NC2C: AUTOMATED CONVEXIFICATION OF GENERIC NON-CONVEX OPTIMIZATION PROBLEMS

Anonymous authors
Paper under double-blind review

ABSTRACT

Non-convex optimization problems are pervasive across mathematical programming, engineering design, and scientific computing, often posing intractable challenges for traditional solvers due to their complex objective functions and constrained landscapes. To address the inefficiency of manual convexification and the over-reliance on expert knowledge, we propose NC2C, an LLM-based end-to-end automated framework designed to transform generic non-convex optimization problems into solvable convex forms using large language models. NC2C leverages LLMs' mathematical reasoning capabilities to autonomously detect non-convex components, select optimal convexification strategies, and generate rigorous convex equivalents. The framework integrates symbolic reasoning, adaptive transformation techniques, and iterative validation, equipped with error correction loops and feasibility domain correction mechanisms. We evaluate NC2C with two leading LLMs (GPT-5.1 and Qwen3-235B-A22B) across four diverse benchmarks: NL4Opt, NLP4LP, ComplexOR, and our curated WireOpt dataset that aggregates non-convex problems drawn from mathematical programming, engineering design, and scientific computing. NC2C consistently outperforms state-of-the-art baselines including Vanilla prompting, Chain-of-Experts, OptiMUS, and Reflexion on both success rate and execution rate. With GPT-5.1, NC2C achieves the best results across all datasets, peaking at a 94.5% success rate and 96.7% execution rate on NL4Opt and 87.0% / 93.0% on the WireOpt benchmark. Ablation studies confirm the critical role of convexification, error correction, and feasibility verification within the framework. These results highlight NC2C's ability to leverage LLMs for automated non-convex to convex transformation, reduce expert dependency, and enable efficient deployment of convex solvers for previously intractable optimization tasks.

1 INTRODUCTION

Non-convex optimization problems are ubiquitous across various domains, including mathematical programming, engineering design, machine learning, and scientific computing (Sui et al., 2024; Mikhalevich et al., 2024). Convex optimization offers stable and efficient solutions, especially when closed-form solutions are available (Garatti & Campi, 2025). However, many real-world optimization problems are inherently non-convex, characterized by complex objective functions, non-linear constraints, and coupled variables that create multiple local optima (Liu et al., 2024). Addressing these challenges often requires sophisticated modeling, transformation, and relaxation techniques to convert non-convex problems into convex ones, typically depending heavily on expert knowledge in advanced mathematical optimization, problem-specific domain expertise, and transformation strategies.

To reduce reliance on expert knowledge, recent research shows that large language models (LLMs) achieve significant performance on complex mathematical problems. For instance, recent prompting methods push LLM accuracy on GSM8K above 97% (Zhong et al., 2024), and a series of LLM-based provers, such as Seed-Prover (Chen et al., 2025b;a), demonstrate strong relational reasoning over complex proof lemmas. LLMs can also address classical optimization problems, such as linear regression and the traveling salesman problem, through iterative methods (Yang et al., 2024). Moreover, LLMs can leverage existing solvers like `Gurobi` and `CVXPY` for optimization tasks (Diamond & Boyd, 2016); for example, the `gurobi.abs` function enables direct modeling of ℓ_1-norm ob-

