih,
        ifelse(t<13.0, 0.0,
        ifelse(t<15, 20.0,
        -1000 #will be clamped
    )))
    D(u_id) ~ 0.0
]


model_eqns = mapreduce(x -> x, vcat, [x_eqns, other_eqns]
    )
u0 = [(x[i] => x0[i]) for (i,item) in enumerate(x)]
ps = [item.first for (i,item) in enumerate(values(pmap))]

@named model = ODESystem(model_eqns, t, x, ps,
    defaults = Dict(vcat(u0, pmap)))
complete_model = structural_simplify(model, io = (u, y))

prob = ODEProblem(complete_model, symbolic_u0 = true,
    jac = true, sparse = true)

#using adaptive time-stepping by default
sol = solve(prob, tspan = [0.0, 120.0]);
```

At this stage we can run the model. A collection of system traces can be seen in **Fig. 3**

4) Future Work on the Diabetes Model Implementation: Note that in **Fig. 3** the model appears to be erroneous. More work is needed to rectify this, and this issue likely stems from poor parameter assumptions for the virtual system under test. In addition, the implementation is not yet composable, nor does it support unit testing. This can be fixed by wrapping the implementation in a manner consistent with the documentation [8].

B. Adaptive Controller Implementation

Assuming the shortcomings of the model are rectified, an adaptive controller for this system would be simple to implement using MTK (a reference implementation of adaptive control with MTK is included in the appendices). Finding the equilibrium point for the model so that it may be represented with it's jacobian form $\dot{\vec{x}} = \mathbf{F}(\vec{x}, t) = \mathbf{J}(\vec{x}, t) \cdot \vec{x}$ is simple.

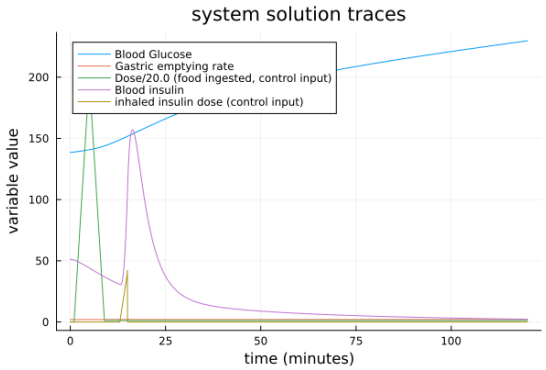

Fig. 3: The simulated trajectory of the system. Note how blood insulin does not respond to food intake, which is representative of diabetes. Blood glucose is also not responsive to administered insulin, which indicates an error in the model implementation.

```
x_simplified = unknowns(complete_model)
p_simplified = parameters(complete_model)
eq_simplified_rhs = [eq.rhs for eq in equations(
     complete_model)];
sym_jacobian = Symbolics.jacobian(eq_simplified_rhs,
     x_simplified)
# display(sym_jacobian)

function f_simplified(u, p, t)
    varmap = vcat(
        pmap,
        [(item => u[i]) for (i, item) in enumerate(
             x_simplified) ]
    )
    return simplify(Symbolics.substitute(
         eq_simplified_rhs, varmap))
end

#don't pass parameters, substitution handled internally
     by f
rootfinding_problem = NonlinearProblem(f_simplified, prob
     .u0)
sol_roots = solve(rootfinding_problem)
#(85, 85)
#retcode: Success
# u: 85-element Vector{Float64}:
#  260.49
#  205.57
#    2.56
#    2.61 ... 0.009

sym_jacobian = Symbolics.jacobian(eq_simplified_rhs,
     x_simplified)
display(size(sym_jacobian)) #(85, 85)

parameter_jacobian = Symbolics.jacobian(eq_simplified_rhs
      .- sol_roots.u, p_simplified)
#subtract the eq point, and derivs will = 0 @
     x_simplified = 0
#display(size(sym_jacobian)) #(85, 68)
```

From here, the a matrix representation of the linearized dynamics with respect to the parameters may be computed, and adaptive algorithms would follow. This numerical method is used instead of a symbolic method on account of the underdeveloped symbolic nonlinear solvers ecosystem. This is still useful if the dynamics of the system are known exactly, but not so much if the system parameters are unknown. Regardless, the numerical solution only takes 0.2s to compute on a $AMDRyzen97940HS$ CPU, so in the case of slow biological systems this may still be suitable for real

time utilization. In the future, after working to improve the model parameter estimates and further vet its implementation, the algorithms described here will be implemented into real embedded systems.

III. Conclusions and Future Work

This paper has presented a method for transpiling high dimensional systems of equations into a computational modeling engine suitable for nonlinear control algorithm development. These systems have then been analyzed to show that adaptive control implementations are simple to implement, and a reference implementation of such an algorithm for a lower dimensional example system was prescribed. Finally, a method to produce the variables needed for adaptive algorithms has been shown, which includes solving for equilibrium points and taking the symbolic jacobian of the structurally simplified model.

IV. Acknowledgements

Thank you so much to Professor Slotine and Dr. Nah for your hard work teaching all of us this semester. You have my heartfelt thanks and enduring admiration.

References

[1] "OpenAPS Outcomes – OpenAPS.org." [Online]. Available: https://OpenAPS.org/outcomes/

[2] "OpenAPS.org – #WeAreNotWaiting to reduce the burden of Type 1 diabetes." [Online]. Available: https://openaps.org/#section2

[3] R. Visentin, E. Campos-Náñez, M. Schiavon, D. Lv, M. Vettoretti, M. Breton, B. P. Kovatchev, C. Dalla Man, and C. Cobelli, "The UVA/Padova Type 1 Diabetes Simulator Goes From Single Meal to Single Day," *Journal of Diabetes Science and Technology*, vol. 12, no. 2, pp. 273–281, Feb. 2018. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5851236/

[4] C. Dalla Man, F. Micheletto, D. Lv, M. Breton, B. Kovatchev, and C. Cobelli, "The UVA/PADOVA type 1 diabetes simulator: New features," *Journal of diabetes science and technology*, vol. 8, pp. 26–34, May 2014.

[5] B. P. Kovatchev, M. Breton, C. D. Man, and C. Cobelli, "In Silico Preclinical Trials: A Proof of Concept in Closed-Loop Control of Type 1 Diabetes," *Journal of diabetes science and technology (Online)*, vol. 3, no. 1, pp. 44–55, Jan. 2009. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2681269/

[6] Y. Ma, S. Gowda, R. Anantharaman, C. Laughman, V. Shah, and C. Rackauckas, "Modelingtoolkit: A composable graph transformation system for equation-based modeling," 2021.

[7] "Home · ModelingToolkit.jl." [Online]. Available: https://docs.sciml.ai/ModelingToolkit/stable/

[8] "Composing Models and Building Reusable Components · ModelingToolkit.jl." [Online]. Available: https://docs.sciml.ai/ModelingToolkit/stable/basics/Composition/

Appendix: Full Implementation Detail

https://github.com/IanBallinger/nonlinear-final