Systems & Control Transactions

Original Research Article
Peer Reviewed Conference Proceeding
ESCAPE 36 - European Symposium on Computer Aided Process Engineering
Sheffield, UK. 21-24 June 2026

PSE PRESS

GlycoPy: An Equation-Oriented and Object-Oriented Python Framework for Process Modeling, Optimization and Optimal Control

Yingjie Ma[ab*], Jing Guo[bc], and Richard D. Braatz[a]

[a] Nanjing University, School of Robotics and Automation, Suzhou, Jiangsu Province, China
[b] Massachusetts Institute of Technology, Department of Chemical Engineering, Cambridge, MA, USA
[c] Polytechnique Montréal, Department of Chemical Engineering, Montréal, QC, Canada
* Corresponding Author: yingjie.ma@nju.edu.cn.

ABSTRACT

Nonlinear model predictive control (NMPC) can substantially improve performance and constraint handling for (bio)chemical processes, but its adoption is still limited by the effort required to build maintainable first-principles models and to implement efficient dynamic optimization-based controllers. This paper presents GlycoPy, an open-source, equation-oriented and object-oriented Python framework that supports hierarchical model construction and integrated workflows for simulation, parameter estimation, dynamic optimization, and NMPC. The case study of the monoclonal antibody glycosylation process based on a multiscale model demonstrates the capability of GlycoPy.

Keywords: Nonlinear Model Predictive Control, Dynamic Modelling, Optimization, Simulation, Multiscale Modelling, software

1. INTRODUCTION

Equation-oriented (EO) environments are well suited to modeling and optimization of (bio)chemical processes: they let researchers develop new models without implementing solvers, and they provide highly accurate derivatives through automatic differentiation [1]. Yet, given the complexity of real-world systems, existing EO software often struggles to balance user-friendliness, developer productivity, flexibility, and extensibility [2]. Commercial EO tools (e.g., gPROMS [3]) provide rich modeling capabilities but are not designed for tight integration with advanced nonlinear model predictive control (NMPC) workflows or for rapid prototyping of bespoke algorithms. Open-source packages such as GEKKO [4], do-mpc [5] and SolACE [6] offer convenient optimization and NMPC, yet lack native support for hierarchical model composition, which is important for multiscale and multi-unit systems. Pyomo supports hierarchical modeling and many advanced process systems engineering (PSE) tasks [7]. However, it is difficult for its users to develop and integrate novel simulation algorithms, which are often essential for optimization workflows driven by computationally expensive simulations.

We therefore introduce GlycoPy, an equation-oriented Python framework built on CasADi to address these gaps. CasADi provides the symbolic infrastructure for efficient numerical optimal control, although it is not itself a domain-specific language for large process models and control [8]. GlycoPy emphasizes (1) object-oriented, hierarchical modeling to ease development and improve user-friendliness and maintainability; (2) advanced PSE functionalities, including parameter estimation, dynamic optimization, and NMPC with and without parameter adaptation; (3) control-vector parameterization (CVP) approach for dynamic optimization, which is typically more robust for large and strongly nonlinear model-constrained problems; and (4) direct access to CasADi symbolic variables and equations, enabling rapid prototyping of customized algorithms. GlycoPy is released as open source.

In the remainder of this paper, we first present the software and then validate its NMPC capability by controlling monoclonal antibody (mAb) glycosylation using a multiscale model.

2. SOFTWARE DESCRIPTION

GlycoPy is built on CasADi together with NumPy/Pandas/SciPy/Matplotlib. CasADi provides symbolic variables and equations and enables efficient differential algebraic equation (DAE) and ordinary differential equation (ODE) simulations and automatic differentiation, which are essential for large-scale dynamic optimization. Compared to native CasADi, GlycoPy provides a structured framework that streamlines model development and algorithm implementation. The following sections provide more details on how GlycoPy is used for modeling, simulation, parameter estimation, dynamic optimization, and NMPC.