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

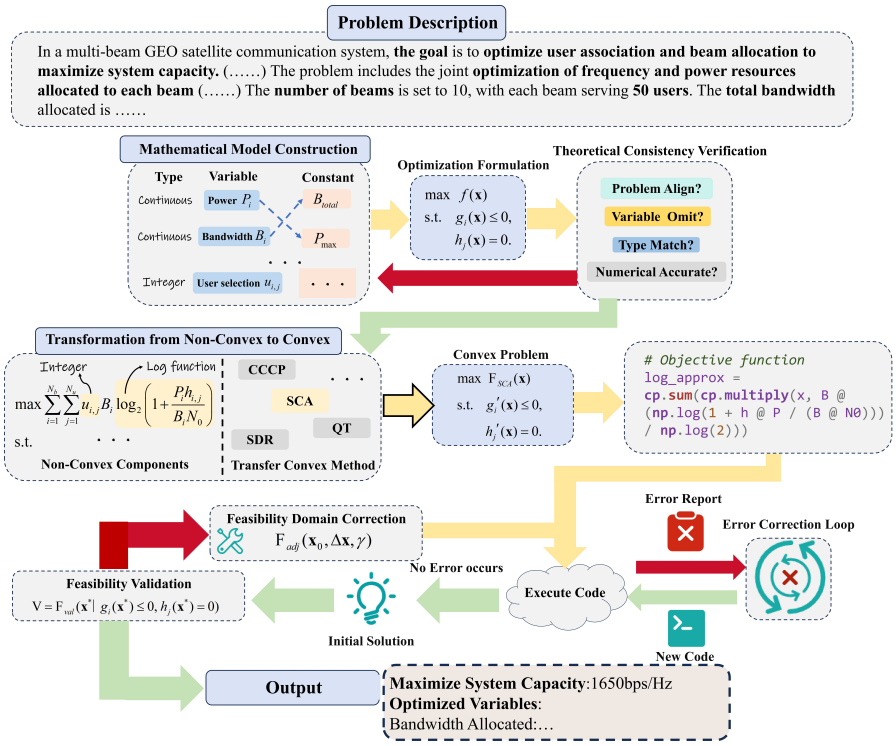

Figure 1: Overview of the NC2C framework, which leverages LLMs to automatically transform generic non-convex optimization problems into convex forms.

jectives, allowing LLMs to avoid redundant auxiliary constraints and variables and thus accelerate solving (Ahmaditeshnizi et al., 2024). However, these solvers primarily handle convex problems, so LLMs must still transform non-convex problems on their own.

To address the existing gap in applying LLMs to automated non-convex to convex transformation, we introduce NC2C, an LLM-based framework that leverages advances in mathematical reasoning, prompt engineering, and few-shot learning. The main contributions of this paper are:

- We propose the NC2C framework, an LLM-based approach to address generic non-convex optimization problems. The framework automatically detects non-convex components and efficiently transforms them into convex problems, significantly reducing reliance on expert knowledge and enabling automated non-convex problem solving across diverse domains.
- We introduce Error Correction Loops (ECL) and Feasibility Domain Correction (FDC) as refinement stages within the NC2C framework. These stages substantially improve the reliability and execution rate of NC2C's solutions on complex non-convex instances.
- Experimental results show that NC2C achieves up to a 94.5% success rate and a 96.7% execution rate on NL4Opt with GPT-5.1, significantly outperforming state-of-the-art baseline methods across four benchmarks. These results highlight NC2C's effectiveness and robustness in automated non-convex optimization through convexification.

2 Related Work

2.1 LLMs for Optimization Problems.

Recent studies have shown that large language models (LLMs) have significant potential in addressing complex optimization problems, both convex and non-convex. LLMs have demonstrated success in solving classical optimization problems like linear regression and the traveling salesman problem using iterative methods (Yang et al., 2024). Moreover, LLMs can leverage existing solvers

like `Gurobi` and `CVXPY` to handle convex optimization tasks, accelerating the solving process (Ahmaditeshnizi et al., 2024). LLMOPT (Jiang et al., 2025) and OptimAI (Thind et al., 2025) focus on end-to-end frameworks that define and solve problems from scratch using autonomous agents. To enhance reliability, OR-LLM-Agent (Zhang & Luo, 2025) leverages reasoning-heavy models to automate complex operations research modeling. OptiBench (Yang et al., 2025b) introduces standardized benchmarks to systematically measure and improve LLM modeling accuracy. However, these solvers primarily focus on convex problems, and direct solution of non-convex problems remains a challenge.

