



MASTERARBEIT | MASTER'S THESIS

Titel | Title

Improving the performance of dynamic control flux balance
analysis models for bioprocess optimization

verfasst von | submitted by

Maria Christina Kovacs BSc

angestrebter akademischer Grad | in partial fulfilment of the requirements for the degree of
Master of Science (MSc)

Wien | Vienna, 2025

Studienkennzahl lt. Studienblatt | Degree
programme code as it appears on the
student record sheet:

UA 066 875

Studienrichtung lt. Studienblatt | Degree
programme as it appears on the student
record sheet:

Masterstudium Bioinformatik

Betreut von | Supervisor:

Univ.-Prof. Dipl.-Ing. Dr. Jürgen Zanghellini

Kurzfassung

Die Simulation und Optimierung von Bioprozessen ermöglicht es, zum Beispiel in der industriellen Biotechnologie oder pharmazeutischen Produktion die Produktivität zu steigern, Kosten zu senken und einen effizienten Einsatz von Ressourcen zu gewährleisten. Dynamic control Flux Balance Analysis (dcFBA) ist dafür eine effektive mathematische Methode und wird zur Simulation und Analyse von metabolischen Prozessen unter dynamischen Bedingungen eingesetzt. Bei der Simulation eines Bioprozesses liefern dcFBA-Modelle wichtige Einblicke in den zellulären Stoffwechsel. Die algorithmische Komplexität solcher Modelle stellt jedoch bei besonders umfangreichen Simulationen eine erhebliche Herausforderung dar. Um die Leistung der Lösungsverfahren solcher Modelle zu verbessern, untersucht diese Arbeit verschiedene Ansätze, darunter die Optimierung der Solver-Einstellungen, die Verfeinerung des Modelles sowie die mögliche Beschleunigung mittels GPU-Rechenleistung.

Den Ausgangspunkt der Studie bilden zwei zuvor entwickelte dcFBA-Modelle, die als Referenz herangezogen werden. Sie repräsentieren unterschiedliche Bioprozesse, diese sind jedoch von der Struktur ähnlich implementiert. Dazu wurde das Modellierungsframework **JuMP** und zur Lösung der Solver **IPOPT** verwendet. Im ersten Teil der Arbeit wird das Lösungsverfahren bestehender Modelle gezielt optimiert, indem unterschiedliche **IPOPT**-Optionen systematisch getestet und ihre Auswirkungen auf die Performanz untersucht werden. Dadurch sollen spezielle Solver-Einstellungen identifiziert werden, die insbesondere die Konvergenz, die Laufzeit und die Lösungsqualität verbessern. Weiters sollen mögliche Muster spezifischer Optionen aufgezeigt werden, die sich modellübergreifend als leistungsfördernd erweisen. Alle Anpassungen werden mit den beiden Referenzmodellen verglichen. Die daraus gewonnenen Erkenntnisse werden anschließend an zwei deutlich größeren Modellen validiert. Im zweiten Teil wird die Leistungsfähigkeit kürzlich entwickelter Alternativen zu **IPOPT** und **JuMP** untersucht, konkret des Modellierungsframeworks **ExaModels** und des Solvers **MadNLP**. Die beiden Programme werden im Hinblick auf ihre Stärken und Schwächen im Kontext von dcFBA-Modellen bewertet, wobei besonderes Augenmerk auf ihr Potenzial zur GPU-Nutzung gelegt wird. Dabei steht die Frage im Mittelpunkt, ob eine parallele Rechenstruktur die Leistungsfähigkeit von diesen speziellen Implementationen verbessern kann.

Bestimmte Konfigurationen der **IPOPT**-Optionen führte in einigen Fällen zu einer erheblichen Reduktion der Laufzeit, insbesondere bei einem der größeren Modelle, wo eine Reduktion von 78% erzielt wurde. Optionen im Zusammenhang mit Warm-Start und Initialisierung hatten dabei den größten Einfluss auf die Laufzeit. Hinsichtlich des zweiten Ansatzes gelang es **MadNLP** im Vergleich zu **IPOPT** in weniger Iterationsschritten eine Lösung mit vergleichbarer Qualität zu finden jedoch nicht erheblich schneller. Die GPU-Beschleunigung mit **ExaModels** brachte keine konsistenten Leistungsverbesserungen und war lediglich bei kleineren Teilproblemen der Modellen von Vorteil. Bei größeren Modellen haben möglicherweise eine unzureichende Skalierung sowie die Struktur der Implementierung die effektive Nutzung von GPU begrenzt.

Zusammenfassend fokussiert sich diese Arbeit auf die Verbesserung des Lösungsprozesses von dcFBA-Modellen durch Optimierung von Solver-Einstellungen und den Einsatz von GPU-Nutzung. Ziel ist es, die rechnerische Effizienz zu steigern, um die Anwendung von dcFBA auf komplexere Simulationen von Bioprozessen zu ermöglichen. Die Ergebnisse machen deutlich, dass sowohl die Solver-Parameter als auch ein gut durchdachtes mathematisches Modell entscheidend für den Erfolg der Simulationen sind. Letztlich kann diese Arbeit dazu beitragen, dcFBA in der computergestützten Biologie nutzbarer zu machen, einem Bereich, in dem schnelle und leistungsfähige Simulationen zunehmend an Bedeutung gewinnen.

Abstract

Simulating and optimizing bioprocesses allows us to improve productivity, reduce costs, and guarantee efficient resource usage in industrial biotechnology or pharmaceutical manufacturing. Therefore, dynamic control flux balance analysis (dcFBA) is a powerful mathematical framework that is used to simulate and analyze metabolic processes under dynamic conditions. When simulating a bioprocess, dcFBA models provide valuable insights into cellular metabolism. However, their computational complexity poses significant challenges, particularly for large-scale simulations. To improve performance, this thesis explores several approaches, including solver optimization, model refinement, and GPU acceleration.

The study begins with two previously developed dcFBA models serving as references. They represent different bioprocesses but are implemented in a similar way using a combination of the modeling framework `JuMP` and the solver `IPOPT`. The first objective is to improve the current solution process by systematically testing different `IPOPT` option configurations and analyzing how they affect performance. The goal is to identify settings that, above all, improve convergence, runtime, and solution quality, while also revealing patterns of specific options that consistently improve performance across different problems. All adaptations will be compared to these two reference models. The findings of these tests will then be validated in two substantially larger models. As a second objective, this study evaluates recently developed alternatives to `IPOPT` and `JuMP`, the solver `MadNLP`, and the modeling framework `ExaModels`, testing them on the two reference models. These tools are assessed for their strengths and limitations in the context of dcFBA, with a particular focus on their ability to leverage GPU acceleration and also reduce computational costs. The goal is to determine whether parallel computing can improve the performance of dcFBA models. The configurations of the solver options (objective 1) led to reduced runtime in some cases, particularly for one of the larger models, where a reduction of 78% was achieved. The options related to warm-start and initialization had the most noticeable impact on run time. Regarding objective 2, `MadNLP` achieved a lower number of iterations compared to `IPOPT` while maintaining similar solution quality, but not particularly faster. The use of GPU with `ExaModels` did not consistently improve performance and was only beneficial for smaller sub-problems of the models. For larger models, factors such as poor scaling and the overall implementation structure may limit the effectiveness of GPU execution.

In summary, this thesis focuses on improving the solution process for dcFBA models by optimizing solver configurations, refining model structures, and leveraging GPU acceleration. The goal is to improve computational efficiency, enabling the application of dcFBA to more complex bioprocess simulations. The results highlight the importance of the solver parameters and a well-posed mathematical model for the successful convergence of the simulations. In the end, this work could help to make dcFBA more useful in computational biology, where fast and powerful simulations are becoming more important.

Acknowledgments

I am very grateful to Univ.-Prof. DI Dr. Jürgen Zanghellini for giving me the chance to work and explore the diverse field of metabolic network modeling. I also want to thank the whole Biochemical Network Analysis team for creating such a friendly and supportive place to work. I owe special thanks to Mathias Gotsmy for guiding me through this thesis, always offering help, and allowing me to work with his models. Finally, I am deeply thankful to my family and friends for their constant support, encouragement, and belief in me.

Contents

1	Introduction	2
1.1	Mathematical background	2
1.1.1	Optimization	2
1.1.2	Linear Programming	3
1.1.3	Nonlinear Programming	5
1.2	Methods for Solving	7
1.2.1	Lagrange Multiplier Method	7
1.2.2	Karush–Kuhn–Tucker conditions	8
1.2.3	Interior Point Method	9
1.3	Solver Packages	13
1.3.1	IPOPT	13
1.3.2	MadNLP	18
1.4	Software	19
1.4.1	JuMP	19
1.4.2	ExaModels	20
1.5	Bioprocesses and Metabolic Modeling	21
1.5.1	Fed-Batch Processes	21
1.5.2	Metabolic Modeling	21
1.5.3	Flux balance analysis	23
1.5.4	Dynamic flux balance analysis	24
1.5.5	Dynamic control flux balance analysis	25
2	Methodology	28

2.1	Reference Models	28
2.1.1	Model formulation	29
2.2	IPOPT - Solver Optimization	31
2.2.1	Testing strategy	31
2.3	Alternative Solver MadNLP	33
2.3.1	Testing strategy	33
2.4	Model set-up	34
2.4.1	JuMP	35
2.4.2	ExaModels	36
2.4.3	GPU Integration	40
2.5	Hard- and Software	41
3	Results and Discussion	42
3.1	IPOPT Option Testing	42
3.1.1	Reference Models	42
3.1.2	Larger Implementations	46
3.1.3	Discussion	47
3.2	Solver performance	50
3.2.1	JuMP and MadNLP (CPU)	50
3.2.2	ExaModels and MadNLP (CPU)	51
3.2.3	JuMP and ExaModels and MadNLP (CPU)	53
3.2.4	JuMP and ExaModels and MadNLP (GPU)	55
3.2.5	Discussion	57
4	Conclusion	62
5	Bibliography	64
A	Supplementary Material	74

Chapter 1

Introduction

1.1 Mathematical background

In this section, we introduce the fundamental mathematical concepts of optimization, with a particular focus on the methods used to mathematically model fed-batch processes. Although optimization covers a wide range of techniques, the discussion is limited to the relevant aspects necessary to understand the underlying principles and their application in this context. An introduction to fed-batch processes is given in Section 1.6.

1.1.1 Optimization

Optimization is a mathematical process of finding the best possible solution from a set of feasible options. This typically involves maximizing or minimizing an objective function by selecting the appropriate input values within defined constraints [1]. Optimization is fundamental in decision making and system analysis across diverse fields [2].

A general optimization model consists of three key components:

- **Decision variables:** The variables are the controllable factors chosen during optimization to help achieve the optimal solution.
- **Objective function:** A mathematical expression that defines the goal of the optimization (maximizing or minimizing).
- **Constraints:** Mathematical expressions, such as equalities or inequalities, that restrict the solution space are often based on real-world limitations. [3]

A key step in the optimization process is to classify the type of problem, as different algorithms are designed for specific types. Optimization methods are classified according to factors such as the type of constraint, allowed values of variables, the problem structure, and the form of equations. Other criteria include whether the variables are deterministic, the number of objectives, and the application area [4].

The main categories of optimization problems include:

Continuous and Discrete: In continuous optimization, variables can take any real values, enabling the use of smooth gradient-based techniques. Discrete optimization restricts variables to specific sets of values (e.g., integers), making these problems more complex but essential in fields such as logistics [4].

Unconstrained and Constrained: Unconstrained problems involve no restrictions on variables, often occurring when constraints are embedded in the objective function. Constrained problems include explicit bounds, equalities, or inequalities, modeling real-world limitations like physical laws, for example [4].

Single and Multiple Objectives Most problems involve only a single objective to optimize. Feasibility problems have no objective, with the only purpose of satisfying all constraints. Multi-objective optimization involves optimizing several objectives simultaneously. These problems need special consideration, as the objectives are often conflicting [4].

Depending on the form of the objective function and constraints, optimization problems can be classified as linear, nonlinear, geometric, or quadratic programming problems [4]. The following sections will focus on linear and nonlinear optimization.

1.1.2 Linear Programming

Linear programming (LP) is a special mathematical optimization technique used to solve problems where the objective function is linear, subject to a set of linear constraints. These constraints, typically expressed as inequalities, define the feasible region by restricting the possible values of the decision variables [5]. Two commonly used special cases of linear programming problems are known as the standard form and the inequality form [6]. In the standard form, all constraints are expressed as equalities, and the decision variables are required to be non-negative:

$$\begin{aligned} &\text{maximize} && c^T x \\ &\text{subject to} && Ax = b \\ &&& x \geq 0. \end{aligned} \tag{1}$$

In contrast, the inequality form includes only inequality constraints, without any equalities:

$$\begin{aligned} &\text{maximize} && c^T x \\ &\text{subject to} && Ax \leq b \\ &&& x \geq 0, \end{aligned} \tag{2}$$

where x is the vector of decision variables, c is the coefficient vector representing the objective function, A is a matrix defining the constraints, and b is a vector specifying the constraint limits [6].

Another important concept used in the dcFBA reference models (see Methods Section 2.1) is the use of slack variables. These special variables help transform inequality constraints into equality constraints. [7]. Given the inequality constraint form (2) we introduce slack variables $s \geq 0$ such that:

$$Ax + s = b. \quad (3)$$

Introducing slack variables replaces each inequality constraint with an equality constraint and a non-negativity constraint [6]. Slack variables simplify the geometry of the feasible region, making it easier for algorithms to explore feasible regions efficiently. Methods like the Simplex Algorithm (see Figure 1.1) iterate over the corners of the feasible region. Slack variables help to define these vertices clearly by making all constraints equalities [8]. However, slack variables increase the number of optimization variables, which can potentially negatively impact the solving process.

LP problems can result in different types of solutions:

A feasible solution satisfies all constraints, while an infeasible solution violates at least one. The set of all feasible solutions forms the feasible region, which may be empty in some cases, resulting in an infeasible problem. In LP, the feasible region is always convex [5].

An LP problem is unbounded if the objective function can be improved indefinitely without violating any constraints. This occurs when the feasible region allows the objective function to increase or decrease without limit, depending on whether the problem is a maximization or minimization type [5].

A problem is bounded if the objective function is restricted to a finite range within the feasible region. Among feasible solutions, an optimal solution produces the most favorable value of the objective function. However, a problem may not have an optimal solution if it is infeasible or unbounded [5].

In linear programming, each primal problem (i.e., the problem as stated in Equations 1 or 2) has an associated counterpart known as the dual problem. This duality is not just a mathematical formality. It is a fundamental concept that offers valuable insight and provides alternative methods for finding optimal solutions. In some cases, solving the dual problem can be more efficient than solving the primal problem.

The dual problem can be seen directly as a mirror image of the primal. The coefficients of the primal objective function become the constraints in the dual, and vice versa. Moreover, the directions of inequalities are reversed, and the roles of variables and constraints are swapped [7].

Formally, the relationship between the primal and dual problems can be expressed as follows.

$$\text{Primal: } \max c^T x, \quad \text{subject to } Ax \leq b, \quad x \geq 0, \quad (4)$$

$$\text{Dual: } \min b^T y, \quad \text{subject to } A^T y \geq c, \quad y \geq 0, \quad (5)$$

where y represents the dual variables associated with the constraints of the primal problem.

Duality has several key properties that establish a direct connection between the primal and dual problem as outlined below:

- If the primal problem has an optimal solution x^* , the dual problem also has an optimal solution y^* .
- The **strong duality theorem** states that the optimal values of the objective functions in both problems are equal:

$$c^T x^* = b^T y^*. \quad (6)$$

- If either the primal problem or the dual problem is unbounded, the other must be infeasible. [7]

Linear programming is a powerful mathematical tool for solving optimization problems in fields such as economics, engineering, and biology. It allows real-world scenarios to be modeled efficiently using linear constraints and objectives. Applications range from manufacturing and logistics to finance and systems biology [5]. In particular, flux balance analysis (see Section 1.5.3) is a prominent example in which LP is used to model and optimize metabolic networks [9]. Its strength lies in its flexibility and the availability of efficient solution methods. Algorithms like the Simplex and Interior Point methods (see Section 1.2.3), implemented in modern solvers, can handle large-scale problems quickly, making LP one of the most practical tools in optimization [8].

1.1.3 Nonlinear Programming

However, many real-world problems involve nonlinear objective functions or constraints and therefore cannot be effectively captured by linear programming. Nonlinear Programming (NLP) problems generalize linear programming by allowing such nonlinearities in the problem formulation [10].

Unlike LP, where both the constraints and the objective function are strictly linear, NLP problems permit general nonlinear functions, such as polynomials, exponential, and trigonometric terms. A problem is classified as non-linear if at least one component (the objective or a constraint) is nonlinear [11]. Although this flexibility expands the range of problems that can be modeled, it also introduces computational challenges, as the feasible region cannot always be assumed to be convex.

Unlike the matrix-based formulation previously used for LP, NLP problems in the literature are typically expressed in terms of general functions. To align with this convention, we now define the general NLP problem using functional notation.

Let $x \in \mathbb{R}^n$ be a vector with $n \geq 1$ components and let $f : \mathbb{R}^n \rightarrow \mathbb{R}$ be a smooth function. Then, the general constrained NLP problem is defined as.

$$\begin{aligned} & \text{maximize} && f(x) \\ & \text{subject to} && g_j(x) \leq 0, \\ & && h_i(x) = 0, \\ & && x \in X. \end{aligned} \tag{7}$$

where x is the vector of decision variables, $f(x)$ is a nonlinear objective function, $g_j(x) \in \mathbb{R}$ are inequality constraint functions, $h_i(x) \in \mathbb{R}$ are equality constraint functions, and X defines the feasible region of decision variables [10].

Slack variables s_j in NLP have the same purpose as in LP:

$$g_j(x) + s_j = 0, \quad s_j \geq 0. \tag{8}$$

It makes the problem structure often more uniform, which is useful for theoretical analysis (e.g., KKT conditions) and practical algorithms (e.g., interior-point and barrier methods) [12]. Those techniques are further discussed in Sections 1.2.2 and 1.2.3.

Duality also plays a central role in NLP, but its characteristics differ from those in LP, where the primal and dual problems are symmetric, and strong duality typically holds under mild feasibility conditions [7]. In contrast, duality in NLP is more complex and does not offer the same general guarantees.

To develop the dual for NLP, we introduce the Lagrangian function for (7):

$$L(x, \lambda, \mu) = f(x) + \sum_{j=1}^m \lambda_j g_j(x) + \sum_{i=1}^p \mu_i h_i(x), \tag{9}$$

where $\lambda \geq 0$ and $\mu \in \mathbb{R}^p$ are the Lagrange multipliers for inequality and equality constraints, respectively [13]. More details about this can be found in Section 1.2.1 and 1.2.2.

The Lagrange dual function is obtained by minimizing the Lagrangian with respect to the primal variables:

$$z(\lambda, \mu) = \inf_{x \in \mathbb{R}^n} L(x, \lambda, \mu). \tag{10}$$

This function is always concave in (λ, μ) , even if the primal problem is nonconvex, because it is the point-wise infimum of affine functions in (λ, μ) [6].

The dual problem then becomes:

$$\begin{aligned} & \max z(\lambda, \mu), \\ & \text{subject to } \lambda \geq 0. \end{aligned} \tag{11}$$

Duality in NLP differs from LP and therefore leads to different theoretical conclusions.

- **Weak Duality:** For any $\lambda \geq 0$ and μ , $z(\lambda, \mu) \leq p^*$, where p^* is the optimal value. This holds for all NLP, convex or not.
- **Strong Duality:** Unlike LP, strong duality does not always hold in NLP. It requires convexity of the primal problem, lower bounds, and a constraint qualification such as Slater's condition (existence of a strictly feasible point). Under these conditions, the primal and dual optimal values match: $p^* = d^*$ [13].

Solving the dual problem in NLP is generally more challenging because of nonlinearity and possible non-convexity, but it remains a fundamental concept and forms the basis of optimality conditions, such as the Karush-Kuhn-Tucker (KKT) conditions (see Section 1.2.2).

In nonlinear programming (NLP), the solution process typically results in a local optimal solution rather than a global one. A vector $x_L \in X$ is considered a local optimum if there exists a small neighborhood around it in which no other feasible point achieves a better objective value [14]. Since global optimal solutions cannot generally be identified without evaluating all local optima, NLP solutions are commonly treated as local by default. Therefore, unless explicitly stated otherwise, the term “optimal solution” usually refers to a local solution in the context of NLP [14]. In practice, it also happens that NLP solvers terminate with a solution that is almost locally solved. This indicates that the solution satisfies the optimality conditions up to a relaxed or reduced tolerance [15].

1.2 Methods for Solving

Many methods have been developed to solve different kinds of optimization problems. To keep it simple, only the methods that are relevant to those specific problems and have been used directly or indirectly to solve the models will be discussed.

1.2.1 Lagrange Multiplier Method

Constrained optimization problems frequently occur in various fields, including physics [16], chemistry [17], and economics [18]. One approach that consistently appears in this context is the method of Lagrange multipliers, which is a powerful tool for solving constrained optimization problems. It is a strategy for finding the local maxima and minima of a function subject to equation constraints [19].

Therefore, consider the general NLP problem (7) but only with the equality constraints being present. The optimization problem reduces to:

$$\begin{aligned} & \text{maximize} && f(x), \\ & \text{subject to} && h_i(x) = 0, \quad i = 1, \dots, r. \end{aligned} \tag{12}$$

The Lagrangian function is constructed:

$$\mathcal{L}(x, \lambda) = f(x) - \sum_{i=1}^r \lambda_i h_i(x), \tag{13}$$

where λ_i are called the Lagrange multipliers. The necessary condition for optimality is that the gradient of the Lagrangian vanishes, which mathematically means:

$$\nabla_x \mathcal{L}(x, \lambda) = \nabla f(x) - \sum_{i=1}^r \lambda_i \nabla h_i(x) = 0, \tag{14}$$

along with the constraint satisfaction:

$$h_i(x) = 0, \quad \forall i \tag{15}$$

[18].

This method applies to linear programming, nonlinear programming, and more general constrained optimization problems, and it forms the basis of the Karush–Kuhn–Tucker conditions for inequality-constrained cases, which will be discussed in the next section [19], [20].

1.2.2 Karush–Kuhn–Tucker conditions

When inequality constraints are also present, the Lagrange multipliers framework is extended to the more general Karush–Kuhn–Tucker (KKT) conditions. They give a set of necessary conditions for optimality for systems involving both equality and inequality constraints [21]. The necessary conditions are defined as follows:

Theorem 1.2.1. *Let x^* be a local minimum. If the objective and constraint functions are continuously differentiable, then there exist multipliers $\lambda_i \in \mathbb{R}$ and $\mu_j \geq 0$ such that the following KKT conditions are satisfied [21]:*

(i). **Stationarity:**

$$\nabla f(x^*) + \sum_{i=1}^r \lambda_i \nabla h_i(x^*) + \sum_{j=1}^m \mu_j \nabla g_j(x^*) = 0, \tag{16}$$

(ii). **Primal feasibility:**

$$h_i(x^*) = 0, \quad \forall i = 1, \dots, r, \quad g_j(x^*) \leq 0, \quad \forall j = 1, \dots, m, \quad (17)$$

(iii). **Dual feasibility:**

$$\mu_j \geq 0, \quad \forall j = 1, \dots, m, \quad (18)$$

(iv). **Complementary slackness:**

$$\mu_j \cdot g_j(x^*) = 0, \quad \forall j = 1, \dots, m. \quad (19)$$

The last condition implies that for each inequality constraint, either the constraint is active ($g_j(x^*) = 0$) and its multiplier may be positive, or the constraint is inactive ($g_j(x^*) < 0$) and its multiplier must be zero [21].

The KKT conditions are often necessary for optimality, but they do not hold universally, meaning that there are certain optimization problems where, for a given optimal solution, there do not exist any multipliers λ and μ that satisfy the KKT conditions [22]. This usually happens when constraint qualifications are violated, such as when active constraint gradients are linearly dependent, distorting the local feasible geometry [23]. Validity is guaranteed under standard regularity conditions, such as the linear independence of active constraint gradients (known as the Linear Independence Constraint Qualification, or LICQ) [24], Slater's condition for convex problems with strictly feasible points [23], and cases where all constraints are affine or the inequality constraints are concave [23].

In dynamic control flux balance analysis (dcFBA, see Section 1.5.5), the KKT conditions cannot be applied to the entire system because it is not formulated as a single optimization problem. Instead, dcFBA involves a sequence of steady-state flux balance analyses (FBA, see Section 1.5.3), typically formulated as linear programs, combined with the numerical integration of extracellular metabolite concentrations. This results in a hybrid system that couples discrete optimization with continuous dynamics [25]. Applying KKT conditions globally would require reformulating dcFBA as a dynamic optimization problem that simultaneously integrates all dynamics and constraints [26]. Such reformulations are highly complex and may violate smoothness assumptions or constraint qualifications required for KKT theory [11], [24].

1.2.3 Interior Point Method

Interior Point Methods (IPM) are a class of algorithms designed to solve optimization problems and are widely used for both linear and nonlinear programming [20], [27], [28]. In contrast to traditional methods like the simplex method, which navigates along the boundaries of the feasible region (see Figure 1.1-a), IPM progresses through its interior (see Figure 1.1-b), using barrier functions to maintain feasibility. IPMs are often more powerful and efficient, particularly for large-scale problems [29].