2.1 Nonlinear modeling

In GlycoPy, a model is defined by implementing the `build()` method of the Model class using two basic objects: Variable and Equation. GlycoPy supports three variable types: differential states x, algebraic variables z, and parameters p (time-invariant or time-varying but fixed during integration). Variables are created via add_var, which returns a GlycoPy symbol (a CasADi symbol or a dictionary of CasADi symbols). Users then write equations directly using standard CasADi-style operators and register them with `add_eq` as either ODE (`ode`) or algebraic (`alg`) equations.

To illustrate, we consider a small nonlinear DAE benchmark adapted from Andersson et al. (2019).

$$\begin{cases} \frac{dx_1(t)}{dt} = z(t)\,x_1(t) - x_2(t) + u(t) \\ \frac{dx_2(t)}{dt} = x_1(t) \\ \frac{d\mathrm{obj}}{dt} = x_1(t)^2 + x_2(t)^2 + u^2 \\ z(t) = p - x_2(t)^2 \end{cases} \quad t \in [0, T] \tag{1}$$

$$x_1(0) = 0, x_2(0) = 1, \mathrm{obj}(0) = 0, p = 1, \tag{2}$$

where $x \in \mathbb{R}^2$ denotes the differential variables, $z \in \mathbb{R}$ is the algebraic variable, and $\mathrm{obj} \in \mathbb{R}$ is the accumulated term to be used as the objective function in optimization. $u \in \mathbb{R}$ is the control action, and $T = 10$ is the considered time horizon. The GlycoPy implementation for the problem is shown below.

```
# Define model
Class SimpleDAE(Model):
 def build(self):
  xg = self.add_var('x', 'xg', m=2, val=[0., 1.])
  obj = self.add_var('x', 'obj', val=0.)
  zg = self.add_var('z', 'zg')
  pg = self.add_var('p', 'pg', val=1.0)
  ug = self.add_var('u', 'ug', val=0.)
  self.add_eq('ode', 'xg', [zg * xg[0] - xg[1] + ug, xg[0]])
  self.add_eq('ode', 'obj', xg[0]**2 + xg[1]**2 + ug**2)
  self.add_eq('alg', 'zg', zg - (1 - xg[1]**2) * pg)
```

Because each GlycoPy model is a Python class, users can naturally apply object-oriented design (inheritance and composition) and construct hierarchical models by assembling submodels. This modular structure improves development, testing, and maintenance for large chemical/bioprocess systems.

2.2 Simulation

For practical process models, time-varying inputs/parameters (e.g., piecewise-constant feed profiles) make a native CasADi workflow using built-in integrator cumbersome, especially when some input profiles are fixed while others are to be optimized. GlycoPy addresses this with the `SimulationDyn` class. With that, the code for the simulation of above DAE example is:

```
# Define simulator
initializer = Initializer('init', model)
integrator = Integrator('integrator', 'idas', dae_prob, initializer=initializer)
sim = SimulationDyn(model, integrator, p_var=[('simple_dae.pg', 0), ])
p_var = np.random.randn(n_points-1, 1)
# Simulate
sim.simulation_with_events_numerical(t_event=time_points, p_var=p_var)
```

For numerical simulation, users specify which parameters are time-varying and provide their values at event times. The method `simulation_with_events_numerical(t_event, p_var)` runs the simulation and stores all results back into the model for convenient plotting and post-processing.

For sensitivity analysis and optimization, numerical trajectories are insufficient because automatic differentiation and symbolic Jacobians require symbolic results. Therefore, GlycoPy also provides `simulation_with_events`, which accepts symbolic input profiles and stores the resulting trajectories as CasADi expressions. Users can then directly build CasADi functions and compute Jacobians (or other derivatives) using standard CasADi operations. To further simplify robust derivative computation, GlycoPy includes `DerivativeEvaluatorSimple`, which wraps sensitivity evaluation with explicit method selection (`forward` or `adjoint`) and a safeguard: if sensitivity equations fail (a common issue for stiff, large-scale DAEs during optimization), it automatically falls back to finite differences. This design improves usability and stability for optimization workflows. The code for derivative computation is:

```
# Define derivative evaluator
evaluator = DerivativeEvaluatorSimple(y,
        p_var_sym,
  gradient_method=GradientMethod.forward)
# Evaluator derivatives
y_val = evaluator.val_fun_py(p_var)
jac_val = evaluator.jac_fun_py(p_var)
```

Finally, symbolic simulation enables developing differentiable simulation algorithms that are difficult to implement in many equation-oriented tools, such as the quasi-steady-state algorithm as will be shown later on in the case study section.