2.2 Tool Learning and Reasoning for LLMs.

The integration of LLMs with external tools represents a paradigm shift from closed-loop generation to open-environment interaction. Beyond basic function calling (Ouyang et al., 2022), frameworks like ReAct (Yao et al., 2023) and ToolChain* (Zhuang et al., 2024) formalize the interleaving of reasoning and acting. To enhance long-horizon consistency, Tool-Planner (Qin et al., 2024; Liu et al., 2025) introduces explicit global planning, transitioning from reactive execution to proactive strategy optimization. To overcome the limitations of linear decision-making, LATS (Zhou et al., 2024) incorporates tree-search heuristics to traverse the strategy space, while Reflexion enables iterative self-correction through feedback loops, establishing a closed-loop refinement mechanism for tool invocation (Shinn et al., 2023). Benchmarks (Ye et al., 2025; Patil et al., 2024) have consequently evolved from evaluating API selection accuracy to assessing higher-order dimensions, with reinforcement learning (Qian et al., 2025) further aligning dynamic reasoning policies.

3 Methodology

As illustrated in Figure 1, we propose the NC2C framework, which leverages LLMs to automatically transform generic non-convex optimization problems into convex forms.

3.1 Problem Formulation and Mathematical Modeling

Given a natural language problem description $\mathcal{D}_p$ provided by the user, NC2C first constructs a formal mathematical representation. The framework employs Named Entity Recognition methods (Wang et al., 2023) to extract optimization-related entities from $\mathcal{D}_p$, identifying decision variables, parameters, and their relationships.

Let $\mathcal{V} = \{v_1, v_2, \ldots, v_n\}$ denote the set of extracted variables, where each variable v_i is associated with a type $\tau_i \in \{\text{continuous}, \text{integer}, \text{binary}\}$. The variable extraction process is formalized as:

$$\mathcal{V} = \text{ExtractVariables}(\mathcal{D}_p) \tag{1}$$

$$\tau_i = \text{InferType}(v_i, \mathcal{D}_p). \tag{2}$$

The objective function is identified by analyzing optimization goals expressed in $\mathcal{D}_p$. The framework constructs the objective function $f : \mathbb{R}^n \to \mathbb{R}$ by mapping extracted variables to their mathematical relationships. Constraint extraction identifies inequality constraints $g_i(\mathbf{x}) \leq 0$ and equality constraints $h_j(\mathbf{x}) = 0$. The complete optimization problem $\mathcal{P}$ is formulated as:

$$\begin{aligned} \underset{\mathbf{x}\in\mathcal{X}}{\text{optimize}} \quad & f(\mathbf{x}) \\ \text{subject to} \quad & g_i(\mathbf{x}) \leq 0, \quad i \in \mathcal{I} = \{1, 2, \ldots, m\}, \\ & h_j(\mathbf{x}) = 0, \quad j \in \mathcal{J} = \{1, 2, \ldots, p\}, \\ & \mathbf{x} \in \mathcal{X}, \end{aligned} \tag{3}$$

where $\mathbf{x} = [x_1, x_2, \ldots, x_n]^T$ is the decision vector, $\mathcal{X}$ denotes the variable domain constraints, and m and p represent the number of inequality and equality constraints.

3.2 Non-Convex Detection and Convexification

NC2C systematically identifies non-convex components in $\mathcal{P}$ and applies appropriate convexification strategies. The detection process analyzes the structure of $f(\mathbf{x})$, $g_i(\mathbf{x})$, and $h_j(\mathbf{x})$ to identify

problematic terms such as bilinear products, fractional expressions, logarithmic functions in maximization contexts, or integer constraints.

Let $\mathcal{N} = \{n_1, n_2, \ldots, n_k\}$ denote the set of identified non-convex components. For each component $n_l \in \mathcal{N}$, the framework selects a convexification strategy σ_l from a strategy set $\Sigma = \{\text{SCA}, \text{SDR}, \text{Lagrangian}, \text{Substitution}, \ldots\}$:

$$\sigma_l = \text{SelectStrategy}(n_l, \mathcal{P}), \quad \forall n_l \in \mathcal{N}. \tag{4}$$