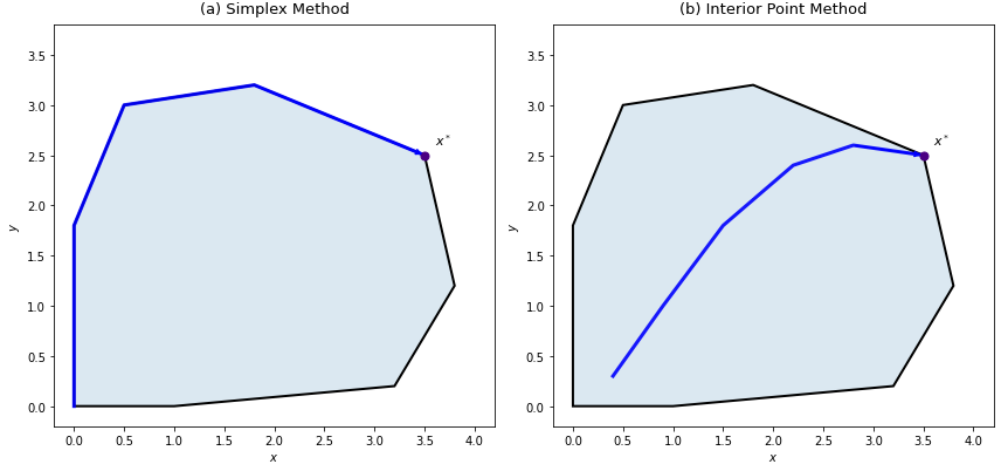


Figure 1.1: Graphical representation of the simplex and interior-point method. The blue line illustrates the path taken by the algorithm as it traverses the feasible region, here represented as a polygon.

Let us consider the general NLP problem (7), where $f(x)$ represents the objective function to be minimized, $g(x)$ denotes the inequality, and $h(x)$ the equality constraints.

One of the main challenges in solving such optimization problems is the efficient handling of inequality constraints. In conventional methods, these constraints are often explicitly incorporated into the solution process, but IPMs use a more sophisticated approach [30]. In IPMs, the core idea lies in the use of barrier functions to handle inequality constraints. Rather than solving the optimization problem directly, a logarithmic barrier function (20) is introduced to penalize infeasible solutions as they approach the boundary of the feasible region [28]. This approach transforms the original problem into a series of simpler subproblems that become easier to solve iteratively [28], [30].

$$\phi(x, \mu) = f(x) - \mu \sum_{i=1}^m \ln(-g_i(x)), \quad (20)$$

where $\mu \geq 0$ is a barrier parameter that controls the weight of the penalty for infeasibility.

The logarithmic term ensures that the iterations remain strictly feasible. As $\mu \rightarrow 0$, the effect of the barrier reduces, and the solution converges to an optimal point where the inequality constraints are satisfied with equality. The iterative process involves adjusting μ progressively, approaching zero as the solution becomes more optimal [30], [31].

The optimal solution to the constrained optimization problem must also satisfy the Karush-Kuhn-Tucker (KKT) conditions (Theorem 1.2.1). To solve the KKT system, IPMs use Newton's method to iteratively find the solution [30]. This involves solving a modified version of the KKT system at each iteration, where the objective function is minimized subject to both the equality and inequality constraints [32].

The KKT system is given by:

$$\begin{aligned}
\nabla f(x) + \nabla h(x)^T \lambda + \nabla g(x)^T \nu &= 0, \\
h(x) &= 0, \\
g(x) - s &= 0, \\
S\nu - \mu e &= 0,
\end{aligned} \tag{21}$$

where S is a diagonal matrix of slack variables, e is a vector of ones, and μ is the barrier parameter that decreases over iterations [28]. Here, $\nabla h(x)$ and $\nabla g(x)$ denote the Jacobian matrices of the equality and inequality constraints:

$$\nabla h(x) = \begin{bmatrix} \nabla h_1(x)^T \\ \vdots \\ \nabla h_r(x)^T \end{bmatrix}, \quad \nabla g(x) = \begin{bmatrix} \nabla g_1(x)^T \\ \vdots \\ \nabla g_m(x)^T \end{bmatrix}.$$

The linearized version of the KKT system can be expressed as:

$$\begin{bmatrix} H & \nabla h(x)^T & \nabla g(x)^T \\ \nabla h(x) & 0 & 0 \\ \nabla g(x) & 0 & -I \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta \lambda \\ \Delta \nu \end{bmatrix} = \begin{bmatrix} -\nabla f(x) - \nabla h(x)^T \lambda - \nabla g(x)^T \nu \\ -h(x) - g(x) + s \\ -h(x) \\ -g(x) + s \\ S\nu - \mu e \end{bmatrix} \tag{22}$$

where H is the Hessian matrix of the Lagrangian, given by:

$$H = \nabla^2 f(x) + \sum_i \nu_i \nabla^2 g_i(x) + \sum_i \lambda_i \nabla^2 h_i(x). \tag{23}$$

This is the second derivative of the Lagrangian with respect to x , which describes the shape of the objective function and the constraints. The solution to this linear system provides the updated direction ∇x , $\nabla \lambda$, and $\nabla \nu$. The Newton step is computed at each iteration, with μ being progressively reduced to ensure convergence to the optimal solution [28], [32].

Once the Newton direction is computed, a line search (see Figure 1.2) is performed to determine the step size that ensures sufficient progress towards the solution while maintaining feasibility [33].

Algorithm 1 General Line Search Algorithm for NLP (7)

```
1: Initialize: Choose initial point  $x_0$ , tolerance  $\epsilon$ , and set iteration counter  $k = 0$ .
2: while  $\|\nabla f(x_k)\| \geq \epsilon$  do
3:   Compute Descent Direction:  $p_k = -\nabla f(x_k)$ 
4:   Define Search Function:  $h(\alpha) = f(x_k + \alpha p_k)$ 
5:   Choose Step Size  $\alpha_k$ 
6:   Update:  $x_{k+1} = x_k + \alpha_k p_k$ 
7:   if using an interior point method then
8:     Ensure feasibility:  $g_i(x_{k+1}) < 0 \quad \forall i$ 
9:     Use descent direction from  $\nabla \phi(x_k)$ 
10:  end if
11:  Set  $k \leftarrow k + 1$ 
12: end while
```

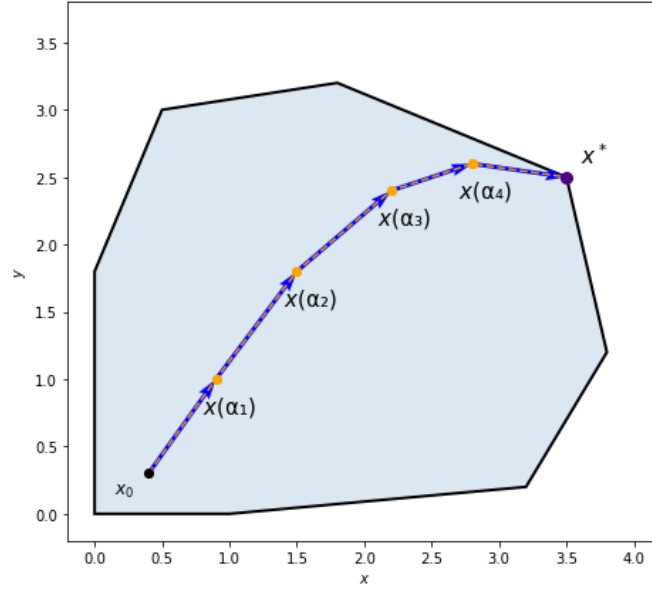


Figure 1.2: Interior point method on Example of Figure 1.1

In interior point methods, line search is applied to the barrier function (20) [31]. Newton's method provides a step direction, but taking a full step may violate feasibility. If the step size is too small, it could diverge or slowly converge. Line search adjusts to balance these factors. A common approach is the filter line search strategy, used in IPOPT [23], [32]. A more detailed description of this specific algorithm will be discussed later.

1.3 Solver Packages

Choosing an appropriate solver for a nonlinear programming problem can be challenging, as factors such as problem size, structure, smoothness, convexity, and feasibility can influence solver performance. Different solvers are based on various mathematical techniques and approaches, making it essential to select one that aligns with the problem characteristics and the available computational resources. In this chapter, we introduce IPOPT and MadNLP, two solvers tested for the dcFBA problems.

1.3.1 IPOPT

IPOPT [28] is short for Interior Point Optimizer (pronounced "Eye-Pea-Opt") is an open-source software package for solving large-scale nonlinear optimization problems. It is designed to handle general nonlinear programming (NLP) problems like (7). The objective function $f(x)$ as well as the constraints $g(x)$ and $h(x)$ can be linear or nonlinear, convex or non-convex, but must be at least twice continuously differentiable [28].

Algorithm

The filter line-search algorithm is a core component of the interior point method implemented in IPOPT. Its main role is to solve a sequence of barrier subproblems (20) by managing both progress toward optimality and feasibility [20].

$$\begin{aligned} \min_{x \in \mathbb{R}^n} \quad & \phi_\mu(x) := f(x) - \mu \sum_{i=1}^n \ln(x^{(i)}) \\ \text{s.t.} \quad & c(x) = 0. \end{aligned} \tag{24}$$

The key components in the following algorithm are:

- $\mathbf{x} \in \mathbb{R}^n$: primal decision variables.
- $\boldsymbol{\lambda} \in \mathbb{R}^m$: dual variables (Lagrange multipliers for equality constraints).
- $\mathbf{z} \in \mathbb{R}^n$: dual variables associated with inequality constraints.
- $\mu > 0$: barrier parameter, controls the weight of the logarithmic barrier term.
- $\theta(x)$: a measure of constraint infeasibility.

Filter Concept A filter $\mathbf{F}$ is an alternative to merit functions for accepting trial steps in constrained optimization. It stores previous iterates based on measures like feasibility, centrality, and optimality [34]. A new point can be accepted if it gets better in one area, as long as it does not get much worse in the others. This non-dominance criterion helps to select a step size that makes progress in either objective or constraint satisfaction, supporting global convergence [28].

Second-Order Correction (SOC): SOC is used when a step would otherwise be rejected because of poor feasibility. It adjusts the direction to better align with the constraints by solving a modified linear system that includes constraint violation corrections [35].

The filter line-search algorithm for solving the barrier problem:

Algorithm 2 Filter Line-Search Algorithm (IPOPT)

```

1: Input: Initial guess  $(x_0, \lambda_0, z_0)$ , with  $x_0, z_0 > 0$ , barrier parameter  $\mu_0 > 0$ , filter  $F_0$ ,
   and tolerances.
2: Set iteration counters  $j \leftarrow 0, k \leftarrow 0$ .
3: while not converged do
4:   if Overall residual  $E_0(x_k, \lambda_k, z_k) < \text{tol}$  then
5:     Stop: Converged to a solution.
6:   end if
7:   if Barrier residual  $E_{\mu_j}(x_k, \lambda_k, z_k) < \kappa_{\mu_j}$  then
8:     Update  $\mu_{j+1}, \tau_{j+1}$ , reset filter  $F_k$ .
9:     if  $k = 0$  then continue
10:    end if
11:  end if
12:  Compute search direction  $(d_x, d_\lambda, d_z)$  by solving the KKT system.
13:  Initialize step size  $\alpha_{k,0}$ , set  $l \leftarrow 0$ .
14:  while trial step not accepted do
15:    Compute trial point  $x_k(\alpha_{k,l}) := x_k + \alpha_{k,l}d_k$ .
16:    if Not acceptable to filter then
17:      Reduce  $\alpha_{k,l}$ , increment  $l$ , continue.
18:    end if
19:    if Sufficient decrease satisfied (case I or II) then
20:      Accept step.
21:    else
22:      Try Second-Order Correction (SOC):
23:      Solve for correction direction.
24:      Check filter acceptability and decrease.
25:    end if
26:  end while
27:  Update  $(x_{k+1}, \lambda_{k+1}, z_{k+1})$ .
28:  Augment filter if necessary.
29:  Increment  $k \leftarrow k + 1$ .
30:  if step size too small then
31:    Enter Feasibility Restoration Phase: Find a better feasible iterate.
32:  end if
33: end while

```

Solver Options

Since IPOPT provides a wide range of options, one key aspect of this thesis is to explore whether the solving process can be improved by experimenting with these options. The following table presents an overview of the most important option groups [15] used for the tests, detailing their functionalities and default values.

All available options and their possible values and detailed explanations can be found in the IPOPT documentation [36].

Initialization

Option	Description	Default
bound_frac	Desired minimum relative distance from the initial point to bound.	0.01
bound_push	Desired minimum absolute distance from the initial point to bound.	0.01
slack_bound_frac	Desired minimum relative distance from the initial slack to bound.	0.01
slack_bound_push	Desired minimum absolute distance from the initial slack to bound.	0.01

Termination

Option	Description	Default
acceptable_iter	Number of "acceptable" iterates before triggering termination.	0
acceptable_tol	"Acceptable" convergence tolerance (relative).	1e-06
compl_inf_tol	Desired threshold for the complementarity conditions.	0.0001
constr_viol_tol	Desired threshold for constraint and variable bound violation.	1e-06
dual_inf_tol	Desired threshold for the dual infeasibility.	1
max_iter	Maximum number of iterations.	3000
tol	Desired convergence tolerance (relative).	1e-08

Linear Solver MA27

Option	Description	Default
linear_solver	Linear solver used for step computations.	Ma27
ma27_pivtol	Pivot tolerance for MA27.	1e-08
ma27_pivtolmax	Maximum pivot tolerance for MA27.	0.0001

Warm-start

Option	Description	Default
warm_start_bound_frac	Same as bound_frac for the regular initializer.	0.001
warm_start_bound_push	Same as bound_push for the regular initializer.	0.001
warm_start_init_point	Warm-start for initial point.	Yes
warm_start_mult_bound_push	Same as mult_bound_push for the regular initializer.	0.001
warm_start_slack_bound_frac	Same as slack_bound_frac for the regular initializer.	0.001
warm_start_slack_bound_push	Same as slack_bound_push for the regular initializer.	0.001

Barrier Parameter Update

Option	Description	Default
mu_max	Maximum value for barrier parameter.	100000
mu_max_fact	Factor for initialization of max value for barrier parameter.	1000
mu_min	Minimum value for barrier parameter.	1e-11
adaptive_mu_globalization	Globalization strategy for adaptive mu mode.	obj-constr-filter
mu_oracle	Oracle for new barrier parameter in adaptive strategy.	quality-function

NLP and Hessian

Option	Description	Default
fixed_variable_treatment	How fixed variables should be handled.	make_parameter
nlp_lower_bound_inf	Any bound $\leq$ this value is considered $-\infty$.	-10^{19}
nlp_upper_bound_inf	Any bound $\geq$ this value is considered $+\infty$.	10^{19}
hessian_approximation	Indicates Hessian usage mode.	exact
limited_memory_initialization	Initialization for limited memory approximation.	scalar1
hessian_approximation_space	Subspace for Hessian approximation.	nonlinear-variables

IPOPT Output

When solving an NLP problem, IPOPT provides detailed iteration-wise output to monitor the optimization process (see Figure 1.3). Before the iterative solution begins, IPOPT displays the statistics of the problem, including the number of variables and non-zero elements in key matrices. If variables have fixed values (with identical upper and lower bounds), IPOPT may exclude them internally from the problem, affecting the reported statistics. As the optimization progresses, IPOPT prints iteration-specific information in a tabular format (see Figure 1.3) [37].

```

This is Ipopt version 3.14.4, running with linear solver ma27.

Number of nonzeros in equality constraint Jacobian...: 32789
Number of nonzeros in inequality constraint Jacobian.: 1770
Number of nonzeros in Lagrangian Hessian.....: 12494

Total number of variables.....: 9481
    variables with only lower bounds: 420
    variables with lower and upper bounds: 3001
    variables with only upper bounds: 1560
Total number of equality constraints.....: 7808
Total number of inequality constraints.....: 1740
    inequality constraints with only lower bounds: 180
    inequality constraints with lower and upper bounds: 0
    inequality constraints with only upper bounds: 1560

iter   objective    inf_pr  inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr ls
  0   0.0000000e+00  4.26e+01  1.00e+00  -1.0  0.00e+00   -  0.00e+00  0.00e+00  0
  1  -1.9028321e-01  4.26e+01  1.00e+00  -1.0  2.92e+02  -2.0  1.47e-04  2.23e-04f  1
  2  -2.5207567e-01  4.25e+01  5.15e+01  -1.0  1.37e+02  -1.6  9.99e-01  2.18e-04f  1
  3  -3.5806578e-01  4.21e+01  9.86e+01  -1.0  4.66e+01  -1.1  8.65e-04  1.06e-02f  1
  4  -4.0368657e-01  4.19e+01  8.62e+01  -1.0  5.05e+01  -1.6  9.53e-03  4.25e-03f  1

```

Figure 1.3: Output of a reference model: Starts by summarizing the model, followed by the statistics

The columns of the table represent the key metrics for each iteration, as described below.

- **iter:** The current iteration count, which includes both regular and restoration-phase iterations. Iterations in the restoration phase are marked with an "r" suffix.
- **Objective:** The value of the unscaled objective function at the current point. During restoration, this value remains linked to the original problem's objective.
- **inf_pr:** The unscaled constraint violation, defined as the infinity norm (maximum value) of the constraint residuals. This reflects the extent to which the constraints are satisfied.
- **inf_du:** The scaled dual infeasibility, representing the maximum violation of dual feasibility conditions, including slack variable adjustments and scaling effects.

- $\lg(\mu)$: The logarithm (base 10) of the barrier parameter μ that controls the influence of the barrier term in interior-point methods.
- $\|d\|$: The infinity norm of the primal step, capturing the step size for primal variables and slack adjustments.
- $\lg(\text{rg})$: The logarithm (base 10) of the regularization term applied to the Hessian of the Lagrangian. A dash ("-") indicates that no regularization was applied.
- α_{du} : The step size for updating the dual variables.
- α_{pr} : The step size for updating the primal variables. This value is followed by a diagnostic character indicating the outcome of the step acceptance test (e.g., "f" for filter-based acceptance or "h" for heuristic-based acceptance). [37]

This structured output helps users monitor convergence and find potential issues during optimization, for a better understanding and control over the solution process.

1.3.2 MadNLP

In addition to IPOPT [28], a similar solver was also tested and analyzed: MadNLP [38]. It is an open-source non-linear programming solver developed entirely in the Julia programming language. Like IPOPT, it is built on a filter line-search interior-point method, making it well suited for solving a wide range of non-linear optimization problems [39] [38].

One of the key distinctions of MadNLP is its ability to use diverse data structures and support both host (CPU) and device (GPU) memory architectures. The CPU (Central Processing Unit) handles general-purpose tasks and controls the system, while the GPU (Graphics Processing Unit) is optimized for parallel processing, making it ideal for graphics and compute-intensive operations. [39]

This flexibility allows it to optimize performance based on the available hardware.

- Host Memory refers to the system's main memory (RAM) used by the CPU. MadNLP can run its algorithms using this memory, which is typical for most computational tasks [40].
- Device Memory refers to memory available on GPU, which is specialized hardware for parallel processing. By leveraging device memory, MadNLP can solve optimization problems more quickly, especially those involving large datasets and complex computations [40].

This is a key advantage of MadNLP to leverage GPU acceleration, allowing it to efficiently solve large-scale optimization problems.

Additionally, MadNLP integrates seamlessly with popular Julia modeling packages such as JuMP [41] and NLPModels [42], simplifying the process of defining and solving optimization problems. To further enhance performance and flexibility, it supports a variety of sparse

and dense linear solvers, including Umfpack [43], Lapack [44], and HSL solvers [45]. This compatibility allows users to choose the most suitable solver for their specific needs [38].

1.4 Software

The modeling software presented here, including `JuMP` [41] and `ExaModels` [39], serves as an interface between the mathematical model and the solvers discussed previously. They convert user-defined model formulations (see Section 2.4) into solver-compatible representations, enabling efficient solutions of diverse classes of optimization problems.

1.4.1 JuMP

The reference models are implemented as a `JuMP` model, which stands for "Julia for Mathematical Programming" and is an open-source modeling language embedded in Julia. It enables users to formulate and solve various classes of optimization problems, including linear, mixed-integer, quadratic, conic quadratic, semi-definite, and nonlinear optimization [46]. It provides a high-level, user-friendly interface to define and solve constrained optimization problems efficiently [41]. It is especially well-suited for problems with the following characteristics, which also provide a foundation for structuring the model in `JuMP`:

- Decision variables (real or complex-valued)
- Objective functions (scalar or vector-valued)
- Constraints (linear, nonlinear, or mixed-integer). [47]

`JuMP` supports nonlinear programming with constraints and user-defined operators. However, all functions must be automatically differentiable. If a function is not differentiable or requires explicit derivative specification, users must provide the derivatives manually. Furthermore, for unconstrained optimization, `JuMP` can introduce computational overhead compared to specialized minimization tools [47]. Consequently, alternative optimization frameworks should be considered in such scenarios.

One of the key advantages of using `JuMP` includes its user-friendly syntax [47]. It closely mirrors standard mathematical notation, making it intuitive for users with backgrounds in optimization and applied mathematics. This feature simplifies the formulation of complex models, reducing the barrier to entry for new users. Another advantage compared to other modeling languages is its solver independence. `JuMP` is built on the `MathOptInterface` [48], which enables seamless integration with a wide range of optimization solvers, both open-source and commercial [49]. Compatible solvers include `IPOPT` and `MadNLP` among others. This independence allows users to switch between solvers effortlessly, depending on the problem requirements [41], [47].

Written entirely in Julia, **JuMP** is fully compatible with other Julia packages, simplifying its use in diverse applications. It can be embedded into larger workflows, including simulations, decomposition algorithms, and web-based applications [47]. **JuMP** provides computational efficiency that matches specialized modeling languages such as **AMPL** [50]. By directly communicating with solvers in memory, it eliminates the need for writing intermediary files, significantly reducing overhead. This efficiency is particularly beneficial for large-scale optimization problems where speed is critical [41].

1.4.2 ExaModels

ExaModels is an algebraic modeling and automatic differentiation framework developed in Julia. It is designed to use Single Instruction Multiple Data (SIMD) abstraction for nonlinear programming [39]. SIMD abstraction allows the same computation to be applied to many data points at once. In large optimization problems, the same kind of computation is repeated many times over different data (e.g. constraints for each node) [51]. The repetitive structure makes it possible to compute derivatives more efficiently [39]. This is especially beneficial for high-performance computing on GPU, where SIMD allows for significant speedups by avoiding redundant computation and enhancing scalability [52].

ExaModels requires Julia version at least 1.9 [53] and is still in an early stage of development. Users may encounter unintended behaviors or gaps in the documentation. The research underlying this thesis was performed between 2024 and early 2025.

ExaModels and **JuMP** are both algebraic modeling tools for mathematical optimization in Julia. While they share a common purpose, they differ significantly in design and performance characteristics. As mentioned before **ExaModels** is built around SIMD-friendly generators, which preserve the model’s parallel structure and enable efficient code generation for derivative evaluation on both CPU and GPU [39]. In contrast, **JuMP** offers a more flexible modeling interface but does not explicitly optimize for SIMD or GPU architectures [41]. It typically relies on general-purpose differentiation tools such as **ForwardDiff** [54] and **ReverseDiff** [55]. **ExaModels** is particularly well-suited for large-scale nonlinear optimization problems with structured sparsity and parallelism, whereas **JuMP** is better suited for general-purpose modeling with broad solver compatibility [39], [53].

ExaModels is particularly advantageous for optimization problems that exhibit:

- Nonlinear objectives and constraints,
- A large number of variables and constraints,
- High structural repetitiveness,
- Sparse Hessian and Jacobian matrices [53].

`ExaModels` is compatible with solvers that support the `NLPModel` [56] data type. Currently, it has been tested with `IPOPT` [28] and `MadNLP` [38]. However, some solver callbacks are not implemented yet, potentially leading to errors in certain configurations [53].

1.5 Bioprocesses and Metabolic Modeling

1.5.1 Fed-Batch Processes

A fed-batch culture is a semi-batch bioprocess in which fresh nutrients are added during cultivation, either intermittently or continuously, without removing culture fluid until the end of the process [57]. This method combines the simplicity of batch operation with extended control over availability of nutrients, allowing improved cell growth and product yield. In contrast, a batch process involves adding all nutrients and inoculum at the beginning of the fermentation, without further additions during the run. As a result, nutrient concentrations decline over time and cannot be actively controlled during cultivation [57]. Originally developed for yeast production in the early 1900s, it was discovered that gradually feeding sugar prevented the formation of ethanol and maximized yeast cell production [58], [57].

By regulating feed rates, fed-batch culture allows tight control over key parameters, for example, over substrate concentration [59], [57]. This enables operators to maintain optimal conditions over time, especially when controlling the availability of limiting nutrients, inducers, or precursors [57]. Additionally, fed-batch processes are ideal for reactions where product or substrate inhibition must be avoided, such as in autocatalytic or enzyme-inhibited reactions. Because cells themselves promote further growth, this method is especially effective for fermentation and cell cultures [57].

Fed-batch culture is widely used in industrial biotechnology as a result of its robustness, productivity, and flexibility. Common applications include the production of yeasts, antibiotics (e.g. penicillin), enzymes, amino acids, recombinant proteins, tissue cultures (e.g., CHO cells, hybridomas), and fine chemicals [57]. The method allows the operator to control not only what compounds are added (carbon sources, nitrogen, phosphates, etc.) but also how they are fed, through open-loop or feedback-controlled feeding strategies designed to maintain optimal physiological states [57]. Today, fed-batch culture remains the industry standard for pharmaceutical manufacturing and is used extensively for both research and commercial production because of its balance between control and scalability [59].