2.3 Dynamic optimization

A standard dynamic optimization problem (optimal control problem, OCP) shown below minimizes an integral (stage cost) plus a terminal cost subject to DAE dynamics, initial/terminal conditions, and bounds on states and inputs: differential states $x(t)$, algebraic states $z(t)$, control inputs $u(t)$, and parameters p.

$$\min_{x(\cdot),z(\cdot),u(\cdot)} \int_0^T L(x(t),z(t),u(t),p)dt + E(x(T),z(T),p)$$

$$\text{s.t.} \quad \left.\begin{aligned} \dot{x}(t) &= \text{ode}(x(t),z(t),u(t),p), \\ 0 &= \text{alg}(x(t),z(t),u(t),p), \\ u(t) &\in \mathcal{U}, x(t) \in \mathcal{X}, z(t) \in \mathcal{Z}, \end{aligned}\right\}, t \in [0,T]$$

$$x(0) \in \mathcal{X}_0, x(T) \in \mathcal{X}_T, z(0) \in \mathcal{Z}_0, z(T) \in \mathcal{Z}_T$$

Here, $\mathcal{U}, \mathcal{X}, \mathcal{Z}$ are given sets that the corresponding variables belong to. To solve the OCP problem numerically, the continuous-time problem is transcribed into a finite-dimensional nonlinear programming (NLP) problem:

$$\min_w f(w)$$

$$\text{s.t. } g_{lb} \le g(w) \le g_{ub},$$

$$w_{lb} \le w \le w_{ub},$$

where w collects decision variables (e.g., discretized controls and possibly some initial states), and $g(w)$ includes the discretized dynamics and path/terminal constraints.

To solve the OCP problem numerically, the continuous-time problem is transcribed into a finite-dimensional NLP. GlycoPy currently uses the CVP method because of its robust convergence for highly nonlinear dynamics-constrained problems. CVP-based optimization repeatedly performs DAE simulations and (for derivative-based solvers) sensitivity/derivative evaluations across iterations.

Building on GlycoPy's differentiable simulation framework, the `MPCTool` class automates the transcription from the DAE-based optimal control problem to the NLP form. Users specify: the control grid (time points), the time-varying variables to be optimized (e.g., inputs), the objective (stage/terminal terms), and constraints (e.g., path bounds on states). `MPCTool.setup()` then generates all NLP ingredients (w, f, g) and their bounds, which are passed to the `Optimization` module through a CasADi-like interface. For usability, `MPCTool` also provides a snapshot/rewind mechanism to store and restore the full model states before optimization. After solving, GlycoPy returns decision variables and key trajectories (controls and states) in convenient DataFrame structures for plotting and further analysis.

2.4 Parameter estimation

Parameter estimation (PE) identifies model parameters (and optionally uncertain initial conditions) by matching model predictions to measured data. For a DAE model, PE can be written as an optimization problem over

$$p = \begin{bmatrix} p_M \\ x(0) \end{bmatrix},$$

subject to the DAE dynamics and measurement equations. After transcription, PE becomes an NLP with decision variables p.

GlycoPy provides a modular PE workflow as shown in Fig. 1.

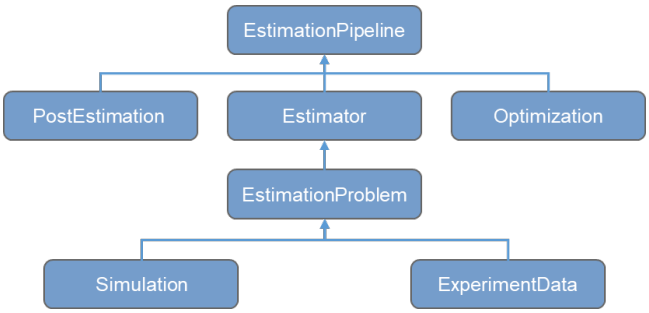

Figure 1. The structure of the parameter estiamtion module in GlycoPy.