1.5.2 Metabolic Modeling

Metabolic modeling is a computational strategy used to represent and analyze the network of biochemical reactions occurring within a biological system. It provides a mathematical framework, primarily based on linear algebra, to study how metabolites are

transformed within cells and how these transformations impact cellular function [60]. By constructing metabolic models, researchers can predict metabolic fluxes [61], or explore cellular responses to environmental changes [62]. These models play a critical role in guiding experimental design and improving biotechnological [63] and medical applications [64].

A central concept of metabolic modeling is the representation of metabolic networks as graphs, where nodes denote metabolites, and edges represent mostly enzymatically catalyzed reactions, each associated with a flux variable. The structure and stoichiometry of the metabolic network are expressed in the stoichiometric matrix S , where rows correspond to metabolites and columns to reactions. Each entry in S corresponds to a stoichiometric coefficient: negative for reactants (consumed) and positive for products (produced). The vector of fluxes defines the rate at which reactions proceed and determines the dynamic behavior of the network [60]. Two examples of metabolic networks are given in Figure 1.4.

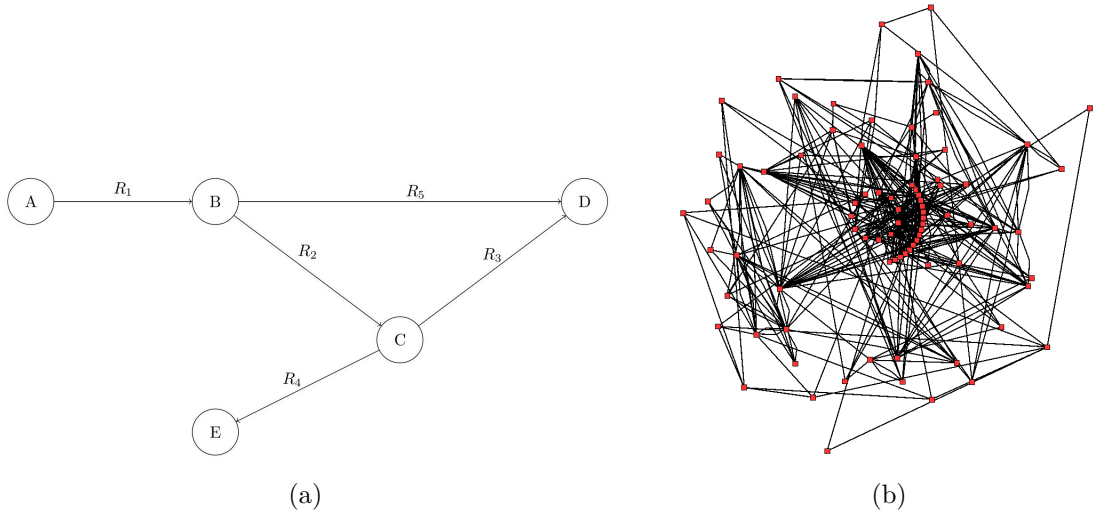


Figure 1.4: Metabolic Networks: (a) simplified representation of a (metabolic) network; (b) representation of the citric acid cycle in *Arabidopsis thaliana*, with enzymes and metabolites as red nodes and their interactions as connecting lines [65].

The respective stoichiometric matrix of Figure 1.5.a would be:

$$S = \begin{array}{c} \begin{array}{ccccc} & R_1 & R_2 & R_3 & R_4 & R_5 \\ \begin{array}{l} A \\ B \\ C \\ D \\ E \end{array} & \begin{bmatrix} -1 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & -1 \\ 0 & 1 & -1 & -1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array} \end{array}$$

Metabolic networks can be modeled at different levels of detail, from large-scale, genome-wide reconstructions using constraint-based approaches to more detailed, mechanistic models involving kinetic information [66]. Two main modeling approaches are commonly used: Constraint-based and kinetic modeling.

These approaches are complementary and can be used in combination to uncover complex system behaviors, explaining regulatory patterns, and building hypotheses for experimental validation [66]. Constraint-based modeling is well-suited for large-scale systems because it does not need kinetic parameters and focuses instead on steady-state fluxes. However, it cannot capture dynamic changes. Kinetic modeling describes time-dependent behavior, but is limited to smaller networks due to data tightness and computational complexity [66].

Stoichiometric models are easy to use and understand, making them popular among many researchers, but capturing dynamic or regulatory behaviors often requires more knowledge and detailed data. Metabolic modeling can range from simple mathematical methods to more advanced approaches. The next section introduces, among others, flux balance analysis (FBA), a common method used to study steady-state metabolic behavior in large networks [66].

1.5.3 Flux balance analysis

Flux Balance Analysis (FBA) is a mathematical framework that is used to analyze the flow of metabolites through a metabolic network [9]. Unlike kinetic models that require detailed information on the enzymatic rate, FBA is a constraint-based approach that relies on stoichiometric relationships [67]. It has become a foundational tool in systems biology, especially for studying genome-scale metabolic network reconstructions of various organisms. These reconstructions integrate known metabolic reactions and the corresponding genes that encode their associated enzymes [68], [69].

FBA operates under the assumption of a steady state, which means that the rate of metabolite production is equal to the rate of consumption, and no accumulation occurs over time within the cell. This principle leads to a system of linear equations based on mass balance constraints. Central to the formulation is the stoichiometric matrix S (see Section 1.5.2), which encodes the relationships between metabolites and biochemical reactions. The method does not rely on kinetic parameters, instead, it imposes constraints derived from reaction stoichiometry and known physiological limits (e.g., uptake rates, enzyme capacities). FBA defines feasible flux distributions (sets of reaction rates) that satisfy these constraints [9].

To extract biologically meaningful predictions from this solution space, FBA requires the specification of an objective function, typically a linear combination of fluxes. A common choice is the maximization of biomass production, which serves as a representation of the growth rate of the organism [70].

Mathematically, FBA is formulated as a linear programming (LP) problem:

$$\begin{aligned} \max_v \quad & c^T v \\ \text{subject to} \quad & Sv = 0, \\ & v_L \leq v \leq v_U \end{aligned} \tag{25}$$

where S is the stoichiometric matrix describing the relationship between metabolites and reactions, v is the flux vector representing the rate of biochemical reactions, $c^T v$ is the objective function, v_L and v_U are the lower and upper bounds of the reaction fluxes.

The result of solving this optimization problem is a flux distribution v that optimally achieves the biological objective under the given constraints [9].

FBA has been applied in numerous contexts and is a powerful method, but it has several limitations. The steady-state assumption does not reflect dynamic changes in cellular metabolism, and nutrient uptake fluxes are often predefined rather than dynamically adjusted based on extracellular metabolite concentrations [26].

1.5.4 Dynamic flux balance analysis

Dynamic flux balance analysis (dFBA) is an extension of FBA that enables the simulation of time-dependent behavior in response to environmental changes [71]. While FBA assumes that all metabolite concentrations remain constant over time, dFBA keeps this steady-state assumption for intracellular metabolites, but allows extracellular metabolite concentrations to change dynamically [72].

dFBA is based on the understanding that intracellular metabolic processes typically operate on a much faster timescale than extracellular dynamics. This separation of timescales allows the intracellular network to be treated as before in a pseudo-steady state, while the extracellular environment is modeled using ordinary differential equations (ODE) that describe substrate uptake, product secretion, and biomass growth [72].

Mathematically, dFBA uses the same network-based representation as FBA, with a stoichiometric matrix $S \in \mathbb{R}^{m \times n}$, where m is the number of metabolites and n the number of reactions. To capture both intracellular and extracellular dynamics, the stoichiometric matrix is extended to include both internal (intracellular) and external (extracellular) metabolites [25]:

$$S = \begin{bmatrix} S_{\text{int,int}} & S_{\text{int,ext}} \\ 0 & S_{\text{ext,ext}} \end{bmatrix} \tag{26}$$

The steady-state assumption is imposed only on the internal metabolites $S_{\text{int}}v(t) = 0$, while the extracellular metabolite dynamics are governed by an ODE:

$$\frac{dz(t)}{dt} = f(v(t), z(t), X(t)), \quad (27)$$

where $z(t)$ denotes the concentrations of extracellular metabolites, $v(t)$ is the time-dependent flux vector obtained from FBA, $X(t)$ is the biomass concentration, and $f(\cdot)$ defines the rates of change of extracellular metabolites [72].

dFBA couples stoichiometric constraints with dynamic mass balances by using FBA-derived fluxes to inform the time evolution of the extracellular environment. In this framework, uptake and secretion fluxes are computed from FBA and serve as inputs to the ODE system [73]. This makes dFBA particularly suitable for simulating batch and fed-batch cultures, where nutrient concentrations and biomass change over time [74].

There are two main approaches to implementing dFBA:

- (i). **Static Optimization Approach (SOA):** This method divides the simulation time into discrete intervals. At each time step, a linear programming problem is solved to determine the fluxes, and the resulting ODEs for biomass and extracellular metabolites are integrated forward using methods such as Euler integration [25], [75]. While conceptually simple, this approach can become computationally expensive or unstable for stiff systems that require small time steps [75].
- (ii). **Dynamic Optimization Approach (DOA):** Instead of solving separate LPs over time, this approach formulates the entire dFBA problem over a continuous time horizon as a non-linear programming problem. The time domain is discretized using methods like collocation or finite differences, converting differential equations into algebraic constraints. All fluxes, concentrations, and biomass profiles over time become decision variables. This allows global objectives, such as maximizing biomass or product yield, to be directly incorporated into optimization [25], [75].

1.5.5 Dynamic control flux balance analysis

Dynamic control flux balance analysis (dcFBA) extends the capabilities of traditional dynamic flux balance analysis (dFBA) by reformulating its bi-level optimization structure into a single-level nonlinear programming problem [76]. This reformulation improves computational efficiency and facilitates the solution of complex bioprocess optimization tasks [72].

In conventional dFBA, a dynamic simulation is coupled with a sequence of steady-state FBA problems, solved at each time step to simulate cellular metabolism over time. A widely used method to determine the intracellular flux distribution within these FBA subproblems is parsimonious flux balance analysis (pFBA), which identifies an efficient flux profile that satisfies biological constraints while minimizing unnecessary cellular effort [77].

pFBA is typically implemented as a two-step procedure. First, a primary cellular objective, such as biomass or product synthesis, is maximized. Then, among all solutions that achieve this maximum, the solution with the minimum total flux is selected. This reflects the biological principle of resource efficiency and can be formally described as [78]:

$$\begin{aligned}
& \max_{v^j} \quad cv^j + \frac{\epsilon}{2}v^j \quad v^j \\
& \text{s.t.} \quad Sv = 0 \\
& \quad v_L \leq v \leq v_U.
\end{aligned} \tag{28}$$

Here, S is the stoichiometric matrix, v represents the flux distribution, c is a weight vector defining the biological objective, ϵ a small value and v_L, v_U are reaction-specific flux bounds [76]. This approach has been widely used in systems biology, including genome-scale metabolic networks [77].

dcFBA enhances this framework by embedding the necessary optimality conditions, which are the KKT conditions of the inner pFBA problem, directly in the dynamic optimization formulation [72]. This results in a single-level NLP problem that captures both system dynamics and metabolic optimality [76]:

$$\begin{aligned}
& \max_u \quad F(\cdot) \\
& \text{s.t.} \quad Sv_j = 0 \\
& \quad v_j^L(u_j) \leq v_j \leq v_j^U(u_j) \\
& \quad \nabla_{v_j} \mathcal{L}_j(v_j, \lambda_j, \alpha_j^L, \alpha_j^U) = 0 \\
& \quad \alpha_j^L, \alpha_j^U \geq 0 \\
& \quad \alpha_j^L \circ (v_j - v_j^L) = 0 \\
& \quad \alpha_j^U \circ (v_j^U - v_j) = 0, \quad j \in \{1, \dots, n_{FE}\}
\end{aligned} \tag{29}$$

In this formulation:

- The index $j \in \{1, \dots, n_{FE}\}$ explicitly denotes discrete time intervals or finite elements over which the dynamic system is evaluated.
- $\mathbf{v}_j \in \mathbb{R}^n$: flux vector at finite element or time segment j
- $\mathcal{L}_j$: Lagrangian of the local pFBA problem at j
- $\lambda_j \in \mathbb{R}^m$: multipliers for the mass-balance constraints $Sv_j = 0$
- $\alpha_j^L, \alpha_j^U \in \mathbb{R}^n$: multipliers for flux bounds
- $\mathbf{v}_j^L, \mathbf{v}_j^U$: time-varying lower and upper bounds
- u_j : control variables

By converting the nested FBA optimization into algebraic constraints through KKT conditions, dcFBA eliminates the need to solve a separate FBA problem at each time

step, significantly improving computational tractability. For instance, in a case study on plasmid DNA production, dcFBA achieved at least a fourfold speed-up over standard dFBA [76].

In dcFBA, control variables represent decision parameters that can be manipulated to steer the system toward desired bioprocess objectives, such as maximizing productivity, titer, or yield. These variables often include feeding profiles, initial concentrations, oxygen transfer rates, or environmental parameters like the speed of the stirrer [76]. The overall goal is to determine the optimal trajectories of these variables over time to achieve the best possible process performance.

To efficiently solve the dynamic optimization problem, dcFBA employs orthogonal collocation on finite elements to discretize the system dynamics. The resulting NLP problem, which includes both the discretized mass balance equations and the embedded FBA KKT conditions, can then be solved using standard large-scale optimization algorithms [72].

Several enhancements have been proposed to improve robustness and convergence:

- **Finite element length reformulations** improve convergence speed during collocation-based discretization [76],
- **Penalty terms** in the objective function help enforce Linear Independence Constraint Qualification (LICQ) in cases where constraint gradients become degenerate [72],
- **Surrogate models**, including deep neural networks, are being explored to approximate the FBA step, enabling online or real-time control strategies with significantly reduced computational overhead [79].

In addition, dcFBA provides a structured framework for handling more complex challenges, such as parameter estimation, state estimation, and the integration of prior experimental knowledge into the model. This positions dcFBA as a highly flexible and scalable modeling approach suitable for advanced bioprocess design and control.

Chapter 2

Methodology

2.1 Reference Models

Two reference models are considered for the simulations, they are implemented similarly but vary in size and complexity. Despite these differences, the fundamental formulation and the underlying principles of both models are comparable. Given that the performance of various algorithms can be influenced by problem size, the smallest available problem size was selected as a reference, and the next larger one as the second reference. To maintain consistency with the original implementations, the following names are assigned:

- **DCO-BUT** (original file name: `ext_but043`) – based on the 2,3-butanediol production model tested in [80].
- **DCO-PDN** (original file name: `P_v2024`) – based on the plasmid DNA production model tested in [76].

This selection allows us to assess whether a simple increase in size and complexity leads to minor variations in performance or introduces significant computational challenges. While these models are already large and complex from a mathematical perspective, they are relatively small compared to biological models. To assess the robustness of the solver configuration strategies, the tests were extended to two significantly larger models `lac_005` and `slim_998`. An overview of all models is given in Table 2.1.

Since the models are implemented similarly, only one of them (**DCO-PDN**) is described in detail. All models are implemented using the Julia programming language. The specific packages and tools, along with their respective versions, are detailed in the supplementary table. For now, we take a closer look at the mathematical formulation. The model describes a fed-batch fermentation process optimized using dynamic control flux balance analysis (dcFBA), based on the formulation presented in [76]. The general idea of this approach is discussed in Section 1.6.3.

Model	# Reactions	# Metabolites	# Finite Elements	# Variables
DCO-PDN	470	454	30	18
DCO-BUT	95	75	30	17
slim_998	2714	1878	20	14
lac_005	2583	1805	20	14

Table 2.1: Overview of the dcFBA models and their dimensions. # Reactions shows the number of biochemical reactions in the metabolic network, while # Metabolites refers to the number of intracellular species in the stoichiometric matrix. # Finite Elements indicates the number of time intervals used for discretization, and # Variables represents the total number of decision variables in the resulting nonlinear program, including fluxes and concentrations across all time points.

2.1.1 Model formulation

The model discussed in this section is based on the work of [76]. To ensure consistency, the names and definitions of variables and constraints, as well as the overall structure, follow those presented in [76]. The optimization problem considers key state variables such as metabolite concentrations, reactor volume, and metabolic fluxes. The finite element discretization is used to describe dynamic changes over time.

State and Decision Variables

The model optimizes the concentrations of metabolites and reactor volume over time, represented as $N_{m,i,j}$ for the amounts of metabolites and $V_{i,j}$ for reactor volume. The feed rate $F[x_i]$ and process time t_{end} are control variables that regulate the process dynamics. The model also accounts for flux variables $q_{r,i}$, representing reaction rates at finite elements, and KKT multipliers λ_{S_v} ensuring steady-state constraints are met. Complementary slackness conditions enforce active or inactive metabolic constraints through variables α and *slack*.

Objective Function

The objective function is designed to maximize the production efficiency of the fed-batch system while ensuring constraint satisfaction. It is formulated as a balance between maximizing product concentration per unit of time and penalizing constraint violations. It is formulated as follows:

$$\max(\psi_1 F() + \psi_2 P). \quad (1)$$

The first term in the objective function promotes the maximization of the final product yield per time unit. The second term represents a penalty for violating complementary slackness conditions.

The slackness penalty term is given by:

$$P := \frac{1}{nFE} \sum_{j=1}^{nFE} \left(\frac{1}{nAB} \sum_{i=1}^{nAB} s_{lb,i}^j + \sum_{i=1}^{nAC} s_{ub,i}^j \right), \quad (2)$$

where nFE represents the number of finite elements, and nAB and nAC account for the number of lower and upper bound complementary slackness conditions, respectively. This ensures that constraint violations are penalized uniformly throughout the time horizon, encouraging feasibility while optimizing productivity.

Constraints

The optimization problem is subject to multiple constraints governing metabolic and process dynamics. The dcFBA framework is built around repeatedly solving an FBA subproblem at each finite element (FE) or time step to compute the optimal intracellular flux distribution. This subproblem is defined by the steady-state mass balance equations:

$$Sv = 0, \quad (3)$$

where S is the stoichiometric matrix and v the vector of metabolic fluxes. Fluxes must also satisfy predefined bounds:

$$v^{\min} \leq v \leq v^{\max}. \quad (4)$$

To ensure optimality at each FE, the dcFBA model applies the KKT conditions to the FBA subproblem. This transforms the original bi-level formulation into a single-level nonlinear programming (NLP) problem, making it solvable with common solvers. The KKT system includes:

- **Stationarity:**

$$\nabla_v \mathcal{L} = 0, \quad (5)$$

ensuring that the flux distribution corresponds to a local optimum of the FBA subproblem's Lagrangian.

- **Non-negativity of duals:**

$$\lambda^{\min} \geq 0, \quad \lambda^{\max} \geq 0, \quad (6)$$

where $\lambda^{\min}$ and $\lambda^{\max}$ are the dual variables associated with the lower and upper flux bounds.

- **Complementary slackness:**

$$\lambda^{\min}(v - v^{\min}) = 0, \quad \lambda^{\max}(v^{\max} - v) = 0, \quad (7)$$

ensuring feasibility and binding conditions at the flux limits.

These KKT conditions are enforced for each time interval, allowing the dcFBA model to dynamically adjust the flux distributions as a function of changing system states and control variables. The resulting fluxes influence metabolite concentrations, which in turn affect the optimal flux configuration at the next FE. This iterative coupling captures the dynamic behavior of the bioprocess over time and supports the identification of optimal control strategies.

To enable flexible time discretizations, finite elements are assigned relative lengths τ_i constrained by

$$\tau_{\min} \leq \tau_i \leq \tau_{\max}, \quad (8)$$

where the actual finite element durations Δt_i are computed as

$$\Delta t_i = \frac{t_{\text{end}}}{n_{\text{FE}}} \tau_i, \quad (9)$$

ensuring that the total process time t_{end} is fully covered:

$$\sum_{i=1}^{n_{\text{FE}}} \Delta t_i = t_{\text{end}}. \quad (10)$$

This allows adaptive but controlled variation of the time step lengths, helping to maintain numerical stability.

Beyond intracellular constraints, the model enforces user-defined process constraints:

$$g(v, x, u) \leq 0, \quad h(v, x, u) = 0, \quad (11)$$

which reflect biological, physical, and technical limitations. These include restrictions on the accumulation of metabolites, feed rate dynamics, volume, and other operational limits critical for process feasibility.

The model is implemented in Julia using the JuMP modeling language and solved with IPOPT and the linear solver MA27 [76].

2.2 IPOPT - Solver Optimization

2.2.1 Testing strategy

The IPOPT solver offers a wide range of options designed for different needs depending on the problem. In total, there are almost 200 options grouped into 14 categories. Furthermore, IPOPT supports 11 different linear solvers, each with 5 to 10 specific parameters that can be further configured. This large amount of parameters presents opportunities and also challenges. On one hand, these options offer great potential for performance improvements. On the other hand, they make systematic testing difficult because of the enormous number of possible combinations.

To address this, a structured and organized approach to explore the effect of these solver options on the optimization models was designed. The strategy was to start by testing solver options individually to assess their isolated impact on solution quality and runtime. Therefore, options were grouped into categories, such as Termination, Initialization, and Warm-Start, for example, and incrementally applied to the original configuration (see Table 2.2) that had already successfully solved the problem. If an option appeared promising, further tests in combination with other settings were conducted.

This procedure was applied primarily to the two reference models `DCO-BUT` and `DCO-PDN`. For each model, 100 simulations were performed. Most of these experiments involved varying a single `IPOPT` option at a time in addition to the original setting. In 33 of the 100 simulations, we tried combinations of options within a single category. We also defined 10 different configurations combining the best-performing options from all categories, resulting in 110 total simulations per reference model.

The simulations were repeated a second or third time only for configurations that showed initial improvement. For each new simulation run, Julia was restarted in the terminal to avoid any potential bias from cached solver states or previously stored variable values. Without restarting, solving appeared to complete in just a few seconds, suggesting that previous solutions were being reused, which could lead to biased results. Therefore, restarting ensured that each run was initialized from a clean state. In the supplementary tables, only one result is shown per option to keep the data manageable, especially since repeated runs produced results in the same dimension.

To test the stability and generalization of these configurations, the analysis was extended to larger models `lac_005` and `slim_998`. Due to their increased size and significantly longer runtime, only 5 simulations were conducted for `lac_005` and `slim_998`. These experiments used only the most promising solver configurations identified in previous tests.

Option	DCO-PDN	DCO-BUT	lac_005	slim_998
warm_start_init_point	yes	yes	yes	yes
print_level	5	5	5	5
linear_solver	ma27	ma27	ma27	ma27
max_iter	Int(1e5)	Int(1e5)	—	Int(1e4)
tol	1e-4	1e-6	1e-4	1e-4
acceptable_iter	15	50	5	15
acceptable_tol	1e-2	1e-6	1e-2	1e-1

Table 2.2: Initial `IPOPT` options used across different problem configurations

To support this large-scale testing effort, we adapted the original Julia-based script to simplify the modification of solver options and to automatically store the selected configurations in the summary output file generated for each simulation. Each simulation is associated with a unique directory name, and results are tracked using corresponding

`.xlsx` files. The directory name acts as a primary key, similar to a relational database. One file contains meta-information about the simulation setup (options used, categories, and directory name), which was created manually, while the second file stores the actual performance data. The summary files include key metrics such as:

- Directory name
- Termination status
- Time elapsed
- Objective value
- Normalized complementary slackness
- Feed and growth rates
- Final product concentration
- Final biomass
- Final substrate concentration S_0

These values were automatically extracted from the IPOPT summary outputs using a Python script. This setup ensures that all data can be easily traced, verified, and compared.

The primary evaluation metric was runtime reduction, as improving solver efficiency was the goal. Throughout the experiments, we ensured that solution quality, especially final biomass, final product, and S_0 , remained consistent and did not differ significantly from the reference results. In most cases, the quality of the solution was stable. A full overview of all simulations, including added options, runtime, and termination status, is available in the Supplements section. Other values were excluded from the final analysis.

2.3 Alternative Solver MadNLP

To explore other options, the goal was to identify suitable alternatives for the given optimization problems. After some research, we came across a blog post in the Julia development forum [81] suggesting that MadNLP is a promising and fast alternative to the IPOPT solver. The article compared the IPOPT solver with the Ma27 linear solver on CPU (which is what is used in the original implementations) to the MadNLP solver using ExaModels on GPU. The results were encouraging, making it a valid choice for testing our examples. Also, it is possible to run a combined approach on the JuMP interface.

2.3.1 Testing strategy

The following workflow was developed after encountering challenges with initial approaches that did not perform as expected. The main objective was to achieve stable performance

and correct implementation on the CPU before transitioning to GPU programming. This allowed potential errors to be addressed early in the testing and development process.

Switching from `JuMP` to `ExaModels` introduced specific challenges due to differences in syntax and overall system behavior. Since `ExaModels` still has limited documentation, a more careful and detailed testing strategy was necessary to avoid syntax errors and unexpected issues.

The process started with testing the existing `JuMP` model using the `MadNLP` solver. This step helped establish a baseline and confirm that the solver could handle this type of optimization problem. As expected, some adjustments were required to get the model running smoothly, as will be discussed in the following section. After establishing a working baseline, the `JuMP` model was incrementally converted into `ExaModels`. Variables and constraints were added one at a time, with the model being tested after each addition. This incremental approach made debugging more efficient and allowed early detection and resolution of potential errors before proceeding further. Next, a hybrid and highly experimental approach was tested by integrating both `JuMP` and `ExaModels`. The primary goal at this stage was to determine whether the hybrid model could produce consistent and valid results, given the experimental nature of this combination.

Once the models were running correctly and producing consistent results on the CPU, the plan involved transitioning to GPU-based computations. This final step aimed to leverage GPU parallelism to improve computational efficiency and assess whether stable performance could be achieved on the GPU at this stage of development.

2.4 Model set-up

Setting up the model using the two different frameworks, with a particular focus on transforming the existing `JuMP` model to `ExaModels`, was a key aspect of this thesis. In the following, we look into the programming-related aspects of this transition. To outline the structural differences and the changes required, specific code snippets are included. These snippets are not complete and do not represent fully functional models. Instead, they illustrate selected variables and constraints to provide a technical overview of how the different models are structured. Based on that, we can highlight the modifications that were necessary and which certain variables and constraints presented challenges during the transition and needed more careful rewriting.

Most modifications mentioned in the upcoming section are covered in the official documentation and guidelines in [46] and [53]. However, certain modifications discussed here are from additional testing and are not part of the official documentation.

2.4.1 JuMP

The reference models are implemented using JuMP and solved with IPOPT. As discussed in Section 1.4.1, a JuMP model closely reflects the mathematical structure of an optimization problem, making it highly intuitive to work with. The code snippet below is taken from the smallest example problem, DCO-BUT, and highlights key variables such as metabolite amounts, fluxes, and feed rates. It also includes specific constraints that later proved challenging when adapting the model to ExaModels. Note that the objective function shown here is simplified. It excludes the penalty term of the slack variables to reduce programming complexity. Additionally, exact numerical values, the matrix *colmat*, and solver options have also been left out, because they are not essential to understand the structure of the model.

```
using JuMP, IPOPT
...

# Create a JuMP model with the IPOPT solver
model = Model(IPOPT.Optimizer)

#Variables
@variable(m, N[1:nN, 1:nFE, 1:nCP]) # metabolite amounts
@variable(m, Ndot[1:nN, 1:nFE, 1:nCP]) # dN/dt
@variable(m, q[ 1:nR, 1:nFE ] # FBA fluxes
...
@variable(m, t_end) # total process length
@variable(m, rtFE[1:nFE]) # relative length of finite elements
@variable(m, F[1:nFE]) # feed rate

# Uptake Rates
@NLexpression(m, q_G[i=1:nFE], Fx[i]*c_G)
@NLexpression(m, tFE[i=1:nFE], t_end/nFE*rtFE[i])
...

# NL objective Function
@NLobjective(m, Max, N[4,end,end])

# NL Process constraints (1-3)
@NLconstraint(m, NLc_q_G[i=1:nFE], -q[glu,i] - q_G[i] == 0)

@NLconstraint(m, NLc_q_A1[i=1:nFE], q[aon,i] + 5.0649596206 -
    36.6856368564 * q[xxx,i] >= 0)

@NLconstraint(m, NLc_t_end, sum(tFE[i] for i in 1:nFE) == t_end)
...
# Collocation equation 1
@NLconstraint(m, coll_N[l=1:nN, i=2:nFE, j=1:nCP], N[l,i,j] == N[l
    ,i-1,nCP]+tFE[i]*sum(colmat[j,k]*Ndot[l,i,k] for k in 1:nCP))
...
```

```

# Other Constraints
@constraint(m, m2[i=1:nFE, j=1:nCP], Ndot[2,i,j] == q[xxx,i]*N[2,i
,j]) #sulfate

@constraint(m, m3[i=1:nFE, j=1:nCP], Ndot[3,i,j] == q[aon,i]*N[2,i
,j]) # product
...

# Solve the model
optimize!(model)

```

2.4.2 ExaModels

To modify the JuMP model, we have two options. First, we explore the approach where ExaModels is used only for setting up the model. The following code snippet shows the implementation of this approach for the DCO-BUT model using ExaModels:

```

using JuMP, ExaModels, MadNLP
...

# Create a ExaCore object
c = ExaCore()

#Variables
N = variable(c, nN, nFE, nCP; start = (NO[k] for k=1:nN), lvar =
0.0)

Ndot = variable(c, nN, nFE, nCP)

q = variable(c, nR, nFE; lvar = (vlb[mc] for mc=1:nR), uvar = (vub
[mc] for mc=1:nR))
...
t_end = variable(c, lvar = t_min, uvar = t_max)
rtFE = variable(c, nFE; start = 1.0, lvar = 0.8, uvar = 1.2)
Fx = variable(c, nFE; lvar = 0.0, uvar = 1.0)

# NL objective Function
objective(c, Max, N[4,end,end])

#Process constraints (1-3)
constraint(c, -q[glu, i] - Fx[i] * c_G for i=1:nFE)
constraint(c, q[aon, i] + 5.0649596206 - 36.6856368564 * q[xxx, i]
for i=1:nFE)
constraint(c, t_end/nFE * rtFE[i] - t_end for i=1:nFE)
...
# Collocation equation 1

```

```

for l in 1:nN, i in 2:nFE, j in 1:nCP
    constraint(m, N[l,i,j] - (N[l,i-1,nCP] + (t_end/nFE*rtFE[1]) *
        sum([colmat[j,k] * Ndot[l,i,k] for k in 1:nCP])))
end
...
# Other Constraints
for i in 1:nFE j in 1:nCP
    constraint(c, Ndot[2, i, j] - ((q[xxx, i] * N[2, i, j])))
end

for i in 1:nFE for j in 1:nCP
    constraint(c, Ndot[3, i, j] - ((q[aon, i] * N[2, i, j])))
end
...

# Create ExaModels
m = ExaModels(c)

# Solve the model
results = MadNLP(m)

```

Several differences can be observed between JuMP and ExaModels. Converting a JuMP model into a ExaModels setup requires more modifications than just the obvious ones. First, instead of defining a model object as in JuMP, an ExaCore object must be created. The overall structure remains similar, with variables, an objective function, and constraints, however, their formulation differs significantly.

Unlike JuMP, ExaModels does not use the @ symbol to define model components. Instead, variables are assigned using an equality sign, where the lower and upper bounds are specified with `lvar` and `uvar`, respectively. To set a start value, the keyword `start` needs to be used within the variable definition. This was a challenge since the original JuMP code used integrated functions to set start values as well as lower and upper bounds, which are excluded in the code snippet of the JuMP model. The definition of the objective function follows a similar approach, though ExaModels currently does not allow explicit specification of whether the function should be minimized or maximized.

Another key distinction is in the constraint formulation. In ExaModels, constraints are not explicitly categorized as linear or non-linear, instead, the solver automatically detects their nature. This is also the case in the latest version of JuMP, which no longer differs between linear and nonlinear constraints. This aspect is particularly relevant when considering the hybrid approach that combines ExaModels and JuMP.

One of the main challenges in implementing more complex constraints in ExaModels is that it does not support explicit inequality or equality constraints. If a constraint equals zero, it does not need to be stated explicitly, as ExaModels handles this automatically. This can be observed by comparing the first constraint in both code snippets. Similarly, the use of the sum operator is also handled implicitly and does not need to be stated, as

demonstrated in constraint 3. However, in our case, not including the `sum()` function in the collocation constraint leads to an error. Therefore, it must be explicitly stated here.

However, not all constraints are equal to zero. To address this, one approach is to introduce a slack variable to transform the constraint into an equality. While this is technically feasible, it would increase the number of variables and is often not wanted. As an alternative, the constraints can be rewritten directly. For example, in constraint 3, instead of expressing

$$\sum_{i=1}^{nFE} \frac{t_{\text{end}}}{nFE} \cdot rtFE[i] = t_{\text{end}}$$

as in JuMP, reformulate it as

$$\left(\sum_{i=1}^{nFE} \frac{t_{\text{end}}}{nFE} \cdot rtFE[i] \right) - t_{\text{end}} = 0.$$

Although this transformation may seem unconventional, it is necessary because writing constraints in the standard JuMP format would result in an error, as inequality or equality signs are not supported by the `ExaModels` API. Another missing element when comparing the two code snippets is the `@NLexpression` object, which is also no longer supported. As a workaround, the corresponding expressions were directly integrated into the constraints themselves, as shown in the constraints 1 and 3.

One unresolved issue is how to handle inequality constraints. For example, constraint number 2 is intended to be greater than or equal to zero. However, `ExaModels` does not support inequality constraints at all. Since no viable workaround has been found, this constraint is currently enforced as an equality to zero in the test simulations.

This behavior and modifications are not explicitly stated in the official documentation, rather, the results gained through tests and experiments by analyzing the error messages encountered, especially when transforming constraints.

As previously mentioned, an approach combining `ExaModels` and JuMP is available. However, it is still considered experimental. While it is expected to function correctly, this approach involves certain performance trade-offs. In particular, model creation and automatic differentiation performance may not be as efficient as when utilizing the native `ExaModels` API at the moment. But since we already have a JuMP model, this approach made us curious. The implementation would be the following:

```
using JuMP, ExaModels, MadNLP
...

# Create a JuMP model
jm = Model()

#Variables
@variable(m, N[k=1:nN, i=1:nFE, j=1:nCP], start = N0[k],
          lower_bound = 0.0)
```

```

@variable(m, Ndot[1:nN, 1:nFE, 1:nCP])
@variable(m, q[mc=1:nR, i=1:nFE], lower_bound = vlb[mc],
    upper_bound = vub[mc])
...
@variable(m, t_end, lower_bound = t_min, upper_bound = t_max)
@variable(m, rtFE[i=1:nFE], start = 1.0, lower_bound = 0.8,
    upper_bound = 1.2)
@variable(m, Fx[1:nFE], lower_bound = 0.0, upper_bound = 1.0)

# NL objective Function
@objective(jm, Max, N[4,end,end])

# Process constraints
@constraint(jm, NLc_q_G[i=1:nFE], -q[glu,i] - q_G[i] == 0)

@constraint(jm, NLc_q_A1[i=1:nFE], q[aon,i] + 5.0649596206 -
    36.6856368564 * q[xxx,i] >= 0)

@constraint(jm, NLc_t_end, sum(tFE[i] for i in 1:nFE) == t_end)
...
# Other Constraints
@constraint(jm, m2[i=1:nFE, j=1:nCP], Ndot[2,i,j] == q[xxx,i]*N[2,
    i,j])

@constraint(jm, m3[i=1:nFE, j=1:nCP], Ndot[3,i,j] == q[aon,i]*N[2,
    i,j])
...

# Solve the model as an ExaModels using MadNLP
set_optimizer(jm, () -> ExaModels.MadNLP0ptimizer())
optimize!(model)

```

For this approach the original JuMP model first needs to be updated with its newest syntax. ExaModels only supports scalar objectives and constraints defined using `@constraint` and `@objective`. Older syntax, such as `@NLconstraint` and `@NLobjective`, is not compatible with the framework [53]. If you compare it with the original setup, we do not notice a lot of differences. The integrated functions to set start values, lower and upper bounds used in the original implementation seemed not to be supported, and therefore need to be set within the variable definition with `start`, `lower_bound`, and `upper_bound`. Since ExaModels is still in its development stage, such limitations were expected. Moreover, the issues discussed here may no longer be relevant by the time this thesis is published, as future updates could address them.

2.4.3 GPU Integration

The initial implementation operates on CPU. To enable GPU execution, further modifications are necessary. Since `JuMP` does not provide direct GPU support, the integration of `ExaModels` becomes essential for this. To extend the implementation to GPU, either `ExaModels` alone or the experimental alternative approach can be employed.

The key package for GPU programming in this context is `CUDA` [82], which functions as the primary interface for leveraging NVIDIA GPUs in Julia. This package supports different levels of usage, from high-level array-based operations to low-level programming with custom CUDA kernels and direct access to CUDA APIs.

As discussed in more detail in Chapter 3, a fully functional setup was only achieved using the experimental approach. Therefore, we focus on presenting the necessary modifications to the previous code snippet. To enable GPU support, the packages `CUDA` and `MadNLPGPU` are additionally required:

```
using JuMP, ExaModels, MadNLPGPU, CUDA
```

The rest of the code stays the same as in the hybrid approach. For solving, we have two options, and the code needs to be adapted as follows:

```
# With Optimizer interface
set_optimizer(jm, () -> ExaModels.MadNLPOptimizer(CUDABackend()))
optimize!(jm)

# Alternative
em = ExaModels(jm; backend = CUDABackend())
result = MadNLP(em)
```

In the case of the alternative setup, which includes only `ExaModels`, two additional critical aspects were encountered when implementing GPU-based computations. These aspects turned out to be particularly challenging and caused considerable difficulties during the development process.

Scalar Indexing

Array operations, especially on GPU, must be carefully structured to avoid scalar indexing, which refers to accessing array elements individually using single indices. While this approach is common and efficient on CPU, it is inefficient on GPU, which are designed for large-scale parallel execution [83]. Scalar indexing can cause threads to run one after another or lead to inefficient data transfer between the CPU and GPU, which can significantly reduce performance.

In Julia, scalar indexing is disallowed by default in non-interactive execution environments such as scripts or applications to prevent these inefficiencies. It is still allowed in interactive sessions to make testing and debugging easier. If scalar indexing is needed, it can be enabled for specific parts of the code using the `@allowscalar` macro or the `allowscalar` do-block [84].

CuArrays

The most straightforward way to exploit the parallelism of GPU in Julia is suggested to express computations in terms of array operations. The package `CUDA` [82] provides the specialized array type `CuArray`, designed to run efficiently on GPU. `CuArray` implements the `AbstractArray` interface, allowing it to behave much like a standard Julia Array. This means `CuArray` can be created and manipulated using familiar syntax and functions, making GPU programming more accessible [85]. Beyond simple array construction, the real strength of `CuArray` lies in Julia’s higher-order array abstractions. Functions like `map` and `broadcast` accept user-defined operations and execute them parallel on GPU, often eliminating the need to write custom `CUDA` kernels manually [86].

2.5 Hard- and Software

The original implementation runs on the JupyterLab server provided by the Biochemical Network Analysis group, powered by a 128-core AMD EPYC 7542 CPU on an x86_64 architecture. For GPU simulations, the Jucuda server was used, also provided by the Biochemical Network Analysis group. It is important to specify the GPU model and `CUDA` version, as not all features are compatible across versions. A list of server specifications is given in Table 2.3.

Category	Details
System Information	
Operating System	Debian GNU/Linux 12 (Bookworm)
Kernel Version	6.1.0-31-amd64
CPU Architecture	x86_64
System Type	GNU/Linux
GPU and CUDA Setup	
GPU Model	2x Tesla V100S-PCIE-32GB
Total Memory	32 GB per GPU
GPU Driver Version	535.216.01
CUDA Toolkit Version	11.8
CUDA Runtime Version	12.6

Table 2.3: System and GPU Specifications

Chapter 3

Results and Discussion

3.1 IPOPT Option Testing

To investigate the impact of IPOPT configurations on solver performance, two implementations were analyzed, DCO-PDN and DCO-BUT, by systematically testing a wide range of solver options across various categories. These categories included especially:

- Barrier method options
- Initialization strategies
- Linear solver settings for MA27
- NLP options
- Termination criteria
- Warm-start options
- combinations of options

The primary evaluation criterion was runtime performance of the solver.

3.1.1 Reference Models

As shown in Table 2.2, the main difference in the default options between DCO-PDN and DCO-BUT lies in how strictly they enforce convergence. DCO-BUT uses a tighter overall tolerance (`tol` = $1e-6$) and acceptable tolerance ($1e-6$), allowing up to 50 acceptable iterations to ensure high solution accuracy. In contrast, DCO-PDN takes looser tolerances (`tol` = $1e-4$, `acceptable_tol` = $1e-2$) and permits only 15 acceptable iterations, prioritizing faster convergence over accuracy.

Starting the experiments with the smaller problem DCO-BUT, testing each category of solver options one by one, also varying their value and observing how they affected the time it took IPOPT to solve the problem. After running all tests, most of the chosen

settings either made the solver slower or had no noticeable effect. A full list of the results is given in the supplementary table. Only two categories clearly improved performance: Initialization and Warm-start.

To evaluate the performance impact of the DCO-BUT implementation, the percentage change in runtime was calculated for the best-performing configurations presented in Table 3.1. Negative values correspond to runtime reductions (performance improvements), while positive values indicate worse performance.

The best runtime reduction was achieved with `warmst1_1`, lowering runtime by 79%. Several other configurations, including `init_combi2`, `warmst3_1`, and `init2_3`, also performed well, each achieving reductions of nearly or over 50%. In contrast, two configurations led to considerable performance degradation: `init2_2` resulted in a more than 16-fold increase in runtime, while `init1_2` increased runtime by approximately 91%. These findings highlight that not all initialization methods are suitable, and inappropriate choices can severely degrade performance.

The file names of the implementations under discussion, along with the respective solver options added to the original configuration and their corresponding percentage change in runtime, are summarized for both implementations in Table 2.2.

File	Options Added	DCO-BUT	DCO-PDN
<code>init_combi2</code>	<code>bound_push => 0.002, bound_frac => 0.002</code>	-78.40%	-11.99%
<code>init_combi3</code>	<code>bound_push => 0.005, bound_frac => 0.005</code>	-53.03%	77.66%
<code>init1_1</code>	<code>bound_push => 0.002</code>	-39.00%	-30.45%
<code>init1_2</code>	<code>bound_push => 0.02</code>	90.70%	100.75%
<code>init2_2</code>	<code>bound_frac => 0.02</code>	1595.65%	-10.63%
<code>init2_3</code>	<code>bound_frac => 0.03</code>	-64.13%	128.99%
<code>warmst1_1</code>	<code>warm_start_bound_push => 0.002</code>	-79.33%	421.46%
<code>warmst1_2</code>	<code>warm_start_bound_push => 0.003</code>	-61.15%	10.11%
<code>warmst3_1</code>	<code>warm_start_bound_push => 0.002</code>	-75.88%	444.84%
<code>warmst3_2</code>	<code>warm_start_bound_push => 0.003</code>	-52.27%	701.18%
<code>warmst4_1</code>	<code>warm_start_slack_bound_frac => 0.002</code>	-6.09%	-4.68%
<code>warmst6_2_2</code>	<code>warm_start_bound_frac => 0.5, warm_start_bound_push => 0.002</code>	-43.25%	NaN

Table 3.1: Performance impact (% change in solve time) of adding the specified IPOPT options to the baseline options from Table 2.2.

The analysis was repeated using the DCO-PDN implementation, applying the same options to allow for direct comparison even if the original options setting differ (see Table 2.2). The results confirmed the same trend observed before: initialization and warm-start settings were again the most promising for improvements in runtime, while the remaining categories had little to no positive effect. Again focusing on the percentage change in runtime, detailed in Table 3.1. Both categories led to some noticeable reductions, with `init1_1` cutting solver time by over 30%. A few other configurations, such as `init2_2` and `init_combi2`, achieved modest gains around 10%. However, DCO-PDN showed less consistent performance than DCO-BUT, with several strategies, particularly `init1_2`, `init2_3`, and `init_combi3` substantially increased runtime.

Warm-start strategies generally performed poorly with DCO-PDN. With the exception of a minor improvement from `warmst4_1`, most settings, including `warmst1_1`, `warmst3_1`, and `warmst3_2` increased runtime in some cases by more than four times. Notably, `warmstart6_2_2`, which had shown strong results under DCO-BUT, resulted in numerical errors and is reported as NaN in the supplementary table. While some configurations lead to small improvements, the majority significantly degrade performance or even cause solver instability.

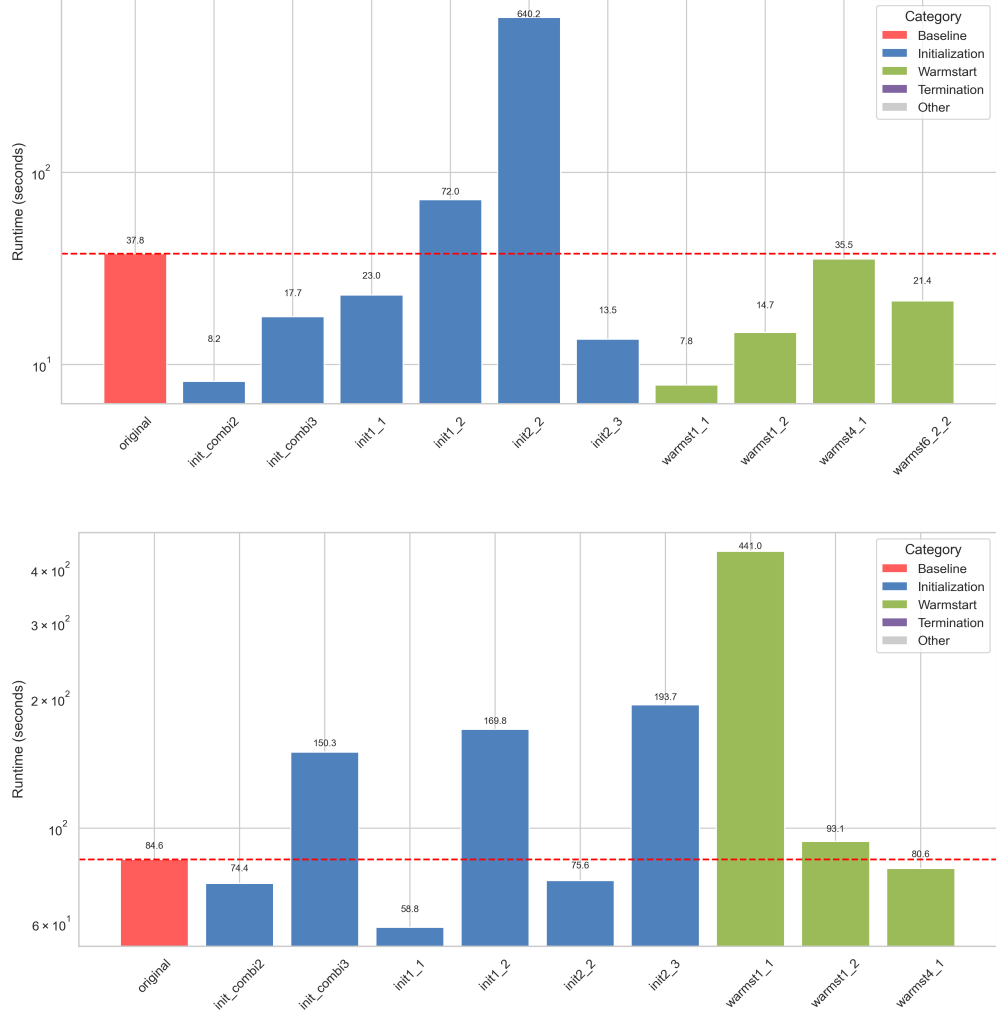


Figure 3.1: Runtime comparisons for the reference models DCO_BUT(top) and DCO-PDN (bottom) categorized by option type.

Some poor results are included on purpose in Figure 3.1, to show how different option settings can lead to very different outcomes. A few configurations are missing in DCO-PDN

but are included in DCO-BUT, as they caused numerical errors and therefore produced no comparable results. Since some options showed actual improvement, also combinations of initialization and warm-start strategies were tested, to see if using them together would lead to better performance than using each one on its own. As shown in Table 3.2, most combinations kept the benefits of the individual methods, and in some cases, even improved performance further.

The best overall result came from combining `init_combi2` with `warmst1_1`, which reduced runtime by nearly 80%, slightly better than each method alone. Similarly, combining `init2_3` with `warmst3_1` or `warmstart6_2_2` led to runtime reductions of almost 55% showing that these combinations worked particularly well together. More moderate improvements were seen with `init1_1` paired with `warmst1_2` and `warmstart6_2_2`. The weakest combination, though still helpful, was `init_combi3` with `warmst4_1`, resulting in a 37% reduction. This suggests that while both methods were effective on their own, combining them did not lead to much extra improvement.

Combination	change in runtime
<code>init_combi2 + warmst1_1</code>	-79.85%
<code>init_combi2 + warmst1_2</code>	-41.50%
<code>init_combi3 + warmst4_1</code>	-37.50%
<code>init1_1 + warmst1_2</code>	-32.19%
<code>init1_1 + warmst6_2_2</code>	-40.43%
<code>init2_3 + warmst3_2</code>	-40.13%
<code>init2_3 + warmst6_2_2</code>	-55.38%
<code>init2_3 + warmst3_1</code>	-54.74%

Table 3.2: Combined initialization and warm-start effects under DCO-BUT

Again, a few selected combinations of initialization and warm-start strategies were tested, to see if using them together would lead to better performance. Since DCO-PDN did not show many positive results overall, the tests were limited to just three combinations. However, unlike DCO-BUT, all combinations in DCO-PDN led to worse performance (see Table 3.3). For example, `init_combi2 + warmst4_1` increased runtime by nearly 7-fold, while `init1_1 + warmst4_1` and `init1_1 + init2_2` each caused more than a 5-fold slowdown, even though the individual components showed minor improvements on their own. These results suggest that combining strategies in DCO-PDN can actually make convergence harder or less stable.