The interface class `EstimationPipeline` defines and solves estimation tasks. Experimental data are stored in `ExperimentData`, including time-invariant operating parameters, time-varying operating profiles, measurements y, and their standard deviations σ_y. Multiple experiments are supported by creating multiple `ExperimentData` objects. For each experiment, GlycoPy runs a symbolic simulation to generate predicted responses $\hat{y}$, which enables efficient derivative-based optimization. An `EstimationProblem` object constructs residuals $r = y - \hat{y}$, stores measurement covariance Σ_y, parameter initialization and bounds (p_0, p_{lb}, p_{ub}), optional constraints, and priors $(\hat{p}, \Sigma_{\hat{p}})$ (when Bayesian estimation is used). An `Estimator` then builds the objective function according to the selected criterion:

$$\text{OLS: } \frac{1}{2} r^T r, \tag{3}$$

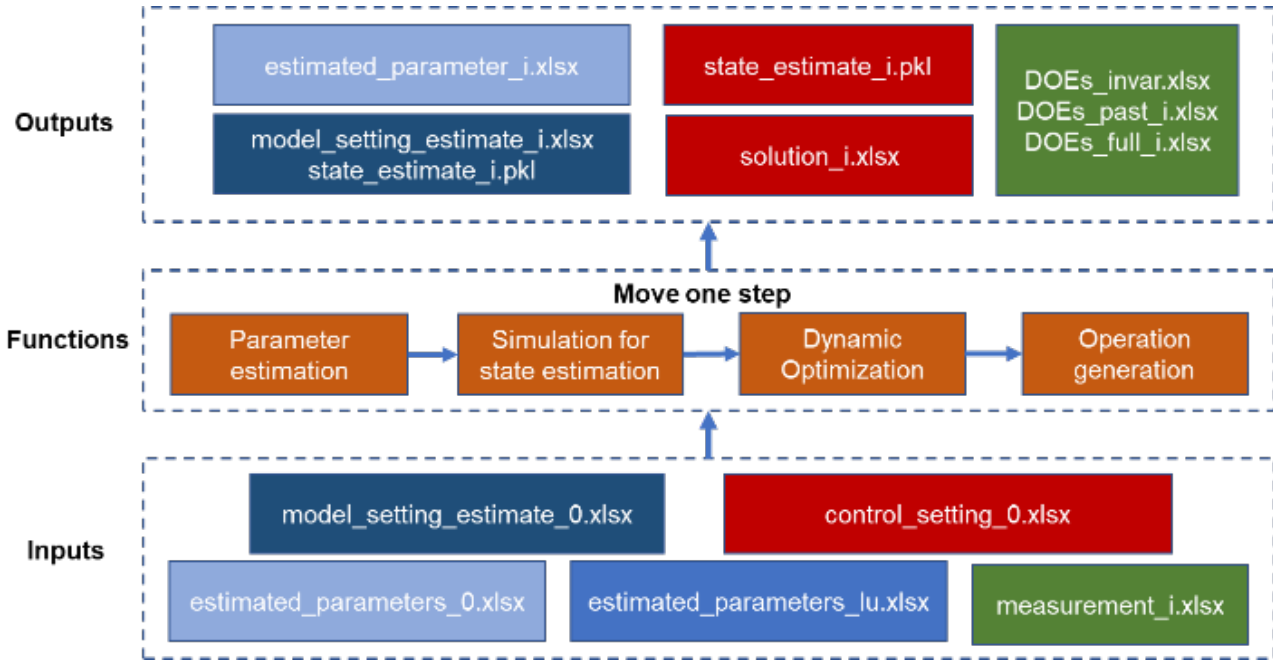

Figure 2. ANMPC implementation in GlycoPy.

$$\text{ML: } \frac{1}{2} r^{\top} \Sigma_y^{-1} r, \tag{4}$$

$$\text{MAP: } \frac{1}{2} r^{\top} \Sigma_y^{-1} r + \frac{1}{2} (p - \hat{p})^{\top} \Sigma_{\hat{p}}^{-1} (p - \hat{p}), \tag{5}$$

where OLS, ML and MAP stand for original least squares, maximum likelihood, and maximum a posteriori, respectively.

The resulting NLP is solved using GlycoPy's optimization interface. For post-analysis, `PostEstimation` computes the Fisher information matrix, parameter covariance, and confidence intervals using the output sensitivity $X = \partial \hat{y} / \partial p$.

For practical usability, GlycoPy supports: (i) estimating initial states together with parameters, (ii) optional log-parameter transforms to improve conditioning when parameters span orders of magnitude, (iii) diagnostic mode to debug simulations/experiments, and (iv) Excel-based inputs for experimental operating conditions, measurement data, initial guesses, bounds, and priors.

2.5 NMPC

The simulation, parameter estimation, and dynamic optimization modules in GlycoPy can be combined to implement state NMPC and adaptive NMPC (ANMPC) for fed-batch processes. In each control interval k, ANMPC follows a standard loop in a shrinking horizon: (i) when new measurements are available, update the parameter estimate $\hat{p}(k)$ using all collected data; (ii) simulate the model to obtain the current state estimate $\hat{x}(k)$; (iii) solve an optimal control problem over the remaining horizon $[t_k, T]$ using $\hat{p}(k)$ and $\hat{x}(k)$; and (iv) apply the first control move and sample again. The algorithm terminates at the end of the batch.

A practical challenge in applying NMPC for large-scale processes is debugging and diagnosis: users often need to stop, inspect intermediate results (states, parameters, predictions, control actions), and restart from any interval without rerunning the full batch. To support this workflow, GlycoPy adopts the implementation structure in Fig. 2, which separates the NMPC loop into reproducible blocks, **inputs**, **functions**, and **outputs**.

In the **inputs** block, all settings and data can be provided via spreadsheets, including model/initial-condition settings, control scheduling (manipulated variables and time grids), parameter priors (values, covariance, confidence intervals), parameter bounds, and measurement files with noise levels.

In the **functions** block, users implement four core steps based on existing pipelines in the previous section: parameter estimation, simulation for state estimation, dynamic optimization, and operation generation. Here, the operation generation function generates files recording not only the optimized future control actions but also the actual past actions. Here, recording the actual past actions is important because low-level actuators may deviate from ideal setpoints; these recorded actions are then reused for subsequent estimation and prediction.

In the **outputs** block, at each interval, the framework stores updated parameter estimates and covariance, updated model settings (including refreshed initial states),

optimized decision variables (past and full horizon), and a solution file containing predicted trajectories and applied controls.

State NMPC uses the same structure but does not update parameters; it relies on simulations with fixed parameters combined with available measurements for state updates.

3. CASE STUDY

We demonstrate GlycoPy on NMPC of the mAb N-glycosylation process using the multiscale glycosylation process model, which has been studied in [9]. The model provides the mapping from extracellular conditions and operating actions to intracellular Golgi processing and the resulting extracellular glycoform distribution, which can be illustrated by Fig. 3.

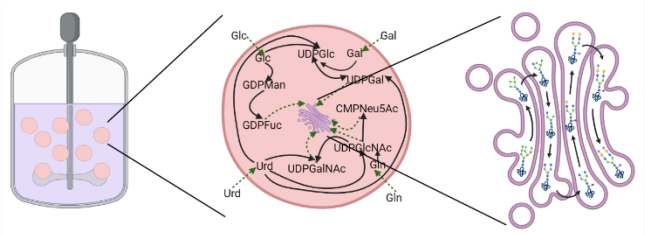

Figure 3. Multiscale mAb glycosylation process. The submodels, from left to right, are: the cell culture and metabolism model, the NSD synthesis model, and the Golgi glycosylation model. Reprinted from (Ma et al., 2025b) with permission.