The quality of the solutions was unaffected across all configurations. The majority of runs terminated with either a `LOCALLY SOLVED` or `ALMOST LOCALLY SOLVED` status. Main outcome metrics (objective values, final biomass, final product, growth rates etc.) remained stable throughout. The only other consistent finding was that all other tested

Combination	change in runtime
init1_1 + init1_2	497.57%
init1_1 + warmst4_1	501.64%
init_combi2 + warmst4_1	676.99%

Table 3.3: Combined initialization and warm-start effects under DCO-PDN

option categories had little to no positive effect on the process and, when they did have an impact, it was mostly negative in both implementations.

3.1.2 Larger Implementations

As mentioned earlier, both reference models are relatively small. To assess the robustness of the founded solver configurations, the testing was extended to two larger models.

In its default configuration, `lac_005` required approximately 973 seconds to converge to a locally optimal solution (see Figure 3.2 and Table ??). Additionally, solver progress was noticeably slow, with iteration steps showing delays. However, applying the selected initialization and warm-start configurations improved runtime performance. The `init1_1` setup reduced the runtime to 787 seconds while `init_combi2` achieved a further reduction to 557 seconds. Most notably, `warmst1_1` and `init_combi3` led to drastic improvements, lowering runtime to 215 and 219 seconds, respectively.

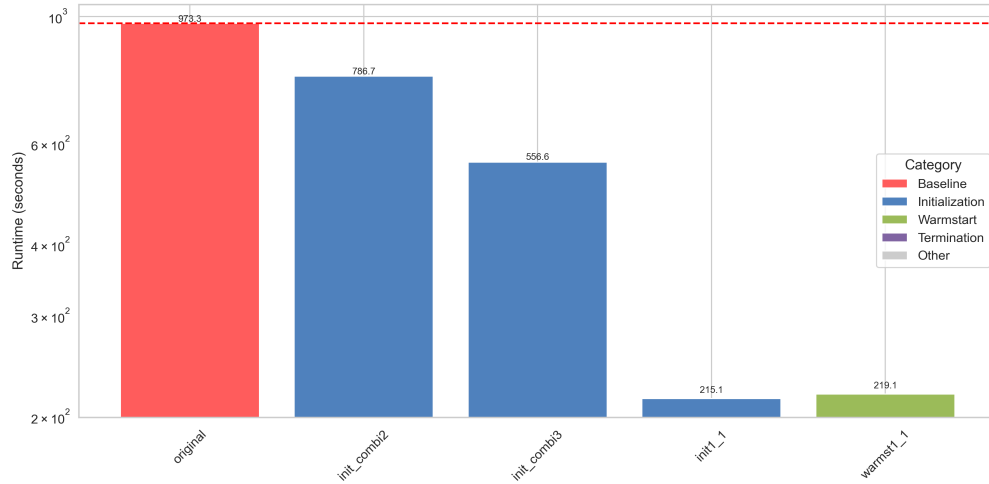


Figure 3.2: Comparison of runtime (`lac_005`) - Categorized by Option Type

This represents a runtime reduction of up to 78% compared to the original configuration, demonstrating that the positive effects of initialization and warm-start strategies gen-

eralize well to a larger problem. Despite the reduction in runtime, the solution quality was also stable across all tested configurations. All simulations either converged to a `LOCALLY SOLVED` or `ALMOST LOCALLY SOLVED` status, and other output metrics remained consistent.

Configuration	Termination Status	Time Elapsed (s)	Change
original	Locally_Solved	973.34	–
init1_1	Almost locally solved	786.73	–19.17%
init_combi2	Almost locally solved	556.57	–42.82%
warmst1_1	Locally solved	215.13	–77.90%
init_combi3	Locally solved	219.12	–77.49%

Table 3.4: Solver performance for different option configurations on `lac_005`

Unlike `lac_005`, the model `slim_998` did not show similar improvements. In the original setup, the solver did not find a solution and stopped after reaching the maximum number of iteration steps (10000). The overall runtime was very long, and the solver progressed slowly as shown in Table 3.5. This led to the decision to stop testing early and limited evaluation to the configurations that were also tested for `lac_005`. The following table does not include the percentage change because there was no solution found.

Configuration	Termination Status	Time Elapsed (s)
original	Iteration Limit	11008.62782
slim_998_init1_1	Locally infeasible	6950.797419
slim_998_initcombi2	Iteration Limit	9487.145253
slim_998_initcombi3	Iteration Limit	5286.980789
slim_998_warmst1_1	Iteration Limit	9726.503575

Table 3.5: Solver termination statuses and runtime for different option configurations on `slim_998`.

3.1.3 Discussion

We did not expect to find different optima, since the underlying optimization problem and solver strategy did not change, but probably achieve faster convergence with the same solution quality. Overall, the experiments confirm that convergence is strongly influenced by the initial guess, which aligns with findings reported in the literature [28], [87] and is also mathematically expected. The quality of the initial point affects solver performance, potentially causing slower convergence or numerical instability. Among all strategies tested, initialization and warm-start options had the most noticeable effect,

but not consistently across all models. While they improved convergence in one of the two reference models, they were less effective in the other one. Several factors could explain the lack of improvement observed in some models.

One possible explanation is that the solver settings were not fully compatible with the additional options used. However, keeping what is referred to as the standard options fixed was an intentional choice. This made a controlled comparison of the impact of individual options possible and helped isolate the effects of specific strategies. It also provided a general insight in the behavior of the solver.

The supposed incompatibility of the settings, particularly in DCO-PDN, caused numerical errors, long runtime, or even solver failure, as provided in the supplementary table. These issues were likely caused by the solver entering infeasible regions, probably triggered by too aggressive warm-starts combined with strict precision settings. Interior-point solvers like IPOPT are sensitive to such conditions, especially when initialization points are near constraint boundaries [28]. Unlike simplex methods, IPMs such as IPOPT require iterations strictly within the interior of the feasible region. Points that lie on or too close to the boundary, potentially coming from warm-starts or poorly scaled models, can violate this assumption and lead to numerical instability or failure [87]. Well-centered starting points, even if further from the solution, are often more reliable than those near the boundary [88].

In addition to testing individual solver options, several logically motivated combinations were tried, not including the warm-start and initial options, to see if other options might perform together. As shown in the supplementary tables, these test series did not lead to success. This confirms the interpretation from before and suggests that combinations can speed up convergence only if they are well-chosen. Those options combined, did also not show any positive effect individually. Aggressive parameters can worsen solver performance. So other options might not improve convergence because some changed the numerical landscape of the problem too much and required adjustments to the fixed standard settings as well.

The behavior can also be explained by how IPOPT operates. It is based on a filter line-search technique to ensure global convergence without relying on penalty functions [33]. This means that, under mild regularity conditions (such as smoothness and valid constraint qualifications), the method guarantees that the algorithm will converge to a feasible solution, and that at least one limit point will be a stationary point of the NLP [28]. However, these convergence assurances only apply if the problem is well-conditioned and numerically consistent. Moreover, this type of global convergence refers to convergence from any starting point to a local solution, not to a globally optimal solution. NLPs often have multiple local optima and IPOPT converges to different solutions depending on the initial point or numerical tolerances. In practice, it might be necessary to solve the model multiple times with different starting points to identify better local optima.

Several studies [89], [90], [91], [88] also highlight the importance of warm-start strategies and increasing the efficiency of interior-point methods. Generalizing the initialization

process can improve solver performance, often reducing computational cost and runtime. In dcFBA, a potential warm-start strategy is to compute a single steady-state FBA at the start and use its fluxes to initialize all time points. This approach provides a biologically meaningful starting point, to improve convergence across the dynamic simulation. Using the optimal solution from the original problem to warm-start is usually an ineffective way. It often lies on the boundary of the feasible region, where standard IPMs cannot operate effectively [87]. This behavior was observed when trying to warm-start the models in the new solver MadNLP. Since MadNLP does not support integrated warm-start, an attempt was made to use the solution from IPOPT. However, this approach was unsuccessful and was not continued.

It is also important to note that there already exist configurations that successfully solved both DCO-BUT and DCO-PDN. Adding further solver options had little to no effect on the solution quality. Most were solved locally or almost locally, with final results in the same order of magnitude. This also supports the decision to keep the original settings, as a reliable baseline. These findings highlight an important point about solver behavior in nonlinear programming. While no variations in the final solution were observed across the different configurations, the path and time the solver took to reach that solution were very different. This suggests that IPOPT consistently converged to the same local optimum, but the efficiency of convergence was sensitive to the solver settings. Some strategies led to slower convergence, possibly due to inefficient step directions, increased iteration counts, or time spent restoring feasibility. In future work, the solver’s output can be analyzed more closely to identify which calculations take the most time and find options to improve their performance.

In nonlinear optimization problems like dcFBA, the objective function and constraints often include terms with different scales. For instance, multipliers or penalty terms can differ by orders of magnitude. Similarly, derivatives like gradients, Jacobians, and Hessians can also vary in scale. If these scales are not properly balanced, solvers can struggle to take effective optimization steps and converge efficiently. This issue is particularly pronounced in IPMs, which require smooth gradient and second-order information, and appears to be a common concern in discussions on the Julia forum related to NLP and IPMs. Furthermore, even when the solution remains the same, solver performance can differ depending on numerical tolerances, globalization strategies, and scaling. In fact, poor scaling appeared to be a significant challenge during testing of MadNLP, as discussed later.

The intention was not to change a functioning setup, but to investigate whether additional tuning could lead to runtime improvements. The goal was to determine whether certain combinations of solver options could consistently enhance performance across different problem instances, ideally uncovering a systematic trend.

When comparing the performance of the two reference models, no consistent pattern appeared regarding which solver options improved the solution process reliably. Both models had contrasting responses to the same configurations. Although the models are structurally very similar, the solver handled them differently. However, the initialization

and warm-start options that improved performance in DCO-BUT had a similar positive effect in the larger model `lac_005`. This suggests that certain solver settings can be successfully reused in other problems, which is a valuable insight. Analyzing a smaller problem with similar characteristics can be quite helpful for identifying promising solver settings. This approach can limit the selection of options to test on the larger model, which often requires more time to solve and makes extensive testing more exhausting.

The main outcome of this part is that there is no specific solver configuration that consistently improves performance. Instead, the best results were achieved by performing multiple runs with different initialization and warm-start settings, specifically adjusting `bound_push`, `bound_frac` and `warm_start_bound_push`, then selecting the run that delivered the best outcome.

3.2 Solver performance

To improve the solving process, a different approach was considered. As described in Methods, we focused on an alternative solver `MadNLP` and investigated the integration of GPU computation within this context and different model set-ups.

3.2.1 JuMP and MadNLP (CPU)

Although both `IPOPT` and `MadNLP` solvers are based on an interior-point method, simply swapping the solvers did not result in a smooth transition without issues. When the solver was first applied to the DCO-PDN implementation, it terminated early due to the occurrence of NaN ('not-a-number') values, which led to $\pm\infty$ during the solving process, eventually causing the solver to fail. In this setup, the specific locations of the NaN values were not identified. However, in the experimental setup `JuMP + ExaModels + MadNLP`, discussed in Section 3.2.3, they were identified occurring in a vector d , suggesting that this might have been the same underlying issue. Further details are provided in that particular section. After adapting the original implementation to the updated `JuMP` syntax and performing some general package updates, this issue was resolved, allowing the program to run through the process and solve the problem locally.

The first set of results revealed that `IPOPT` and `MadNLP` operate and solve the problem differently. To evaluate their efficiency, the focus was again on runtime and additionally the number of iteration steps. It was observed that `MadNLP` required fewer iteration steps (587 compared to 730), but it took more time to complete these fewer steps, as shown in Figure 3.3. Although the solver found a solution, it converged to a different one than `IPOPT`. The final product concentration is lower (0.588909491 vs. 0.920876097) and also the biomass amount shows a notable difference (39.44418855 vs. 81.86880806). Given that `MadNLP` also offers a range of options, future work could explore different configurations or develop warm-start strategies to bring its solution closer to that obtained with `IPOPT`.

However, when the DCO-BUT implementation was tested under the same conditions, the issue with NaN values was not encountered. In this case, the solver successfully completed the optimization process and produced results that closely matched those of the original with no notable differences. Analyzing the runtime and iteration steps, the behavior was consistent with the previous results. The solver required only 474 iteration steps to converge, compared to 1645 steps in the original implementation. Although the overall runtime did not show a reduction, IPOPT completed the process in 62 seconds, whereas MadNLP required 116 seconds to perform only a quarter of the steps as shown in Figure 3.3.

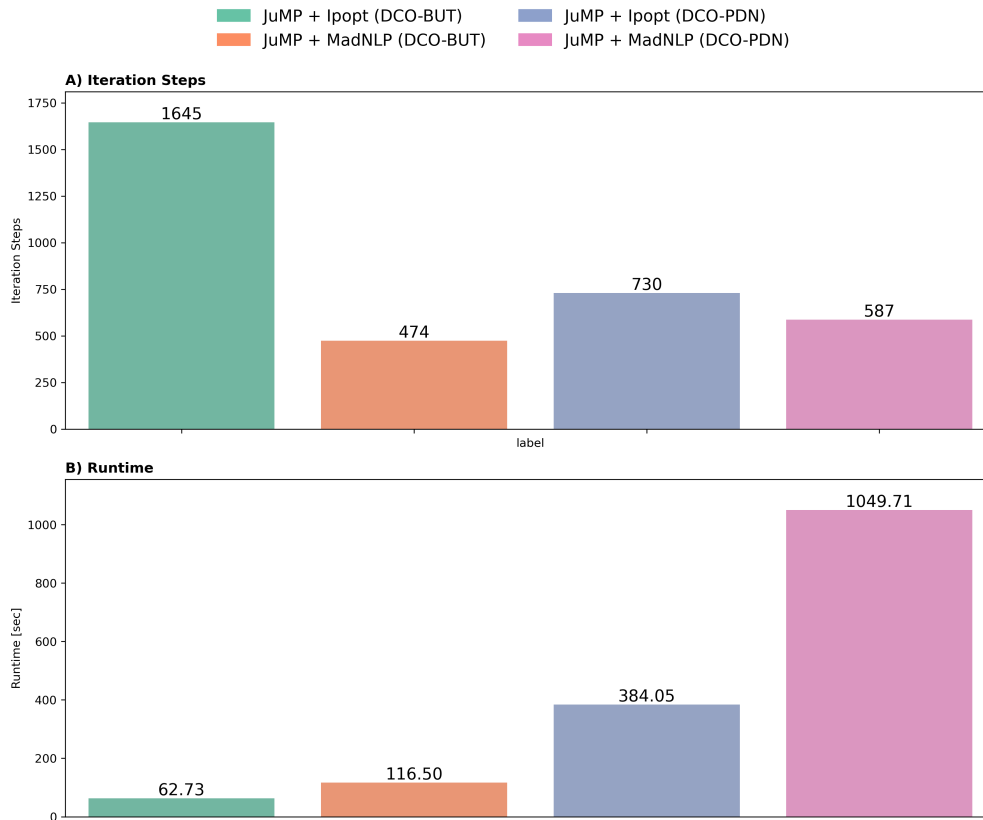


Figure 3.3: Iteration steps and runtime of DCO-BUT and DCO-PDN

3.2.2 ExaModels and MadNLP (CPU)

To address the encountered issues, the initial JuMP model was converted to ExaModels. While JuMP and ExaModels share similar syntax, the limited documentation of ExaModels made the transition non-trivial, requiring a trial-and-error approach to achieve a functional setup. The process began by testing the smaller problem DCO-BUT as it had produced

There was a delay before **MadNLP** started, followed by the generation of long symbolic output (see Figure 3.4) in the terminal suggesting that this is how the model is preprocessed within **ExaModels**. Although the solver did start after pre-processing, it immediately failed with the termination message **OTHER ERROR**. Unfortunately, errors in **ExaModels** are currently not well documented or fully implemented, which means that necessary information is missing to troubleshoot and improve the process further.

Figure 3.4: Output of DCO-BUT

52

ExaModels occurred. Defining constraints inside for loops led to complications. In such cases, **ExaModels** failed to properly recognize the constraints as part of the model, which prevented a successful setup of the model. Also, debugging this issue was difficult at that point. In conclusion, it was observed that constraint definition involving loops like those used in **JuMP** did not translate directly to the **ExaModels** framework. This limitation is also probably related to the GPU-friendly architecture, as loop-based constructions are generally inefficient on GPUs.

Since these issues could not be resolved within a reasonable time frame, the focus shifted to an experimental approach elaborated in the following section.

3.2.3 **JuMP and ExaModels and MadNLP (CPU)**

Because integrating **ExaModels** is essential to explore GPU compatibility, the only remaining option is the experimental approach that combines **JuMP** and **ExaModels**. For initial tests, the smallest problem **DCO-BUT** was used. After making the modifications specified in the documentation and described in Methods, the solver ran successfully but failed to converge and terminated after reaching the maximum number of iteration steps. As the setup seemed correct and the solver ran without errors or warnings, the focus was on understanding and resolving the lack of convergence. Therefore, the solver output was carefully analyzed to identify potential causes of convergence failures. Since there is limited documentation on how to interpret **MadNLP** outputs, I referred to the explanation given of **IPOPT**, assuming there might be similarities.

To better understand the solver’s performance, various **MadNLP** option configurations were tested, as shown in the supplementary tables. Therefore, key metrics were selected to evaluate convergence behavior and the effectiveness of various solver options, with the focus on the objective value, as it provides a meaningful insight into the solving process. This analysis aimed to gain a deeper understanding of **ExaModels** pre-processing and **MadNLP** performance.

Variables and Constraints Analysis

Before analyzing the solver output in detail, the initial focus was on the model pre-processing and how much they differ. Since **JuMP + ExaModels + MadNLP** was the only approach that worked with **ExaModels**, the comparison was limited to this approach. The comparison between **ExaModels + MadNLP** and **JuMP + IPOPT** revealed a notable difference in the number of variables with only lower bounds. The **JuMP + IPOPT** implementation had 420 such variables, while **MadNLP** configurations resulted in 510. This discrepancy may indicate differences in the way **ExaModels** pre-processes the model with **MadNLP**. Despite this, the total number of variables, constraints, and their classifications were consistent across all **MadNLP** configurations except for the total number of lower bounds. Unclear whether this difference was important or why it occurred, it was set aside temporarily. If later analysis reached a dead end or produced unexpected results, the issue would be revisited.

Convergence Analysis

After extensive experimentation, two configurations of options achieved convergence and produced results consistent with those from IPOPT but in fewer iterations:

Parameter	option set 8	option set 9
tol	1e-7	1e-6
max_iter	5000	5000
required_infeasibility_reduction	0.7	0.5
acceptable_tol	1e-7	1e-5
acceptable_iter	10	10
barrier_tol_factor	20.0	20.0
soft_resto_perror_reduction_factor	0.7	--
output_file	"INFO"	"INFO"

Table 3.6: MadNLP Optimizer settings of option set 8 and option set 9

MadNLP successfully solved the problem using option set 8 in 728 iterations and option set 9 in 441 iterations. Both configurations required fewer iterations compared to the original implementation with IPOPT, which required 1645 iterations to reach termination.

Name	Final Product Amount [mmol]	Final Biomass Amount [g]
DCO-BUT	760.739182940161	9.85616981276809
Option Set 8	760.739540726352	9.85616482215877
Option Set 9	760.7430191673736	9.856125039234705

Table 3.7: Comparison of final product and biomass for DCO-BUT, option set 8, and option set 9.

While the solver found an optimal solution, the exit status remained `OTHER ERROR`, and there is no section in the MadNLP documentation that explains this status or the associated errors, unlike IPOPT. This might indicate a bug in the implementation, as not all error messages seem to be defined yet. But one optimal solution was found.

Another observation is the unusually small and negative objective value across all simulations. Since the objective value is critical to the optimization performance, plotting its evolution over time (see Figure 3.5 provides insights into the impact of different solver options. This analysis helps identify which parameters should be adjusted, either increased or decreased, to improve convergence behavior.

Unlike IPOPT (light blue curve), which produced a positive objective value (indicating a good result for a maximization problem), MadNLP's objective value raised questions. However, after comparing it to the example in the blog post [81], it appears as typical behavior of MadNLP. Despite the unusual objective value, the optimal solution found

matches the result from the IPOPT implementation. As seen in Figure 3.5, the objective value in the original implementation follows a similar trend to the other simulations. As `ExaModels` defaults to `minimize = true` in its `ExaCore` setup, the objective function is treated as a minimization problem, explaining why the objective value is behaving in the opposite direction.

Figure 3.5 illustrates the evolution of the normalized objective value as a function of relative progress (%) for different implementations varying in their solver options. The graphs were scaled to account for different iteration limits, allowing for a fair comparison. The primary objective of this analysis is to assess whether the optimization process exhibits convergence or not.

Among the tested implementations, option set 8 (olive green curve) and 9 (grey curve) demonstrate a convergence behavior. In both cases, the same solution was reached in 728 and 441 iteration steps, fewer than those required by IPOPT. Their curves follow a similar trajectory, showing a steady decrease in the objective value over time before reaching a stable minimum. This indicates that these implementations successfully found a feasible and (local) optimal solution to the problem. In contrast, the other option sets do not exhibit the same convergence properties. Instead, they either terminate due to the predefined iteration limit and show a strictly decreasing trend without achieving a solution or reaching a stable minimum. Although strictly decreasing objective values are a promising sign for a minimization problem, they do not reach a stable minimum before the solver terminates, unlike in cases where an optimal solution was found.

Contrary to what is shown in Figure 3.5, the non-converging implementations do not actually achieve the same objective value (see Supplements). This appearance in the graph is only due to the normalization applied. The purpose was to illustrate the solver’s behavior during the optimization process and to compare how convergence and non-convergence are reflected in the objective value.

The second test involved DCO-PDN. Initially, the solver failed due to the occurrence of NaN values, which were reproduced through the calculations and resulted in $\pm\infty$ values. Upon further investigation, the issue was traced to the Lagrange constraint B and the parameter d , a vector, where NaN values occur during the initialization process. To reduce this issue, a temporary workaround was implemented by replacing all NaN values in d with zeros during the initialization phase, using a function available in JuMP. This adjustment allowed the solver to proceed with the optimization. However, despite running without immediate errors, the problem still failed to converge, indicating deeper unresolved issues within the setup.

3.2.4 JuMP and ExaModels and MadNLP (GPU)

Since the experimental approach was the only working setup and effective for DCO-BUT, it was the only option to test JuMP + ExaModels + MadNLP on GPU. After implementing the necessary modifications, the DCO-BUT model successfully executed on the GPU on the

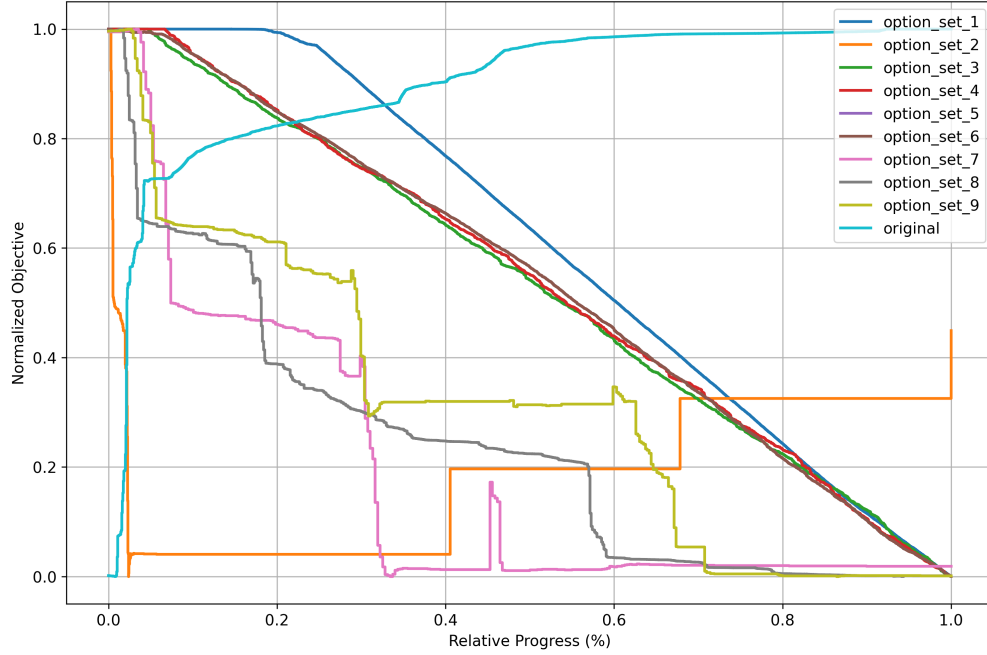


Figure 3.5: Objective Value over time: different option sets of DC0-BUT on CPU

first attempt, utilizing approximately 41% of the available GPU resources, a considerable amount given the problem size. However, it failed to converge and terminated upon reaching the maximum number of allowed iterations. After employing the same solver configurations that worked on CPU, on GPU, the results differed. In response, the solver options were refined again, with a focus on the objective value over time (see Figure 3.6). The objective value curves all show similar behavior, failing to reach a stable minimum.

During this process, an unexpected issue was observed. The same solver settings do not consistently produce the same results (see Figure 3.7, even if the terminal and Julia environment were restarted). While output variations can occur during simulations, the differences observed were noticeable, especially on the GPU.

Since solver option tuning did not lead to improvements, one potential explanation for the observed issues is the difference in memory handling between CPU and GPU execution. Specifically, certain objects, such as arrays and matrices, may be allocated in different memory locations when transferred to the GPU. To address this, some objects were explicitly moved to GPU using `CuArray()`. However, this introduced additional challenges related to scalar indexing, which is frequently used in the current implementation and not efficiently supported on GPU. This leads to performance warnings and reduced computational efficiency. As a consequence, these indexing operations became a bottleneck in the solving process, highlighting the need for refactoring to achieve better GPU compatibility.