The multiscale model consists of three interconnected layers [10]. At the bioreactor level, a cell culture submodel describes viable cell growth, extracellular metabolite dynamics, specific productivity, and secreted monoclonal antibody concentrations. These extracellular conditions are then passed to an intracellular nucleotide sugar donor synthesis submodel, which maps culture-level metabolite information to intracellular donor pools through a mechanistic representation of the relevant biosynthetic pathways and kinetics. Finally, a Golgi-level glycosylation submodel uses these donor species to predict the distribution of antibody glycoforms emerging from the secretory pathway by combining enzyme-mediated reaction networks with glycoprotein transport dynamics.

In the multiscale model, the Golgi subsystem is particularly complex, involving a large reaction network and spatially distributed dynamics. Implementing the full system as a single monolithic model would be difficult to maintain and reuse. Instead, GlycoPy enables a structured decomposition as shown in Fig. 4, where the overall model is built from the three main submodels, and the Golgi module is further organized into reusable components: transport proteins, enzyme, reaction, and network blocks. This hierarchy supports scalable simulation, estimation, optimization, and control for realistic biomanufacturing applications.

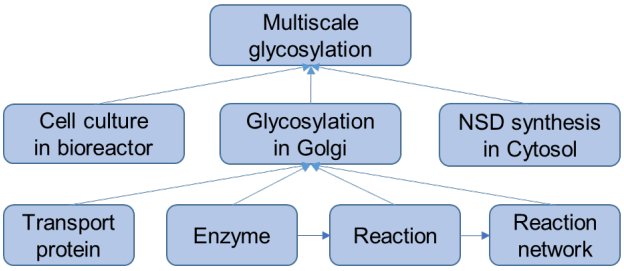

Figure 4. Implementation of the multiscale glycosylation model in GlycoPy.

The multiscale model takes the form of a large-scale partial differential–algebraic equation (PDAE) system, comprising 30 ordinary differential equations, 34 partial differential equations, and many strongly nonlinear algebraic relationships. The simulation using the finite difference method will be too slow to be used for NMPC. The symbolic simulation in GlycoPy enables the implementation of the quasi-steady-state (QSS) simulation algorithm and ensures differentiable simulations, which will be difficult in other modeling and control platforms. The steps of the QSS algorithm are as follows [11]: (i) the cell culture and NSD submodels with slow dynamics are simulated at selected time points and generate state variables env to be used for the following Golgi model simulation; (ii) the Golgi submodel is solved at steady state at each time point (parallelizable) with env as inputs, and generates intracellular glycosylation profiles $Y_{\mathrm{glycan}}^{\mathrm{intra}}$; (iii) the resulting $Y_{\mathrm{glycan}}^{\mathrm{intra}}$ are treated as time-varying parameters for the cell culture submodel to generate the extracellular glycosylation profiles $Y_{\mathrm{glycan}}^{\mathrm{extra}}$. This algorithm is packaged in `SimulationQSS` (derived from `Simulation`) and can return numerical or symbolic results depending on input types. Reported speedups exceed 300 folds for the multiscale glycosylation case. Because the QSS simulation is used, the total computation time for a single ANMPC step, including parameter estimation, state estimation, and dynamic optimization, remains below 2 h [9].

In the control problem, we maximize the galactosylation index (GI) at harvest time ($T = 288$ h),

$$\mathrm{GI}(T) = [\mathrm{FA2G1}](T) + 2\,[\mathrm{FA2G2}](T), \tag{6}$$

where $[\mathrm{FA2G1}]$ and $[\mathrm{FA2G2}]$ are the extracelluar concentrations of the corresponding glycoforms at harvest. The NMPC problem includes operational feasibility constraints on the bioreactor and product quality: (i) cell viability must remain above 60%; (ii) the media in the bioreactor volume must stay between 0.75 L and 2.25 L ; and (iii) the fraction of Man5-attached mAb is constrained by an upper bound 5% to avoid unacceptable high-mannose