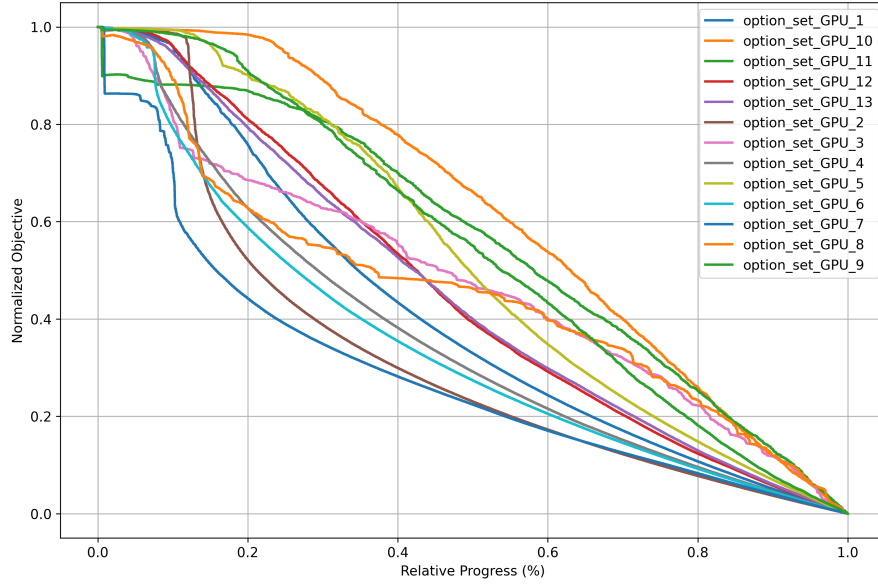


Figure 3.6: Objective Value over time: different option sets of DCO-BUT on GPU

3.2.5 Discussion

So far, we have explored various frameworks and solver configurations for the given optimization problems and compared their performance. Since not all approaches were successful, Table 3.8 provides an overview of the tested implementations, indicating which were successful and which were not.

Set-Up	ext_but_043	P_v2024
JuMP + IPOPT on CPU (initial situation)	✓	✓
JuMP + MadNLP on CPU	✓	✓*
ExaModels + MadNLP on CPU	✗	✗
ExaModels + MadNLP on GPU	no test	no test
JuMP + ExaModels + MadNLP on CPU	✓	✓*
JuMP + ExaModels + MadNLP on GPU	✓*	✗

Table 3.8: Implementation Setup

Summary of different setups. A ✓ indicates that the setup works, while a ✗ indicates failure. The * denotes setups that worked but had worse results compared to the reference or did not converge.

Since ExaModels is still under development and not as well-tested or documented as tools like IPOPT and JuMP, setting up complex optimization problems was challenging. The

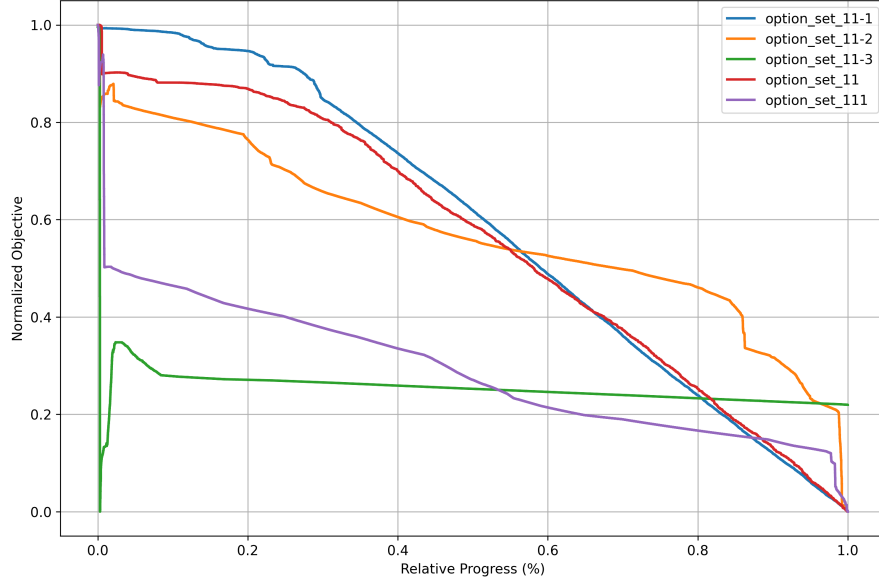


Figure 3.7: Objective Value over time: Same solver setting; different outcome

correct implementation of the model is essential for an accurate solver performance, but some features are not fully explained in the current documentation. For instance, it is unclear how **ExaModels** handles inequalities, as no operators for this functionality appear to be supported at this stage. Additionally, not all error messages are documented, which prevents a deeper understanding of errors beyond terminal feedback.

Several key observations can be drawn from the testing conducted on **ExaModels** and **JuMP**, particularly regarding their performance on GPU-based computations.

One important observation is that both frameworks, **JuMP** and **ExaModels**, rely on symbolic computations. While this is not necessarily an error, delays during model construction may be caused by recursion over large tuples or argument lists (see Figure 3.4). This seems plausible, given that both frameworks use symbolic placeholders to represent variables and constraints, and the model itself is mathematically large and complex.

This symbolic nature seems to create challenges when trying to solve the model efficiently on GPU. Although GPU is well-known for accelerating large-scale computations, they are best suited for numerical tasks that are highly parallel and have regular memory access patterns [92]. Applications of GPUs outside graphics focused on solving large systems of differential equations and linear algebra problems, where the computation could be split across many threads running parallel [92]. For example, in [93], they demonstrated GPU-based methods for linear algebra by efficiently representing matrices and vectors using graphics hardware.

In contrast, symbolic models typically involve more complex data structures, resulting in irregular memory access and control logic that varies across threads. These characteristics are known to reduce performance on GPUs [92], [94]. While smaller and simpler models ran successfully on GPU, larger models with more symbolic parts ran noticeably slower. In such cases, numerical calculations still performed well, indicating that symbolic computation and low-level issues like scalar indexing can create bottlenecks. This suggests that while GPU is indeed a powerful tool for “big data,” their strengths lie primarily in numeric, data-parallel workloads. Symbolic models with many constraints and decision points do not fit well with this structure yet. To run the models efficiently on GPU, they may need to be rewritten with a focus on parallel programming.

During testing, an issue related to the sensitivity of models to updates was noticed. After updating all packages to their latest versions, some scripts that previously worked as expected began to produce errors or inconsistent results. For example, the DCO-PDN script, which had previously executed the solver without convergence (but with normal log output), failed to start the solver after the update, instead producing symbolic output typical of **ExaModels**. Additionally, solver settings that had worked for convergence in DCO-BUT no longer worked after the update, and new settings were required. Which versions changed can be looked up in the Table 3.9. This suggests that solver settings may need to be adjusted over time, as some configurations may become outdated. This also shows that **ExaModels** is still in development and might not be fully reliable for large-scale problems at the moment.

Package	Before Update	After Update
CSV	v0.10.14	v0.10.15
CUDA	v5.4.3	v5.6.1 (dev)
DataFrames	v1.6.1	v1.7.0
Dierckx	v0.5.3	v0.5.4
ExaModels	v0.7.1	v0.8.0
JLD2	v0.4.53	v0.5.11
JuMP	v1.23.1	v1.23.6
MadNLP	v0.8.4	v0.8.5
MadNLPGPU	v0.7.3	v0.7.6

Table 3.9: Packages with version changes in the latest update

Another issue identified was the inconsistency in error messages. In some cases, error messages were not informative or documented, making it difficult to determine the exact cause of failures. This inconsistency may indicate potential bugs in either the solver or the model’s implementation. Notably, despite similarities in formulation, objects, and parameters, the two tested models produced different error messages, an unexpected result that complicates troubleshooting and raises questions about the robustness and stability of the software.

Besides all these issues and errors could also suggest that the current model may not be optimally formulated. The suboptimal performance on the GPU underlines the need to rethink the model structure and consider alternative mathematical formulations or solution strategies. In online discussions, it was suggested that the model might suffer from poor numerical scaling. This idea is supported by research, which shows that scaling or normalization of variables and constraints can significantly influence the performance of interior-point methods. For example, proper scaling improves convergence in linear and conic optimization problems [95], and other studies highlight how techniques such as diagonal scaling or preconditioning can enhance solver efficiency in large-scale or nonlinear problems [96], [97].

With this in mind, we explored an alternative mathematical approach, including an experimental attempt to implement the Forward Euler method, even though it is generally not mathematically well-suited for the problem, as it involves a stiff system.

The Forward Euler method is generally unsuitable for solving stiff ordinary differential equations (ODEs) [98]. Stiff systems of ODEs involve processes that evolve on very different time scales, where the fast dynamics are typically stable [99]. This results in computationally expensive simulations that are often impractical [98]. However, since the implementation of the Forward Euler method is simple, it was tested out of curiosity. Surprisingly, the method produced results with two different option configurations (see Table 3.10). The results did not exactly match those of the reference but are in a similar dimension (see Table 3.11). Notably, the feed and growth rates were identical.

Parameter	Option Set 1	Option Set 4
tol	1e-5	1e-7
max_iter	5000	5000
required_infeasibility_reduction	0.3	0.7
acceptable_tol	1e-5	1e-6
acceptable_iter	10	10
barrier_tol_factor	20.0	20.0
output_file	"INFO"	"INFO"

Table 3.10: Two successful solver configurations for the Forward-Euler implementation

Name	Final Product Amount [mmol]	Final Biomass Amount [g]
original	760.739182940161	9.85616981276809
Option Set 1	752.5971369807318	10.66048433907264
Option Set 4	752.5637075916009	10.660446169915785

Table 3.11: Comparison of final product and biomass

These results were again evaluated by tracking the objective value over time (see Figure 3.8), showing slightly different behavior in the curves compared to previous observations.

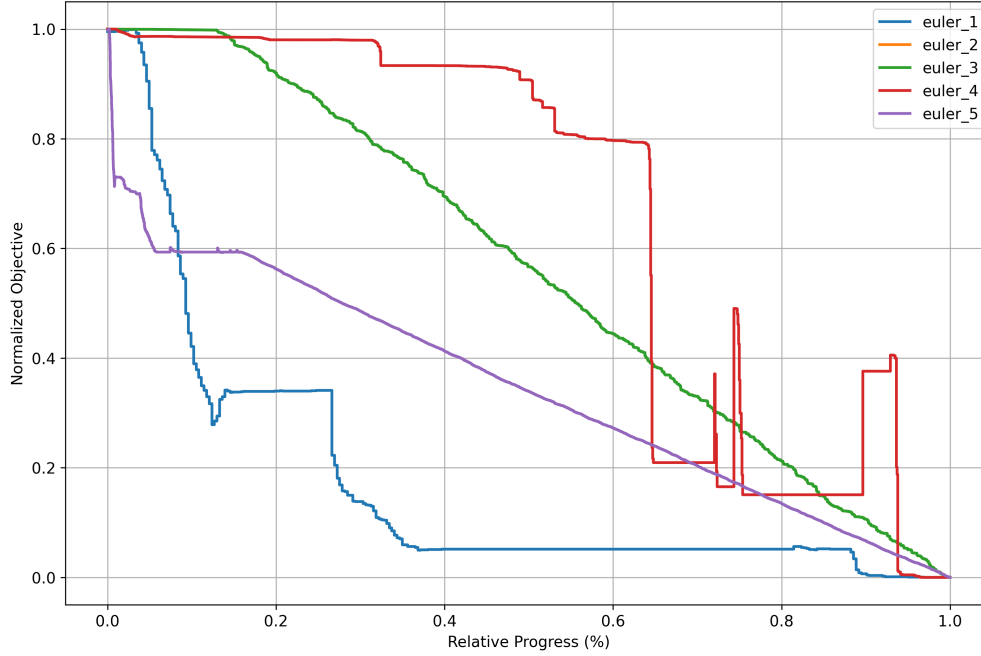


Figure 3.8: Objective Value over time: different option sets of the Forward-Euler implementation

Stiff ODE systems occur frequently in scientific and engineering fields like chemistry, biology, and physics, where they describe complex dynamic processes [99], [98]. Numerous studies have investigated the limitations of the Forward Euler method for stiff systems, particularly its stability issues [100]. As a result, researchers have proposed various strategies, including automatic step-size control [101], to improve its performance and extend its application to such problems.

However, since the robustness of this approach is uncertain and multiple options lead to solutions of the same dimension, it is presented as an observation rather than a definitive conclusion.

Chapter 4

Conclusion

The overall goal of this thesis was to investigate potential improvements to the solving process of dynamic control flux balance analysis models.

It started with previously solved models and explored whether solver performance could be enhanced through tuning the large pool of available IPOPT options. The intention was to identify a systematic approach for setting these parameters, particularly since the models shared many structural similarities. We were able to identify two categories where individual options led to a reduction in runtime. The expected consistent improvements were not achieved, but some configurations did lead to noticeable runtime reductions. Especially for one of the larger models, which marks an important step forward.

This outcome emphasized the need for alternative solving strategies. While IPOPT is widely adopted across different platforms, it has limitations. This motivated the exploration of MadNLP, a solver based on IPOPT but with built-in GPU support. Also ExaModels, a framework designed to facilitate large-scale modeling on GPU. MadNLP did not consistently outperform IPOPT in terms of runtime, but it did reduce the number of iteration steps in most cases while delivering comparable solution quality in some cases. However, MadNLP's convergence proved to be more sensitive to initialization and restoration settings, requiring careful tuning to achieve stable results.

One of the major challenges encountered was the setup and use of ExaModels. The framework is still under development, and its sparse documentation, limited support for constraints like inequalities, and inconsistent error messages made debugging difficult. Additionally, there is also restricted documentation on the underlying optimization algorithm used by MadNLP, which can make debugging more challenging as well.

While GPU is a powerful tool, it did not accelerate the solving process of those models in their current implementation. In fact, GPU-based execution revealed underlying issues in the models. Particularly related to numerical scaling and the use of scalar indexing, which is not well-suited for GPU architectures. Also, frameworks like ExaModels and JuMP rely heavily on symbolic computation, which can introduce additional challenges

during model processing. These symbolic operations may lead to errors or convergence issues, especially when models are not carefully defined or scaled.

In addition, updating packages introduced inconsistencies in solver behavior and underline the importance of version control when working with tools still under development. **MadNLP** has demonstrated potential for better performance on large-scale problems but demands careful attention to solver configurations and model formulation. Similarly, **ExaModels** enables GPU compatibility, but the current implementation is not yet mature enough to fully replace more established frameworks in our cases.

Overall, while **MadNLP** and **ExaModels** show promising directions for future work, **JuMP** with **IPOPT** remains the more robust and reliable choice for solving complex dcFBA models at this stage. This thesis can serve as a foundation for future comparisons, especially once **MadNLP** and **ExaModels** reach the level of stability currently available for **JuMP** and **IPOPT**.

Chapter 5

Bibliography

- [1] Ideen Sadrehaghighi. Optimization problem, 06 2022. URL https://www.researchgate.net/publication/361306781_Optimization_Problem.
- [2] Xin-She Yang and Mehmet Karamanoglu. Chapter 1 - nature-inspired computation and swarm intelligence: a state-of-the-art overview. In Xin-She Yang, editor, *Nature-Inspired Computation and Swarm Intelligence*, pages 3–18. Academic Press, 2020. ISBN 978-0-12-819714-1. doi: <https://doi.org/10.1016/B978-0-12-819714-1.00010-5>. URL <https://www.sciencedirect.com/science/article/pii/B9780128197141000105>.
- [3] Suman Dutta. *Optimization in Chemical Engineering*. Cambridge University Press, 2016. ISBN 9781316691793. URL <https://books.google.at/books?id=Jwa-DAAAQBAJ>.
- [4] Mohammad Ali Bagherian, Kamyar Mehrazamir, Amin Beiranvand Pour, Shahabaldin Rezaei, Elham Taghavi, Hadi Nabipour-Afrouzi, Mohammad Dalvi-Esfahani, and Seyed Morteza Alizadeh. Classification and analysis of optimization techniques for integrated energy systems utilizing renewable energy sources: A review for chp and cchp systems. *Processes*, 9(2), 2021. ISSN 2227-9717. doi: 10.3390/pr9020339. URL <https://www.mdpi.com/2227-9717/9/2/339>.
- [5] Frederick Hillier and Gerald Lieberman. *Introduction To Operations Research*, volume 139, chapter 3: Introduction to Linear Programming. 01 1969. doi: 10.2307/2345190.
- [6] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.
- [7] David G. Luenberger and Yinyu Ye. *Linear and Nonlinear Programming*, volume 116 of *International Series in Operations Research & Management Science*. Springer, New York, NY, 3 edition, 2008. ISBN 978-0-387-74502-2. doi: 10.1007/978-0-387-74503-9. URL <https://doi.org/10.1007/978-0-387-74503-9>.

- [8] J. Nocedal and S. Wright. *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering. Springer New York, 2006. ISBN 9780387400655. URL <https://books.google.at/books?id=VbHYoSye1FcC>.
- [9] Jeff Orth, Ines Thiele, and Bernhard Palsson. What is flux balance analysis? *Nature biotechnology*, 28:245–8, 03 2010. doi: 10.1038/nbt.1614.
- [10] M.S. Bazaraa, H.D. Sherali, and C.M. Shetty. *Nonlinear Programming: Theory and Algorithms*. Wiley, 2013. ISBN 9781118626306. URL <https://books.google.at/books?id=nDYz-NIpIuEC>.
- [11] Julio Banga. Optimization in computational systems biology. *BMC systems biology*, 2:47, 02 2008. doi: 10.1186/1752-0509-2-47.
- [12] J. Nocedal and S. Wright. *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering. Springer New York, 2006. ISBN 9780387400655. URL <https://books.google.at/books?id=VbHYoSye1FcC>.
- [13] Vsevolod I. Ivanov. Duality in nonlinear programming. *Optimization Letters*, 7 (8):1643–1658, 2013. ISSN 1862-4480. doi: 10.1007/s11590-012-0512-6. URL <https://doi.org/10.1007/s11590-012-0512-6>.
- [14] Jon W. Tolle. Nonlinear programming. In Robert A. Meyers, editor, *Encyclopedia of Physical Science and Technology (Third Edition)*, pages 583–596. Academic Press, New York, third edition edition, 2003. ISBN 978-0-12-227410-7. doi: <https://doi.org/10.1016/B0-12-227410-5/00486-5>. URL <https://www.sciencedirect.com/science/article/pii/B0122274105004865>.
- [15] Andreas Wächter. Short tutorial: Getting started with ipopt in 90 minutes. In U. Naumann, O. Schenk, and H. D. Simon, editors, *Combinatorial Scientific Computing*. Chapman and Hall/CRC, 2009. Tutorial presented at CSC09.
- [16] Sri Krishna Vadlamani, Tianyao Patrick Xiao, and Eli Yablonovitch. Physics successfully implements lagrange multiplier optimization. *Proceedings of the National Academy of Sciences*, 117(43):26639–26650, 2020. doi: 10.1073/pnas.2015192117. URL <https://www.pnas.org/doi/abs/10.1073/pnas.2015192117>.
- [17] Christos Tsanas, Erling Stenby, and Wei Yan. Calculation of simultaneous chemical and phase equilibrium by the method of lagrange multipliers. *Chemical Engineering Science*, 174, 09 2017. doi: 10.1016/j.ces.2017.08.033.
- [18] Huijuan Li. Lagrange multipliers and their applications. Department of Electrical Engineering and Computer Science, University of Tennessee, September 2008. URL https://sces.phys.utk.edu/~moreo/mm08/method_HLi.pdf.
- [19] D.P. Bertsekas and W. Rheinboldt. *Constrained Optimization and Lagrange Multiplier Methods*. Computer science and applied mathematics. Academic

- Press, 2014. ISBN 9781483260471. URL <https://books.google.at/books?id=j6LiBQAAQBAJ>.
- [20] Jorge Nocedal, Andreas Wächter, and Richard A. Waltz. Adaptive barrier update strategies for nonlinear interior methods. *SIAM Journal on Optimization*, 19(4):1674–1693, 2009. doi: 10.1137/060649513. URL <https://doi.org/10.1137/060649513>.
 - [21] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004. ISBN 9781107394001. URL <https://books.google.at/books?id=IUZdAAAAQBAJ>.
 - [22] *Karush—Kuhn—Tucker Theory*, pages 153–177. Springer New York, New York, NY, 2006. ISBN 978-0-387-31256-9. doi: 10.1007/978-0-387-31256-9_7. URL https://doi.org/10.1007/978-0-387-31256-9_7.
 - [23] J. Nocedal and S. Wright. *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering. Springer New York, 2006. ISBN 9780387400655. URL <https://books.google.at/books?id=VbHYoSye1FcC>.
 - [24] L.T. Biegler. *Nonlinear Programming: Concepts, Algorithms, and Applications to Chemical Processes*. MOS-SIAM Series on Optimization. Society for Industrial and Applied Mathematics, 2010. ISBN 9780898717020. URL <https://books.google.at/books?id=ZmIC7w9QnPEC>.
 - [25] Radhakrishnan Mahadevan, Jeremy S. Edwards, and Francis J. Doyle. Dynamic flux balance analysis of diauxic growth in escherichia coli. *Biophysical Journal*, 83(3):1331–1340, 2002. ISSN 0006-3495. doi: [https://doi.org/10.1016/S0006-3495\(02\)73903-9](https://doi.org/10.1016/S0006-3495(02)73903-9). URL <https://www.sciencedirect.com/science/article/pii/S0006349502739039>.
 - [26] Steffen Waldherr, Diego A. Oyarzún, and Alexander Bockmayr. Dynamic optimization of metabolic networks coupled with gene expression, 2014. URL <https://arxiv.org/abs/1309.4936>.
 - [27] Jéssica A. Delgado, Edméa C. Baptista, Antonio R. Balbo, Edilaine M. Soler, Diego N. Silva, André C.P. Martins, and Leonardo Nepomuceno. A primal–dual penalty-interior-point method for solving the reactive optimal power flow problem with discrete control variables. *International Journal of Electrical Power Energy Systems*, 138:107917, 2022. ISSN 0142-0615. doi: <https://doi.org/10.1016/j.ijepes.2021.107917>. URL <https://www.sciencedirect.com/science/article/pii/S0142061521011297>.
 - [28] Andreas Wächter and Lorenz Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical programming*, 106:25–57, 03 2006. doi: 10.1007/s10107-004-0559-y.

- [29] Usman Aliyu Abdullahi, Ahmad Abubakar Suleiman, and Aminu Suleiman. Comparative analysis of simplex method and the family of interior point method. pages 11–15, 08 2017. doi: 10.9790/5728-1304021115.
- [30] Florian A. Potra and Stephen J. Wright. Interior-point methods. *Journal of Computational and Applied Mathematics*, 124(1):281–302, 2000. ISSN 0377-0427. doi: [https://doi.org/10.1016/S0377-0427\(00\)00433-7](https://doi.org/10.1016/S0377-0427(00)00433-7). URL <https://www.sciencedirect.com/science/article/pii/S0377042700004337>. Numerical Analysis 2000. Vol. IV: Optimization and Nonlinear Equations.
- [31] Nai-Yuan Chiang and Victor M. Zavala. An inertia-free filter line-search algorithm for large-scale nonlinear programming. *Computational Optimization and Applications*, 64(2):327–354, 2016. doi: 10.1007/s10589-015-9820-y. URL <https://doi.org/10.1007/s10589-015-9820-y>.
- [32] Jacek Gondzio. Interior point methods in the year 2025. *EURO Journal on Computational Optimization*, 13:100105, 2025. ISSN 2192-4406. doi: <https://doi.org/10.1016/j.ejco.2025.100105>. URL <https://www.sciencedirect.com/science/article/pii/S2192440625000024>.
- [33] Andreas Wächter and Lorenz Biegler. Line search filter methods for nonlinear programming: Motivation and global convergence. *SIAM Journal on Optimization*, 16:1–31, 01 2005. doi: 10.1137/S1052623403426556.
- [34] Hande Benson, Robert Vanderbei, and David Shanno. Interior-point methods for nonconvex nonlinear programming: Filter methods and merit functions. *Computational Optimization and Applications*, 23, 11 2002. doi: 10.1023/A:1020533003783.
- [35] S.J. Wright. *Primal-Dual Interior-Point Methods*. Other Titles in Applied Mathematics. Society for Industrial and Applied Mathematics, 1997. ISBN 9780898713824. URL <https://books.google.lu/books?id=2hANEPgs7CkC>.
- [36] Ipopt documentation - options, . URL <https://coin-or.github.io/Ipopt/OPTIONS.html>. [Online; Accessed: 05.02.2025].
- [37] Ipopt documentation - ouput, . URL <https://coin-or.github.io/Ipopt/OUTPUT.html>. [Online; Accessed: 05.02.2025].
- [38] Sungho Shin, Carleton Coffrin, Kaarthik Sundar, and Victor M Zavala. Graph-based modeling and decomposition of energy infrastructures. *arXiv preprint arXiv:2010.02404*, 2020.
- [39] Sungho Shin, Mihai Anitescu, and François Pacaud. Accelerating optimal power flow with gpus: Simd abstraction of nonlinear programs and condensed-space interior-point methods. *Electric Power Systems Research*, 236:110651, 2024. ISSN 0378-7796. doi: <https://doi.org/10.1016/j.epsr.2024.110651>. URL <https://www.sciencedirect.com/science/article/pii/S0378779624005376>.