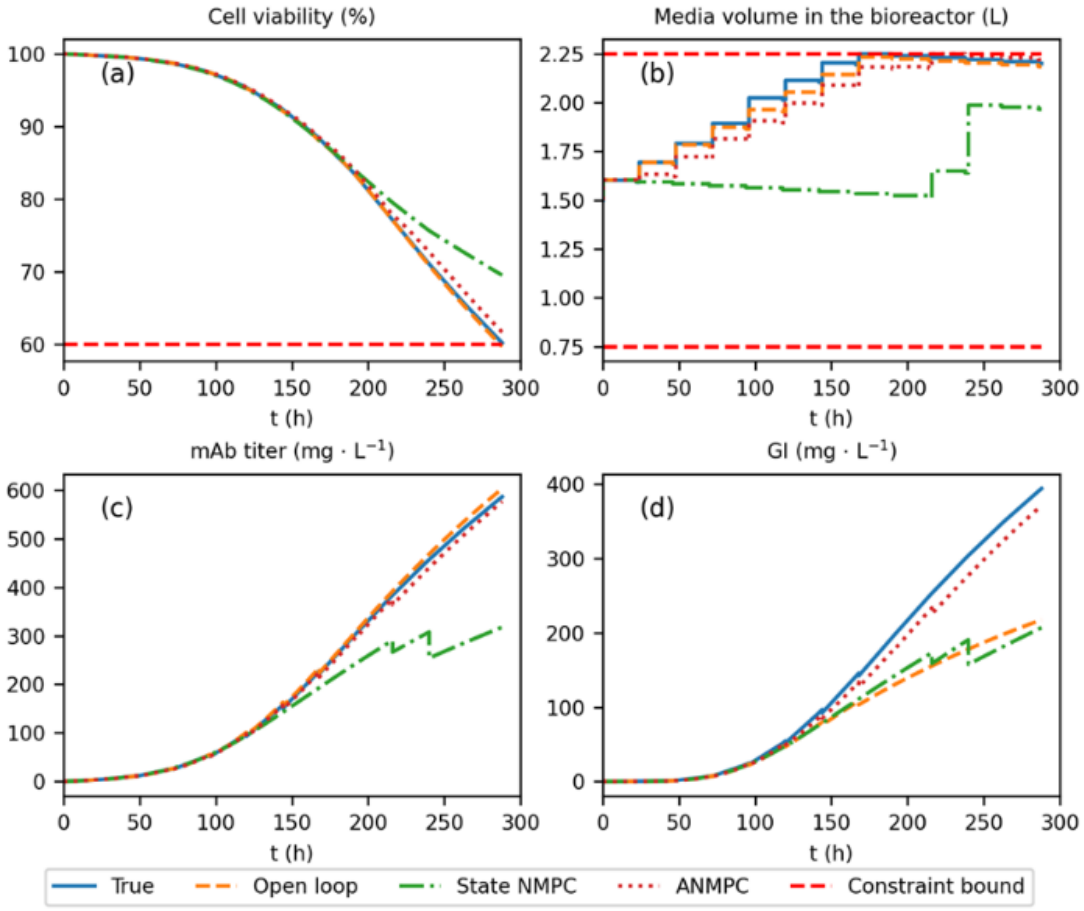

Figure 5. State variable trajectories under ground-truth optimization results and different control strategies.

content, where Man5 refers to High-mannose glycan containing five mannose residues. The controller adjusts two feeds commonly used in practice for the control purpose: a media supplement and a galactose (Gal) supplement. These inputs influence cell growth/metabolism and NSD availability, thereby shaping Golgi reaction fluxes and final glycoform composition.

The optimization and NMPC frameworks in the last section are used for the case study. We compare three control strategies: open-loop optimization, state NMPC, and ANMPC. Because of the use of the efficient quasi-steady-state simulation. The results are shown in Fig. 5.

All three strategies produce feasible trajectories satisfying the viability, volume, and Man5 constraints. However, ANMPC achieves a substantially higher terminal GI, which approximately doubles the GI achieved by open-loop optimization and state NMPC, and approaches the performance of ground-truth optimization. This highlights that, for glycosylation process control, closed-loop optimal control and adaptation can deliver large gains in product quality while maintaining process feasibility.

4. CONCLUSION

To promote the practical deployment of NMPC in real-world (bio)chemical processes, this work introduces GlycoPy, an equation-oriented (EO), object-oriented Python framework for modeling, optimization, and optimal control. GlycoPy allows users to focus on writing equations while providing hierarchical modeling capabilities that simplify the development, use, and maintenance of large-scale and multiscale process models. GlycoPy retains CasADi's powerful symbolic framework, enabling the use of symbolic simulation to develop efficient and differentiable simulation algorithms. The framework also includes parameter estimation and dynamic optimization modules, which form core building blocks for NMPC implementations. In addition, GlycoPy can export key intermediate results (e.g., parameter estimates and state trajectories), supporting transparent interpretation and efficient diagnosis of NMPC runs. Finally, a multiscale mAb glycosylation case study illustrates the effectiveness of the proposed framework.

DIGITAL SUPPLEMENTARY MATERIAL

GlycoPy is available at https://github.com/eaglema/glycopy, and the libraries for the glycosylation process modeling, optimization and control based on GlycoPy is available at https://github.com/eaglema/glycosylation.

ACKNOWLEDGEMENTS

This work was supported by a Project Award Agreement from the National Institute for Innovation in Manufacturing Biopharmaceuticals (NIIMBL), with financial assistance from the U.S. Department of Commerce, National Institute of Standards and Technology (NIST), awards70NANB17H002 and 70NANB20H037.

AUTHOR IDENTIFIERS

Author ORCIDs:
Yingjie Ma: 0009-0006-4458-9859
Jing Guo: None
Richard D. Braatz: 0000-0003-4304-3484

REFERENCES

1. P. I. Barton. The modelling and simulation of combined discrete/continuous processes. Imperial College London (University of London), London, 1992.
2. Gunnell L, Nicholson B, Hedengren JD. Equation-based and data-driven modeling: open-source software current state and future directions. Computers & Chemical Engineering 181:108521 (2024). https://doi.org/10.1016/j.compchemeng.2023.108521
3. Siemens Process Systems Engineering Ltd. *gPROMS ModelBuilder*. (2025).
4. Beal LDR, Hill DC, Martin RA, Hedengren JD. GEKKO optimization suite. Processes 6:106 (2018). https://doi.org/10.3390/pr6080106
5. Fiedler F, Karg B, Lüken L, Brandner D, Heinlein M, Brabender F, Lucia S. Do-mpc: towards FAIR nonlinear and robust model predictive control. Control Engineering Practice 140:105676 (2023). https://doi.org/10.1016/j.conengprac.2023.105676
6. S. S. Bhonsale, M. Vallerio, D. Telen, D. Vercammen, F. Logist, J. van Impe. SolACE: An open source package for nonlinear model predictive control and state estimation for (bio)chemical processes." P. 1971–1976 in Computer Aided Chemical Engineering. Vol. 38, edited by Z. Kravanja and M. Bogataj. Elsevier. (2016)
7. Hart WE, Watson JP, Woodruff DL. Pyomo: modeling and solving mathematical programs in python. Math. Prog. Comp. 3:219-260 (2011). https://doi.org/10.1007/s12532-011-0026-8
8. Andersson JAE, Gillis J, Horn G, Rawlings JB, Diehl M. Casadi: a software framework for nonlinear optimization and optimal control. Math. Prog. Comp. 11:1-36 (2018). https://doi.org/10.1007/s12532-018-0139-4
9. Ma Y, Guo J, Dubs AB, Ganko K, Braatz RD. Adaptive nonlinear model predictive control of monoclonal antibody glycosylation in CHO cell culture. Control Engineering Practice 169:106731 (2026). https://doi.org/10.1016/j.conengprac.2025.106731
10. Kotidis P, Jedrzejewski P, Sou SN, Sellick C, Polizzi K, del Val IJ, Kontoravdi C. Model?based optimization of antibody galactosylation in CHO cell culture. Biotech & Bioengineering 116:1612-1626 (2019). https://doi.org/10.1002/bit.26960
11. Ma Y, Guo J, Maloney AJ, Braatz RD. Quasi-steady-state approach for efficient multiscale simulation and optimization of mab glycosylation in CHO cell culture. Chemical Engineering Science 318:122162 (2025). https://doi.org/10.1016/j.ces.2025.122162