- [40] Zulfiqar A Memon, Fahad Samad, Zafar Rehman Awan, Abdul Aziz, and Shafaq Siraj Siddiqi. Cpu-gpu processing. *International Journal of Computer Science and Network Security*, 17(9):188–193, 2017.
- [41] Miles Lubin, Oscar Dowson, Joaquim Dias Garcia, Joey Huchette, Benoît Legat, and Juan Pablo Vielma. JuMP 1.0: Recent improvements to a modeling language for mathematical optimization. *Mathematical Programming Computation*, 2023. doi: 10.1007/s12532-023-00239-3.
- [42] Dominique Orban, Abel Soares Siqueira, and contributors. Nlpmodels.jl: Data structures for optimization models, 2023. URL <https://github.com/JuliaSmoothOptimizers/NLPModels.jl>.
- [43] Timothy A. Davis. Algorithm 832: Umfpack v4.3—an unsymmetric-pattern multifrontal method. *ACM Trans. Math. Softw.*, 30(2):196–199, June 2004. ISSN 0098-3500. doi: 10.1145/992200.992206. URL <https://doi.org/10.1145/992200.992206>.
- [44] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users’ Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999. ISBN 0-89871-447-8 (paperback).
- [45] HSL. A collection of fortran codes for large scale scientific computation. <http://www.hsl.rl.ac.uk/>.
- [46] Jump documentation. [urlhttps://jump.dev/JuMP.jl/stable/](https://jump.dev/JuMP.jl/stable/), 2025. Accessed: 2025-04-06.
- [47] Should i use jump? [urlhttps://jump.dev/JuMP.jl/stable/should_i_use/](https://jump.dev/JuMP.jl/stable/should_i_use/), 2025. Accessed: 2025-08-06.
- [48] Benoît Legat, Oscar Dowson, Joaquim Dias Garcia, and Miles Lubin. MathOpt-Interface: a data structure for mathematical optimization problems. *INFORMS Journal on Computing*, 34(2):672–689, 2021. doi: 10.1287/ijoc.2021.1067.
- [49] Algebraic modeling languages (background). [urlhttps://jump.dev/JuMP.jl/stable/background/algebraic_modeling_languages/](https://jump.dev/JuMP.jl/stable/background/algebraic_modeling_languages/), 2025. Accessed: 2025-04-06.
- [50] Robert Fourer, David Gay, and Brian Kernighan. *AMPL: A Modeling Language for Mathematical Programming*, volume 36. 01 2002. doi: 10.1287/mnsc.36.5.519.
- [51] Klaus Wüst. *Single Instruction Multiple Data (SIMD)*, pages 249–257. Vieweg+Teubner, Wiesbaden, 2011. ISBN 978-3-8348-9881-4. doi: 10.1007/978-3-8348-9881-4_13. URL https://doi.org/10.1007/978-3-8348-9881-4_13.

- [52] T. Brandes, A. Arnold, T. Soddemann, and D. Reith. Cpu vs. gpu - performance comparison for the gram-schmidt algorithm. *The European Physical Journal Special Topics*, 210(1):73–88, August 2012. ISSN 1951-6401. doi: 10.1140/epjst/e2012-01638-7. URL <https://doi.org/10.1140/epjst/e2012-01638-7>.
- [53] ExaModels.jl contributors. Examodels.jl documentation. <https://exanauts.github.io/ExaModels.jl/dev/>, 2025. Accessed: 2025-05-13.
- [54] J. Revels, M. Lubin, and T. Papamarkou. Forward-mode automatic differentiation in Julia. *arXiv:1607.07892 [cs.MS]*, 2016. URL <https://arxiv.org/abs/1607.07892>.
- [55] Jarrett Revels. ReverseDiff.jl. <https://github.com/JuliaDiff/ReverseDiff.jl>, 2020. Julia package.
- [56] Tangi Migot, Dominique Orban, Abel Soares Siqueira, and contributors. Nlpmodelsmodifiers.jl: Model modifiers for nlpmodels, 2023. URL <https://github.com/JuliaSmoothOptimizers/NLPModelsModifiers.jl>. Released on 2023-07-07.
- [57] H.C. Lim and H.S. Shin. *Fed-Batch Cultures: Principles and Applications of Semi-Batch Bioreactors*. Cambridge Series in Chemical Engineering. Cambridge University Press, 2013. ISBN 9780521513364. URL <https://books.google.at/books?id=qVpob0mh0HsC>.
- [58] HSE Chuo, MK Tan, HJ Tham, KTK Teo, and Kota Kinabalu. Optimization of fed-batch baker’s yeast fermentation process using learning algorithm. In *Proceedings of 18th Regional Symposium on Chemical Engineering*, pages 27–28, 2011.
- [59] G. Jagschies, E. Lindskog, K. Lacki, and P.M. Galliher. *Biopharmaceutical Processing: Development, Design, and Implementation of Manufacturing Processes*. Elsevier, 2018. ISBN 9780128125526. URL <https://books.google.at/books?id=b9F1DQAAQBAJ>.
- [60] T. Sauter and M. Albrecht. *Introduction to Systems Biology: Workbook for Flipped-classroom Teaching*. Open Book Publishers, 2023. ISBN 9781800644120. URL <https://books.google.at/books?id=kNayEAAAQBAJ>.
- [61] Joshua Kaste and Yair Shachar-Hill. Accurate flux predictions using tissue-specific gene expression in plant metabolic modeling. *Bioinformatics (Oxford, England)*, 39, 04 2023. doi: 10.1093/bioinformatics/btad186.
- [62] Christopher D. Garay, Jonathan M. Dreyfuss, and James E. Galagan. Metabolic modeling predicts metabolite changes in mycobacterium tuberculosis. *BMC Systems Biology*, 9:57, 2015. ISSN 1752-0509. doi: 10.1186/s12918-015-0206-7. URL <https://bmcsystbiol.biomedcentral.com/articles/10.1186/s12918-015-0206-7>. PMCID: PMC4574064.

- [63] J. A. Jones and M. A. G. Koffas. Optimizing metabolic pathways for the improved production of natural products. *Methods in Enzymology*, 575:179–193, 2016. ISSN 1557-7988. doi: 10.1016/bs.mie.2016.02.010. URL <https://doi.org/10.1016/bs.mie.2016.02.010>. Epub 2016 Mar 10.
- [64] D. Jia, Y. Su, M. Lu, and X. Li. *Systems Biology and Single-cell Analysis of Cancer Metabolism and its Role in Cancer Emergent Properties*. Frontiers Media SA, 2023. ISBN 9782832526965. URL https://books.google.at/books?id=yF_GEAAAQBAJ.
- [65] TimVickers. Metabolic network of *A. thaliana* tca cycle, 2007. URL https://commons.wikimedia.org/wiki/File:A_thaliana_metabolic_network.png. Created with Cytoscape using data from VirtualPlant 0.9. Public domain via Wikimedia Commons.
- [66] Svetlana Volkova, Marta Matos, Matthias Mattanovich, and Igor Marin de Mas. Metabolic modelling as a framework for metabolomics data integration and analysis. *Metabolites*, 10:303, 07 2020. doi: 10.3390/metabo10080303.
- [67] Karthik Raman and Nagasuma Chandra. Flux balance analysis of biological systems: applications and challenges. *Briefings in Bioinformatics*, 10(4):435–449, 03 2009. ISSN 1467-5463. doi: 10.1093/bib/bbp011. URL <https://doi.org/10.1093/bib/bbp011>.
- [68] Michael MacGillivray, Amy Ko, Emily Gruber, Miranda Sawyer, Eivind Almaas, and Allen Holder. Robust analysis of fluxes in genome-scale metabolic pathways. *Scientific Reports*, 7(1):268, 2017. ISSN 2045-2322. doi: 10.1038/s41598-017-00170-3. URL <https://doi.org/10.1038/s41598-017-00170-3>.
- [69] Joshua A. M. Kaste and Yair Shachar-Hill. Model validation and selection in metabolic flux analysis and flux balance analysis. *arXiv*, March 2023. doi: 10.48550/arXiv.2303.12651. URL <https://arxiv.org/abs/2303.12651>. Preprint.
- [70] Carlos García Sánchez and Rodrigo Torres Sáez. Comparison and analysis of objective functions in flux balance analysis (fba). *Biotechnology Progress*, 30, 09 2014. doi: 10.1002/btpr.1949.
- [71] Michael Henson and Timothy Hanly. Dynamic flux balance analysis for synthetic microbial communities. *IET systems biology*, 8:214–229, 10 2014. doi: 10.1049/iet-syb.2013.0021.
- [72] Rafael D. de Oliveira, Galo A.C. Le Roux, and Radhakrishnan Mahadevan. Nonlinear programming reformulation of dynamic flux balance analysis models. *Computers Chemical Engineering*, 170:108101, 2023. ISSN 0098-1354. doi: <https://doi.org/10.1016/j.compchemeng.2022.108101>. URL <https://www.sciencedirect.com/science/article/pii/S0098135422004343>.

- [73] Kai Höffner, Stuart Harwood, and P Barton. A reliable simulator for dynamic flux balance analysis. *Biotechnology and bioengineering*, 110, 03 2013. doi: 10.1002/bit.24748.
- [74] Yuki Kuriya and Michihiro Araki. Dynamic flux balance analysis to evaluate the strain production performance on shikimic acid production in escherichia coli. *Metabolites*, 10:198, 05 2020. doi: 10.3390/metabo10050198.
- [75] Jose Gomez, Kai Höffner, and Paul Barton. Dfbalab: A fast and reliable matlab code for dynamic flux balance analysis. *BMC bioinformatics*, 15:409, 12 2014. doi: 10.1186/s12859-014-0409-8.
- [76] Mathias Gotsmy, Dafni Giannari, Radhakrishnan Mahadevan, and Jürgen Zanghellini. Optimizing fed-batch processes with dynamic control flux balance analysis. *IFAC-PapersOnLine*, 58(23):109–114, 2024. ISSN 2405-8963. doi: <https://doi.org/10.1016/j.ifacol.2024.10.019>. URL <https://www.sciencedirect.com/science/article/pii/S2405896324017658>. 10th IFAC Conference on Foundations of Systems Biology in Engineering FOSBE 2024.
- [77] Nathan E Lewis, Kim K Hixson, Tom M Conrad, Joshua A Lerman, Pep Charusanti, Ashoka D Polpitiya, Joshua N Adkins, Gunnar Schramm, Samuel O Purvine, Daniel Lopez-Ferrer, Karl K Weitz, Roland Eils, Rainer König, Richard D Smith, and Bernhard Ø Palsson. Omic data from evolved *e. coli* are consistent with computed optimal growth from genome-scale models. *Molecular Systems Biology*, 6(1):390, 2010. doi: <https://doi.org/10.1038/msb.2010.47>. URL <https://www.embopress.org/doi/abs/10.1038/msb.2010.47>.
- [78] Masood Khaksar Toroghi, William R Cluett, and Radhakrishnan Mahadevan. A multi-scale model of the whole human body based on dynamic parsimonious flux balance analysis. *IFAC-PapersOnLine*, 49(7):937–942, 2016. ISSN 2405-8963. doi: <https://doi.org/10.1016/j.ifacol.2016.07.319>. URL <https://www.sciencedirect.com/science/article/pii/S2405896316305262>. 11th IFAC Symposium on Dynamics and Control of Process Systems Including Biosystems DYCOPS-CAB 2016.
- [79] Milad Banitalebi Dehkordi, Mahmoud Reza Pishvaie, and Ehsan Vafa. Model predictive control of a fermenter using dynamic flux balance analysis coupled with convolutional neural networks. *Computers Chemical Engineering*, 179:108444, 2023. ISSN 0098-1354. doi: <https://doi.org/10.1016/j.compchemeng.2023.108444>. URL <https://www.sciencedirect.com/science/article/pii/S0098135423003149>.
- [80] Mathias Gotsmy, Anna Erian, Hans Marx, Stefan Pflügl, and Jürgen Zanghellini. Predictive dynamic control accurately maps the design space for 2,3-butanediol production, 08 2024.
- [81] Sungho Shin. Examodels and madnlp: Solving large-scale non-linear programs on gpus. <https://discourse.julialang.org/t/>

- `examodels-jl-and-madnlp-jl-on-gpus/111377`, 2024. Posted on JuliaLang Discourse forum, accessed 2025-03-14.
- [82] Tim Besard, Christophe Foket, and Bjorn De Sutter. Effective extensible programming: Unleashing Julia on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 2018. ISSN 1045-9219. doi: 10.1109/TPDS.2018.2872064.
 - [83] Cristóbal A. Navarro, Nancy Hitschfeld-Kahler, and Luis Mateu. A survey on parallel computing and its applications in data-parallel problems using gpu architectures. *Communications in Computational Physics*, 15(2):285–329, 2014. doi: 10.4208/cicp.110113.010813a.
 - [84] Cuda.jl documentation - scalar indexing, 2024. URL <https://cuda.juliagpu.org/stable/usage/workflow/>. Accessed: 2025-05-21.
 - [85] Tim Besard, Valentin Churavy, Alan Edelman, and Bjorn De Sutter. Rapid software prototyping for heterogeneous and distributed platforms. *Advances in Engineering Software*, 132:29–46, 2019.
 - [86] Cuda.jl documentation - array programming, 2024. URL <https://cuda.juliagpu.org/stable/usage/array/>. Accessed: 2025-05-21.
 - [87] Sertalp Çay, Imre Pólik, and Tamás Terlaky. Warm-start of interior point methods for second order cone optimization via rounding over optimal jordan frames, 05 2017.
 - [88] Ziyu Zhang, Qianchuan Zhao, and Fa-An Dai. A warm-start strategy in interior point methods for shrinking horizon model predictive control with variable discretization step. *IEEE Transactions on Automatic Control*, PP:1–8, 01 2022. doi: 10.1109/TAC.2022.3201026.
 - [89] Alexander Engau, Miguel F. Anjos, and Anthony Vannelli. A primal-dual slack approach to warmstarting interior-point methods for linear programming. In John W. Chinneck, Bjarni Kristjansson, and Matthew J. Saltzman, editors, *Operations Research and Cyber-Infrastructure*, pages 195–217, Boston, MA, 2009. Springer US. ISBN 978-0-387-88843-9.
 - [90] Jacek Gondzio and Andreas Grothey. A new unblocking technique to warmstart interior point methods based on sensitivity analysis. *SIAM Journal on Optimization*, 19(3):1184–1210, 2008. doi: 10.1137/060678129. URL <https://doi.org/10.1137/060678129>.
 - [91] Jacek Gondzio. Warm start of the primal-dual method applied in the cutting-plane scheme. *Mathematical Programming*, 83(1):125–143, 1998. ISSN 1436-4646. doi: 10.1007/BF02680554. URL <https://doi.org/10.1007/BF02680554>.

- [92] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, 2008. doi: 10.1109/JPROC.2008.917757.
- [93] Jens Krüger and Rüdiger Westermann. Linear algebra operators for gpu implementation of numerical algorithms. *ACM Trans. Graph.*, 22(3):908–916, July 2003. ISSN 0730-0301. doi: 10.1145/882262.882363. URL <https://doi.org/10.1145/882262.882363>.
- [94] Tim Besard, Christophe Foket, and Bjorn De Sutter. Effective extensible programming: Unleashing julia on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 30(4):827–841, 2019. doi: 10.1109/TPDS.2018.2872064.
- [95] Yu. E. Nesterov and M. J. Todd. Primal-dual interior-point methods for self-scaled cones. *SIAM Journal on Optimization*, 8(2):324–364, 1998. doi: 10.1137/S1052623495290209. URL <https://doi.org/10.1137/S1052623495290209>.
- [96] Reza Takapoui and Hamid Javadi. Preconditioning via diagonal scaling, 2016. URL <https://arxiv.org/abs/1610.03871>.
- [97] Juraj Kardoš, Drosos Kourounis, and Olaf Schenk. Structure exploiting interior point methods, 2019. URL <https://arxiv.org/abs/1907.05420>.
- [98] E. Hairer and G. Wanner. *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*. Springer Series in Computational Mathematics. Springer Berlin Heidelberg, 2010. ISBN 9783642052217. URL <https://books.google.at/books?id=g-jvCAAQBAJ>.
- [99] Colby Fronk and Linda Petzold. Training stiff neural ordinary differential equations with implicit single-step methods, 10 2024.
- [100] Pini Gurfil and Itzik Klein. Stabilizing the explicit euler integration of stiff and undamped linear systems. *Journal of Guidance Control and Dynamics - J GUID CONTROL DYNAM*, 30:1659–1667, 08 2007. doi: 10.2514/1.29148.
- [101] Topi Korhonen and Pasi Tavi. Automatic time-step adaptation of the forward euler method in simulation of models of ion channels and excitable cells and tissue. *Simulation Modelling Practice and Theory*, 16(6):639–644, 2008. ISSN 1569-190X. doi: <https://doi.org/10.1016/j.simpat.2008.03.007>. URL <https://www.sciencedirect.com/science/article/pii/S1569190X08000634>.

Appendix A

Supplementary Material

Software

The original implementation used Julia Version **1.8.5** (2023-01-08). The new packages we wanted to test require a higher version of Julia. Therefore we are working with Julia version **1.10.5** (2024-08-27). The following table presents the respective installed package and their versions:

Package	v1.8.5	v1.10.5
CSV	0.10.14	0.10.15
CUDA	5.4.3	5.6.1 (dev)
DataFrames	1.6.1	1.7.0
DelimitedFiles	–	1.9.1
Dierckx	0.5.3	0.5.4
ExaModels	–	0.8.0
ForwardDiff	0.10.36	0.10.38
IJulia	1.25.0	1.26.0
IPOPT	1.6.6	1.7.1
JLD2	0.5.8	0.5.11
JuMP	1.23.1	1.23.6
MadNLP	0.7.0	0.8.5
MadNLPGPU	0.7.3	0.7.6
MadNLPHSL	0.5.2	–
NLPModelsIPOPT	–	0.10.2
Plots	1.40.8	1.40.9
SIMD	–	3.7.1

Table A.1: Comparison of installed Julia packages and versions between the original and updated implementations

Data

Option Testing for DCO-BUT

File	Options	Termination Status	Time Elapsed
original	warm_start_init_point => yes, print_level => 5, linear_solver => ma27, max_iter => Int(1e5), tol => 1e-6, acceptable_iter => 50, acceptable_tol => 1e-6	locally solved	37.75688004

Initialization

File	Options added to original settings	Termination Status	Time Elapsed
_init_combi1	bound_push => 0.02, bound_frac => 0.02	numerical error	219.872502
_init_combi2	bound_push => 0.002, bound_frac => 0.002	locally solved	8.155020
_init_combi3	bound_push => 0.005, bound_frac => 0.005	locally solved	17.734669
_init_combi4	bound_push => 0.02, bound_frac => 0.002	locally solved	24.839642
_init_combi5	bound_push => 0.002, bound_frac => 0.02	locally solved	27.926222
_init_combi6	slack_bound_push => 0.02, slack_bound_frac => 0.02	locally solved	35.701368
_init1_1	bound_push => 0.002	locally solved	23.030438
_init1_2	bound_push => 0.02	locally solved	72.001554
_init1_3	bound_push => 0.0001	locally solved	48.903838
_init1_4	bound_push => 0.03	locally solved	355.272660
_init1_5	bound_push => 0.003	locally solved	25.450764
_init2_1	bound_frac => 0.1	numerical error	1148.580079
_init2_2	bound_frac => 0.02	locally solved	640.222667
_init2_3	bound_frac => 0.03	locally solved	13.544733
_init3_1	slack_bound_push => 0.02	locally solved	34.989186
_init3_2	slack_bound_push => 0.03	locally solved	37.479663
_init3_3	slack_bound_push => 0.1	locally solved	33.412059
_init4_1	slack_bound_frac => 0.02	locally solved	38.914980
_init4_2	slack_bound_frac => 0.5	locally solved	33.865397

Barrier Parameter

File	Options added to original settings	Termination Status	Time Elapsed
_barrier1_1	mu_strategy => adaptive	locally solved	20.007734
_barrier1_2	mu_strategy => adaptive, mu_oracle => probing	locally solved	39.859928
_barrier1_3	mu_strategy => adaptive, mu_oracle => loqo	locally solved	23.524486
_barrier1_4	mu_strategy => adaptive, mu_oracle => quality-function	locally solved	22.412152
_barrier2_1	mu_strategy => adaptive, mu_max_fact => 500.0	locally solved	27.831788
_barrier2_2	mu_strategy => adaptive, mu_max_fact => 1500.0	locally solved	100.746391
_barrier2_3	mu_strategy => adaptive, mu_max_fact => 2000.0	locally solved	26.616612
_barrier3_1	mu_strategy => adaptive, mu_min => 1e-10, mu_max => 50000.0	locally solved	82.367090
_barrier3_1_1	mu_strategy => adaptive, mu_min => 1e-12, mu_max => 50000.0	locally solved	128.903224
_barrier3_1_2	mu_strategy => adaptive, mu_min => 1e-13, mu_max => 50000.0	locally solved	87.176043
_barrier3_1_3	mu_strategy => adaptive, mu_min => 1e-14, mu_max => 50000.0	locally solved	46.789541
_barrier3_2	mu_strategy => adaptive, mu_min => 1e-10, mu_max => 200000.0	locally solved	84.895981
_barrier3_2_1	mu_strategy => adaptive, mu_min => 1e-12, mu_max => 200000.0	locally solved	155.555292
_barrier4_1	mu_strategy => adaptive, adaptive_mu_globalization => kkt-error	locally solved	19.287146
_barrier4_2	mu_strategy => adaptive, adaptive_mu_globalization => never-monotone-mode	locally solved	20.471800
_barrier4_3	mu_strategy => adaptive, adaptive_mu_globalization => obj-constr-filter	locally solved	21.031635

MA27 solver

File	Options added to original settings	Termination Status	Time Elapsed
_ma27_1_1	ma27_pivtol => 1e-7	locally solved	10.755168
_ma27_1_2	ma27_pivtol => 1e-9	locally solved	10.214215
_ma27_1_3	ma27_pivtol => 1e-6	locally solved	15.232976
_ma27_1_4	ma27_pivtol => 0.99 (<1)	locally solved	81.925986
_ma27_2_1	ma27_pivtolmax => 0.001	locally solved	39.748941
_ma27_2_2	ma27_pivtolmax => 0.0005	locally solved	36.841080

NLP

File	Options added to original settings	Termination Status	Time Elapsed
_nlp1_1	nlp_lower_bound_inf => -1e20, nlp_upper_bound_inf => 1e20	locally solved	39.500147
_nlp1_2	nlp_lower_bound_inf => -1e18, nlp_upper_bound_inf => 1e18	locally solved	38.922271
_nlp2_1	fixed_variable_treatment => relax_bounds	locally solved	36.970973
_nlp2_2	fixed_variable_treatment => make_constraint	locally solved	37.103988
_nlp2_3	fixed_variable_treatment => make_parameter_nodual	locally solved	41.171283
_nlp_set1_1	nlp_lower_bound_inf => -1e18, nlp_upper_bound_inf => 1e18, fixed_variable_treatment => relax_bounds	locally solved	33.559155
_nlp_set1_2	nlp_lower_bound_inf => -1e18, nlp_upper_bound_inf => 1e18, fixed_variable_treatment => make_constraint	locally solved	34.077148
_nlp_set1_3	nlp_lower_bound_inf => -1e18, nlp_upper_bound_inf => 1e18, fixed_variable_treatment => make_parameter_nodual	locally solved	35.800896
_nlp_set2_1	nlp_lower_bound_inf => -1e20, nlp_upper_bound_inf => 1e20, fixed_variable_treatment => relax_bounds	locally solved	39.769118
_nlp_set2_2	nlp_lower_bound_inf => -1e20, nlp_upper_bound_inf => 1e20, fixed_variable_treatment => make_constraint	locally solved	38.730812
_nlp_set2_3	nlp_lower_bound_inf => -1e20, nlp_upper_bound_inf => 1e20, fixed_variable_treatment => make_parameter_nodual	locally solved	40.503626

Termination

File	Options added to original settings	Termination Status	Time Elapsed
_term_1_1	dual_inf_tol => 1e-4	locally solved	42.662879
_term_1_2	dual_inf_tol => 1e-6	locally solved	36.786181
_term_1_3	dual_inf_tol => 1e-6	locally solved	41.764213
_term_2_1	constr_viol_tol => 1e-4	locally solved	42.362455
_term_2_2	constr_viol_tol => 1e-5	locally solved	40.727690
_term_2_3	constr_viol_tol => 1e-6	locally solved	31.485907
_term_3_1	compl_inf_tol => 1e-3	locally solved	43.134871
_term_3_2	compl_inf_tol => 1e-5	locally solved	36.533393
_term_3_3	compl_inf_tol => 1e-6	locally solved	41.802793

Warm Start

File	Options added to original settings	Termination Status	Time Elapsed
warmst-combi1	warm_start_bound_push" => 0.0002, warm_start_bound_frac" => 0.0002	locally solved	38.022616
warmst-combi2	warm_start_bound_push" => 0.002, warm_start_bound_frac" => 0.002	locally solved	48.979694
_warmst1_1	warm_start_bound_push => 0.002	locally solved	7.806066
_warmst1_2	warm_start_bound_push => 0.003	locally solved	14.670347
_warmst1_3	warm_start_bound_push => 0.0001	locally solved	35.281279
_warmst2_1	warm_start_mult_bound_push => 0.002	locally solved	207.589725
_warmst2_2	warm_start_mult_bound_push => 0.003	locally solved	755.828428
_warmst2_3	warm_start_mult_bound_push => 0.0001	locally solved	34.572824
_warmst2_4	warm_start_mult_bound_push => 0.004	numerical error	958.944726
_warmst3_1	warm_start_bound_push => 0.002	locally solved	9.108752
_warmst3_2	warm_start_bound_push => 0.003	locally solved	18.022576
_warmst3_3	warm_start_bound_push => 0.004	locally solved	31.052439
_warmst3_4	warm_start_bound_push => 0.0001	locally solved	36.442103
_warmst4_1	warm_start_slack_bound_frac => 0.002	locally solved	35.455754
_warmst4_2	warm_start_slack_bound_frac => 0.5	locally solved	41.346887
_warmst4_3	warm_start_slack_bound_frac => 0.0001	locally solved	39.578350
_warmst5_1	warm_start_slack_bound_push=> 0.002	locally solved	34.194773
_warmst5_2	warm_start_slack_bound_push=> 0.0001	locally solved	51.415721
_warmst5_3	warm_start_slack_bound_push=> 0.003	locally solved	253.234822
_warmst5_4	warm_start_slack_bound_push=> 0.004	locally solved	390.951186
_warmst6_1	warm_start_bound_frac => 0.002	locally solved	28.444991
_warmst6_2	warm_start_bound_frac => 0.5	numerical error	1309.145556
warmst6-2_1	warm_start_bound_frac => 0.4	numerical error	1417.892771
_warmstart6_2_2	warm_start_bound_frac => 0.5, warm_start_bound_push => 0.002	locally solved	21.427629
_warmst6_3	warm_start_bound_frac => 0.0001	locally solved	160.764786
_warmst6_4	warm_start_bound_frac => 0.1	numerical error	1186.746675
_warmst6_5	warm_start_bound_frac => 0.09	numerical error	1157.712913
warmst6-5_1	warm_start_bound_frac => 0.09, warm_start_bound_push => 0.9	locally solved	35.195810

Sets

File	Options added to original settings	Termination Status	Time Elapsed
test_script_-set1_1	bound_relax_factor => 1.0e-7, adaptive_mu_globalization => kkt-error, nlp_upper_bound_inf => 1.0e18, mu_max => 50000.0, mu_max_fact => 500.0, ma27_pivtol => 1.0e-7, nlp_lower_bound_inf => -1.0e18, ma27_pivtolmax => 0.001, mu_strategy => adaptive, mu_min => 1.0e-10,	locally solved	27.780853
test_script_-set1_1_1	bound_relax_factor => 1.0e-7, fixed_variable_treatment => make_constraint, adaptive_mu_globalization => kkt-error, nlp_upper_bound_inf => 1.0e18, mu_max => 50000.0, mu_max_fact => 500.0, ma27_pivtol => 1.0e-7, nlp_lower_bound_inf => -1.0e18, ma27_pivtolmax => 0.001, mu_strategy => adaptive, mu_min => 1.0e-10	locally solved	28.005315
test_script_-set1_1_2	bound_relax_factor => 1.0e-7, fixed_variable_treatment => relax_bounds, adaptive_mu_globalization => kkt-error, nlp_upper_bound_inf => 1.0e18, mu_max => 50000.0, mu_max_fact => 500.0, ma27_pivtol => 1.0e-7, nlp_lower_bound_inf => -1.0e18, ma27_pivtolmax => 0.001, mu_strategy => adaptive, mu_min => 1.0e-10	locally solved	27.945659
test_script_-set1_2	bound_relax_factor => 1.0e-7, adaptive_mu_globalization => kkt-error, nlp_upper_bound_inf => 1.0e18, mu_max => 50000.0, mu_max_fact => 500.0, ma27_pivtol => 1.0e-7, nlp_lower_bound_inf => -1.0e18, ma27_pivtolmax => 0.001, mu_strategy => adaptive, mu_min => 1.0e-10, ma27_skip_inertia_check => yes	numerical error	13.803799
test_script_-set1_2_1	bound_relax_factor => 1.0e-9, fixed_variable_treatment => make_constraint, nlp_upper_bound_inf => 1.0e20, mu_max => 200000.0, mu_max_fact => 1500.0, ma27_pivtol => 1.0e-8, nlp_lower_bound_inf => -1.0e20, ma27_pivtolmax => 0.0005, mu_strategy => adaptive, mu_min => 1.0e-12	numerical error	13.961371
test_script_-set1_2_2	bound_relax_factor => 1.0e-9, fixed_variable_treatment => relax_bounds, nlp_upper_bound_inf => 1.0e20, mu_max => 200000.0, mu_max_fact => 1500.0, ma27_pivtol => 1.0e-8, nlp_lower_bound_inf => -1.0e20, ma27_pivtolmax => 0.0005, mu_strategy => adaptive, mu_min => 1.0e-12	locally solved	101.375528
test_script_-set1_3	bound_relax_factor => 1.0e-9, adaptive_mu_globalization => kkt-error, nlp_upper_bound_inf => 1.0e20, mu_max => 200000.0, mu_max_fact => 1500.0, ma27_pivtol => 1.0e-8, nlp_lower_bound_inf => -1.0e20, ma27_pivtolmax => 0.0005, mu_strategy => adaptive, mu_min => 1.0e-12, ma27_skip_inertia_check => yes	numerical error	6.091224

Continued on next page

File	Options added to original settings	Termination Status	Time Elapsed
test_script_-set1_3_1	bound_relax_factor => 1.0e-9, adaptive_mu_globalization => kkt-error, nlp_upper_bound_inf => 1.0e20, mu_max => 200000.0, mu_max_fact => 1500.0, ma27_pivtol => nlp_lower_bound_inf => -1.0e20, ma27_pivtolmax => 0.0005, mu_strategy => adaptive, mu_min => 1.0e-12	locally solved	13.396965

Option Testing for DCO-PDN

File	Options	Termination Status	Time Elapsed
original	warm_start_init_point => yes, print_level => 5, linear_solver => ma27, max_iter => Int(1e5), tol => 1e-4, acceptable_iter => 15, acceptable_tol => 1e-2	locally solved	84.5774660110473

Initialization

File	Options added to original settings	Termination Status	Time Elapsed
_init_combi1	bound_push => 0.02, bound_frac => 0.02	locally solved	85.527812
_init_combi2	bound_push => 0.002, bound_frac => 0.002	locally solved	74.438062
_init_combi3	bound_push => 0.005, bound_frac => 0.005	locally solved	150.263304
_init_combi4	bound_push => 0.02, bound_frac => 0.002	locally solved	407.952839
_init_combi5	bound_push => 0.002, bound_frac => 0.02	locally solved	413.822878
_init_combi6	slack_bound_push => 0.02, slack_bound_frac => 0.02	locally solved	113.874844
_init1_1	bound_push => 0.02	locally solved	58.827105
_init1_2	bound_push => 0.002	almost-locally solved	169.787589
_init1_3	bound_push => 0.0001	locally solved	218.750171
_init1_4	bound_push => 0.03	almost-locallysolved	2262.614914
_init1_5	bound_push => 0.003	almost-locallysolved	168.076116
_init2_1	bound_frac => 0.1	almost-locallysolved	116.693525
_init2_2	bound_frac => 0.02	locally solved	75.585706
_init2_3	bound_frac => 0.03	almost-locallysolved	193.671177
_init3_1	slack_bound_push => 0.02	locally solved	87.783950

Continued on next page

File	Options added to original settings	Termination Status	Time Elapsed
_init3_2	slack_bound_push => 0.03	locally solved	88.935759
_init3_3	slack_bound_push => 0.1	locally solved	86.737355
_init4_1	slack_bound_frac => 0.02	locally solved	85.822191
_init4_2	slack_bound_frac => 0.5	locally solved	85.645885

Barrier Parameter

File	Options added to original settings	Termination Status	Time Elapsed
_barrier1_1	mu_strategy => adaptive	locally solved	83.669690
_barrier1_2	mu_strategy => adaptive, mu_oracle => probing	locally solved	272.673712
_barrier1_3	mu_strategy => adaptive, mu_oracle => loqo	almost-locally solved	141.254581
_barrier1_4	mu_strategy => adaptive, mu_oracle => quality-function	locally solved	90.742598
_barrier2_1	mu_strategy => adaptive, mu_max_fact => 500.0	locally solved	88.731549
_barrier2_2	mu_strategy => adaptive, mu_max_fact => 1500.0	locally solved	92.028869
_barrier2_3	mu_strategy => adaptive, mu_max_fact => 2000.0	locally solved	83.657547
_barrier3_1	mu_strategy => adaptive, mu_min => 1e-10, mu_max => 50000.0	almost-locally solved	236.597175
_barrier3_1_1	mu_strategy => adaptive, mu_min => 1e-12, mu_max => 50000.0	locally solved	64.073825
_barrier3_1_2	mu_strategy => adaptive, mu_min => 1e-13, mu_max => 50000.0	locally solved	120.531188
_barrier3_1_3	mu_strategy => adaptive, mu_min => 1e-14, mu_max => 50000.0	locally solved	76.241428
_barrier3_2	mu_strategy => adaptive, mu_min => 1e-10, mu_max => 200000.0	almost-locally solved	234.754454
_barrier3_2_1	mu_strategy => adaptive, mu_min => 1e-12, mu_max => 200000.0	locally solved	64.978402
_barrier4_1	mu_strategy => adaptive, adaptive_mu_globalization => kkt-error	locally solved	258.643766
_barrier4_2	mu_strategy => adaptive, adaptive_mu_globalization => never-monotone-mode	locally solved	79.077668
_barrier4_3	mu_strategy => adaptive, adaptive_mu_globalization => obj-constr-filter	locally solved	94.460678

MA27 solver

File	Options added to original settings	Termination Status	Time Elapsed
_ma27_1_1	ma27_pivtol => 1e-7	almost-locally solved	141.006655
_ma27_1_2	ma27_pivtol => 1e-9	locally solved	78.889802
_ma27_1_3	ma27_pivtol => 1e-6	iteration limit	1014.871370
_ma27_1_4	ma27_pivtol => 0.99 (<1)	locally solved	630.610368
_ma27_2_1	ma27_pivtolmax => 0.001	locally solved	186.251466
_ma27_2_2	ma27_pivtolmax => 0.0005	locally solved	192.404274

NLP

File	Options added to original settings	Termination Status	Time Elapsed
_nlp1_1	nlp_lower_bound_inf => -1e20, nlp_upper_bound_inf => 1e20	locally solved	85.026898
_nlp1_2	nlp_lower_bound_inf => -1e18, nlp_upper_bound_inf => 1e18	locally solved	83.232627
_nlp2_1	fixed_variable_treatment => relax_bounds	locally solved	82.566306
_nlp2_2	fixed_variable_treatment => make_constraint	locally solved	84.614874
_nlp2_3	fixed_variable_treatment => make_parameter_nodual	locally solved	82.721417
_nlp_set1_1	nlp_lower_bound_inf => -1e18, nlp_upper_bound_inf => 1e18, fixed_variable_treatment => relax_bounds	locally solved	83.817813
_nlp_set1_2	nlp_lower_bound_inf => -1e18, nlp_upper_bound_inf => 1e18, fixed_variable_treatment => make_constraint	locally solved	82.582818
_nlp_set1_3	nlp_lower_bound_inf => -1e18, nlp_upper_bound_inf => 1e18, fixed_variable_treatment => make_parameter_nodual	locally solved	83.709894
_nlp_set2_1	nlp_lower_bound_inf => -1e20, nlp_upper_bound_inf => 1e20, fixed_variable_treatment => relax_bounds	locally solved	89.574309
_nlp_set2_2	nlp_lower_bound_inf => -1e20, nlp_upper_bound_inf => 1e20, fixed_variable_treatment => make_constraint	locally solved	86.499969
_nlp_set2_3	nlp_lower_bound_inf => -1e20, nlp_upper_bound_inf => 1e20, fixed_variable_treatment => make_parameter_nodual	locally solved	91.971341

Termination

File	Options added to original settings	Termination Status	Time Elapsed
_term_1_1	dual_inf_tol => 1e-4	locally solved	918.014638
_term_1_2	dual_inf_tol => 1e-6	locally solved	344.668831
_term_1_3	dual_inf_tol => 1e-6	locally solved	82.391111
_term_2_1	constr_viol_tol => 1e-4	locally solved	84.216000
_term_2_2	constr_viol_tol => 1e-5	locally solved	85.303132
_term_2_3	constr_viol_tol => 1e-6	locally solved	184.955586
_term_3_1	compl_inf_tol => 1e-3	locally solved	82.668375
_term_3_2	compl_inf_tol => 1e-5	locally solved	81.769473
_term_3_3	compl_inf_tol => 1e-6	locally solved	82.331229

Warm Start

File	Options added to original settings	Termination Status	Time Elapsed
warmst-combi1	warm_start_bound_push" => 0.0002, warm_start_bound_frac" => 0.0002	locally solved	164.717945
warmst-combi2	warm_start_bound_push" => 0.002, warm_start_bound_frac" => 0.002	locally solved	168.974709
_warmst1_1	warm_start_bound_push => 0.002	locally solved	441.041739
_warmst1_2	warm_start_bound_push => 0.003	almost-locally solved	93.129288
_warmst1_3	warm_start_bound_push => 0.0001	locally solved	510.737985
_warmst2_1	warm_start_mult_bound_push => 0.002	locally solved	220.540217
_warmst2_2	warm_start_mult_bound_push => 0.003	locally solved	209.403775
_warmst2_3	warm_start_mult_bound_push => 0.0001	almost-locally solved	292.947777
_warmst2_4	warm_start_mult_bound_push => 0.004	almost-locally solved	138.704461
_warmst3_1	warm_start_bound_push => 0.002	NaN	NaN
_warmst3_2	warm_start_bound_push => 0.003	NaN	NaN
_warmst3_3	warm_start_bound_push => 0.004	NaN	NaN
_warmst3_4	warm_start_bound_push => 0.0001	NaN	NaN
_warmst4_1	warm_start_slack_bound_frac => 0.002	locally solved	80.619573
_warmst4_2	warm_start_slack_bound_frac => 0.5	locally solved	89.354160
_warmst4_3	warm_start_slack_bound_frac => 0.0001	locally solved	82.169910
_warmst5_1	warm_start_slack_bound_push => 0.002	locally solved	90.090864
_warmst5_2	warm_start_slack_bound_push => 0.0001	locally solved	440.820690
_warmst5_3	warm_start_slack_bound_push => 0.003	almost-locally solved	154.484579

Continued on next page

File	Options added to original settings	Termination Status	Time Elapsed
_warmst5_4	warm_start_slack_bound_push=> 0.004	almost-locally solved	148.596463
_warmst6_1	warm_start_bound_frac => 0.002	locally solved	183.821482
_warmst6_2	warm_start_bound_frac => 0.5	almost-locally solved	118.608571
warmst6-2_1	warm_start_bound_frac => 0.4	other error	NaN
_warm-start6_2_2	warm_start_bound_frac => 0.5, warm_start_bound_-push => 0.002	other error	NaN
_warmst6_3	warm_start_bound_frac => 0.0001	other error	NaN
_warmst6_4	warm_start_bound_frac => 0.1	other error	NaN
_warmst6_5	warm_start_bound_frac => 0.09	other error	NaN
warmst6-5_1	warm_start_bound_frac => 0.09, warm_start_bound_-push => 0.9	other error	NaN

Sets

File	Options added to original settings	Termination Status	Time Elapsed
test_script_-set1_1	bound_relax_factor => 1.0e-7, adaptive_mu_globalization => kkt-error, nlp_upper_bound_inf => 1.0e18, mu_max => 50000.0, mu_max_fact => 500.0, ma27_pivtol => 1.0e-7, nlp_lower_bound_inf => -1.0e18, ma27_pivtolmax => 0.001, mu_strategy => adaptive, mu_min => 1.0e-10,	locally solved	307.631803
test_script_-set1_1_1	bound_relax_factor => 1.0e-7, fixed_variable_treatment => make_constraint, adaptive_mu_globalization => kkt-error, nlp_upper_bound_inf => 1.0e18, mu_max => 50000.0, mu_max_fact => 500.0, ma27_pivtol => 1.0e-7, nlp_lower_bound_inf => -1.0e18, ma27_pivtolmax => 0.001, mu_strategy => adaptive, mu_min => 1.0e-10	locally solved	324.228162
test_script_-set1_1_2	bound_relax_factor => 1.0e-7, fixed_variable_treatment => relax_bounds, adaptive_mu_globalization => kkt-error, nlp_upper_bound_inf => 1.0e18, mu_max => 50000.0, mu_max_fact => 500.0, ma27_pivtol => 1.0e-7, nlp_lower_bound_inf => -1.0e18, ma27_pivtolmax => 0.001, mu_strategy => adaptive, mu_min => 1.0e-10	locally solved	319.109225
Continued on next page			

File	Options added to original settings	Termination Status	Time Elapsed
test_script_-set1_2	bound_relax_factor => 1.0e-7, adaptive_mu_globalization => kkt-error, nlp_upper_bound_inf => 1.0e18, mu_max => 50000.0, mu_max_fact => 500.0, ma27_pivtol => 1.0e-7, nlp_lower_bound_inf => -1.0e18, ma27_pivtolmax => 0.001, mu_strategy => adaptive, mu_min => 1.0e-10, ma27_skip_inertia_check => yes	locally solved	311.374419
test_script_-set1_2_1	bound_relax_factor => 1.0e-9, fixed_variable_treatment => make_constraint, nlp_upper_bound_inf => 1.0e20, mu_max => 200000.0, mu_max_fact => 1500.0, ma27_pivtol => 1.0e-8, nlp_lower_bound_inf => -1.0e20, ma27_pivtolmax => 0.0005, mu_strategy => adaptive, mu_min => 1.0e-12	locally solved	149.013968
test_script_-set1_2_2	bound_relax_factor => 1.0e-9, fixed_variable_treatment => relax_bounds, nlp_upper_bound_inf => 1.0e20, mu_max => 200000.0, mu_max_fact => 1500.0, ma27_pivtol => 1.0e-8, nlp_lower_bound_inf => -1.0e20, ma27_pivtolmax => 0.0005, mu_strategy => adaptive, mu_min => 1.0e-12	locally solved	152.386315
test_script_-set1_3	bound_relax_factor => 1.0e-9, adaptive_mu_globalization => kkt-error, nlp_upper_bound_inf => 1.0e20, mu_max => 200000.0, mu_max_fact => 1500.0, ma27_pivtol => 1.0e-8, nlp_lower_bound_inf => -1.0e20, ma27_pivtolmax => 0.0005, mu_strategy => adaptive, mu_min => 1.0e-12, ma27_skip_inertia_check => yes	locally solved	262.334085
test_script_-set1_3_1	bound_relax_factor => 1.0e-9, adaptive_mu_globalization => kkt-error, nlp_upper_bound_inf => 1.0e20, mu_max => 200000.0, mu_max_fact => 1500.0, ma27_pivtol => nlp_lower_bound_inf => -1.0e20, ma27_pivtolmax => 0.0005, mu_strategy => adaptive, mu_min => 1.0e-12	locally solved	275.430708

MadNLP Performance

MadNLP + JuMP + ExaModels (CPU) DCO-BUT

File	Options added to original settings	Termination Status	Iteration	final Objective value
option set 1	tol => 1e-7, max_iter => 5000, required_infeasibility_reduction => 0.6, acceptable_tol => 1e-7, acceptable_iter => 10, barrier_tol_factor => 20.0, soft_resto_pder-ror_reduction_factor => 0.7, output_file => "INFO"	max.	Iteration	-2.16e+07
option set 2	tol => 1e-6, max_iter => 5000, required_infeasibility_reduction => 0.6, acceptable_tol => 1e-5, acceptable_iter => 15, barrier_tol_factor => 20.0, soft_resto_pder-ror_reduction_factor => 0.7, output_file => "INFO"	max.	Iteration	-2.89e+02
option set 3	tol => 1e-6, max_iter => 10000, required_infeasibility_reduction => 0.6, acceptable_tol => 1e-5, acceptable_iter => 10, output_file => "INFO"	max.	Iteration	-1.12e+08
option set 4	tol => 1e-6, max_iter => 7000, required_infeasibility_reduction => 0.7, acceptable_tol => 1e-6, acceptable_iter => 10	max.	Iteration	-7.54e+07
option set 5	tol => 1e-6, max_iter => 5000, required_infeasibility_reduction => 0.9, acceptable_tol => 1e-6, acceptable_iter => 10, output_file => "INFO"	max.	Iteration	-8.58e+04
option set 6	tol => 1e-6, max_iter => 5000, required_infeasibility_reduction => 0.8, acceptable_tol => 1e-6, acceptable_iter => 15, output_file => "INFO"	max.	Iteration	-8.58e+04
option set 7	tol => 1e-6, max_iter => 5000, required_infeasibility_reduction => 0.5, acceptable_tol => 1e-5, acceptable_iter => 10, barrier_tol_factor => 20.0, soft_resto_pder-ror_reduction_factor => 0.75, output_file => "INFO"	max.	Iteration	-4.95e+02
option set 8	tol => 1e-7, max_iter => 5000, required_infeasibility_reduction => 0.7, acceptable_tol => 1e-7, acceptable_iter => 10, barrier_tol_factor => 20.0, soft_resto_pder-ror_reduction_factor => 0.7, output_file => "INFO"	Optimal Solution		-7.75e+02
option set 9	tol => 1e-6, max_iter => 5000, required_infeasibility_reduction => 0.5, acceptable_tol => 1e-5, acceptable_iter => 10, barrier_tol_factor => 20.0, output_file => "INFO"	Optimal Solution		-7.75e+02

MadNLP + JuMP + ExaModels (GPU) DCO-BUT

File	Options added to original settings	Termination Status		final Objective value
option set 1	tol = 1e-6, max_iter = 5000, required_infeasibility_reduction = 0.5, acceptable_tol = 1e-5, acceptable_iter = 10, barrier_tol_factor = 20.0, soft_resto_pderror_reduction_factor = 0.75, output_file = "INFO"	max. Iteration	-	5.3248e+4
option set 2	tol = 1e-7, max_iter = 5000, required_infeasibility_reduction = 0.7, acceptable_tol = 1e-7, acceptable_iter = 10, barrier_tol_factor = 20.0, soft_resto_pderror_reduction_factor = 0.7, output_file = "INFO"	max. Iteration	-	8.7454e+4
option set 3	tol = 1e-7, max_iter = 5000, required_infeasibility_reduction = 0.6, acceptable_tol = 1e-7, acceptable_iter = 10, barrier_tol_factor = 20.0, soft_resto_pderror_reduction_factor = 0.7, output_file = "INFO"	max. Iteration	-	1.3581e+4
option set 4	tol = 1e-6, max_iter = 5000, required_infeasibility_reduction = 0.6, acceptable_tol = 1e-5, acceptable_iter = 15, barrier_tol_factor = 20.0, soft_resto_pderror_reduction_factor = 0.7, output_file = "INFO"	max. Iteration	-	5.5248e+4
option set 5	tol = 1e-6, max_iter = 5000, required_infeasibility_reduction = 0.3, acceptable_tol = 1e-6, acceptable_iter = 10, barrier_tol_factor = 20.0, output_file = "INFO"	max. Iteration	-	4.8945e+4
option set 6	tol = 1e-6, max_iter = 5000, required_infeasibility_reduction = 0.5, acceptable_tol = 1e-6, acceptable_iter = 10, barrier_tol_factor = 20.0, output_file = "INFO"	max. Iteration	-	5.5965e+4
option set 7	tol = 1e-4, max_iter = 5000, required_infeasibility_reduction = 0.5, acceptable_tol = 1e-4, acceptable_iter = 15, barrier_tol_factor = 30.0, output_file = "INFO"	max. Iteration	-	8.7997e+4
option set 8	tol = 1e-9, max_iter = 5000, required_infeasibility_reduction = 0.1, acceptable_tol = 1e-9, acceptable_iter = 10, output_file = "INFO"	max. Iteration	-	1.4082e+4
option set 9	tol = 1e-9, max_iter = 5000, required_infeasibility_reduction = 0.2, acceptable_tol = 1e-9, acceptable_iter = 10, output_file = "INFO"	max. Iteration	-	6.6380e+4
option set 10	tol = 1e-9, max_iter = 5000, required_infeasibility_reduction = 0.3, acceptable_tol = 1e-9, acceptable_iter = 10, output_file = "INFO"	max. Iteration	-	5.8716e+4
option set 11	max_iter = 10000, tol = 1e-11, acceptable_tol = 1e-10, required_infeasibility_reduction = 1.0, nlp_scaling = true	max. Iteration	-	8.8672e+3
option set 12	max_iter = 10000, tol = 1e-11, acceptable_tol = 1e-10, acceptable_iter = 10, required_infeasibility_reduction = 1.0, nlp_scaling = true, output_file = "INFO"	max. Iteration	-	9.4676e+4
option set 13	max_iter = 10000, tol = 1e-11, acceptable_tol = 1e-8, required_infeasibility_reduction = 0.3, output_file = "INFO"	max. Iteration	-	9.3251e+4