For non-convex objective functions, NC2C commonly employs Successive Convex Approximation (SCA). At a reference point $\mathbf{x}^{(0)} \in \mathcal{X}$, each non-convex term is replaced by a convex surrogate that matches its value and gradient at $\mathbf{x}^{(0)}$ and bounds it on $\mathcal{X}$. For a concave term, this surrogate is the affine Taylor model

$$\tilde{f}(\mathbf{x}; \mathbf{x}^{(0)}) = f(\mathbf{x}^{(0)}) + \nabla f(\mathbf{x}^{(0)})^T (\mathbf{x} - \mathbf{x}^{(0)}), \tag{5}$$

which is affine, hence convex, and upper-bounds the concave term (a valid majorizer). Convex terms are kept unchanged, so the surrogate is tight at the reference point and the resulting sub-problem is convex. Non-convex inequality constraints $g_i(\mathbf{x}) \leq 0$ are majorized in the same way, yielding the convexified problem $\mathcal{P}_c$ with constraints $\tilde{g}_i, \tilde{h}_j$ and domain $\mathcal{X}_c$; the sub-problem is re-solved from updated reference points until the objective stabilizes.

3.3 Code Generation and Execution

Based on the convexified problem $\mathcal{P}_c$, NC2C generates executable Python code $C(\mathcal{P}_c)$ using an appropriate solver $\mathcal{S} \in \{\text{CVXPY}, \text{Gurobi}, \text{SCIPY}\}$: `CVXPY` (Diamond & Boyd, 2016) for standard convex optimization, `Gurobi` for large-scale linear and mixed-integer problems, and `SCIPY` for general nonlinear convex problems.

The code generation is formalized as $C(\mathcal{P}_c) = \text{GenerateCode}(\mathcal{P}_c, \mathcal{S})$. Upon execution, the code either succeeds and returns a solution $\mathbf{x}^*$, or fails and generates an error report $\mathcal{E}$. The execution status is captured by the indicator function $\mathbb{I}_e(C(\mathcal{P}_c)) \in \{0, 1\}$, where 1 indicates successful execution. When $\mathbb{I}_e = 1$, the solver returns $\mathbf{x}^* = \arg\min_{\mathbf{x} \in \mathcal{X}_c} f_c(\mathbf{x})$ subject to the convexified constraints.

3.4 Error Correction Loop (ECL)

When code execution fails ($\mathbb{I}_e = 0$), NC2C activates the Error Correction Loop (ECL) to iteratively refine the generated code. In the k-th iteration ($k = 1, 2, \ldots, K$), ECL analyzes the error report $\mathcal{E}^{(k)}$ to identify failure causes, such as syntax errors, dimension mismatches, or solver incompatibilities:

$$C^{(k+1)}(\mathcal{P}_c) = \text{CorrectCode}(C^{(k)}(\mathcal{P}_c), \mathcal{E}^{(k)}, \mathcal{S}, \mathcal{P}_c), \tag{6}$$

where $C^{(k)}$ denotes the code at iteration k and $\text{CorrectCode}(\cdot)$ is an LLM-based correction function. The ECL process terminates when execution succeeds or the maximum iteration count K is reached.

3.5 Solution Validation

After obtaining a solution $\mathbf{x}^*$ from $\mathcal{P}_c$, NC2C validates its feasibility with respect to the original problem $\mathcal{P}$, because convexification may relax constraints. The validation checks that $\mathbf{x}^*$ satisfies every original inequality $g_i(\mathbf{x}^*) \leq 0$ and equality $h_j(\mathbf{x}^*) = 0$. Additionally, NC2C performs theoretical consistency validation during the mathematical modeling stage to ensure the constructed problem $\mathcal{P}$ accurately reflects the original description $\mathcal{D}_p$ along four criteria: formula-description alignment, completeness of variables and constraints, type correctness, and numerical accuracy.

3.6 Feasibility Domain Correction (FDC)

When the solution $\mathbf{x}^*$ fails feasibility validation ($\mathbb{I}_f(\mathbf{x}^*) = 0$), NC2C activates the Feasibility Domain Correction (FDC) mechanism to iteratively refine the solution over a maximum of L iterations.

Stage 1: Initial Value Adjustment (iterations $l = 1, 2, \ldots, \lfloor L/2 \rfloor$). FDC adjusts the initial point $\mathbf{x}_0$ using a correction function that gradually moves toward the feasible region: $\mathbf{x}_0^{(l+1)} = \mathbf{x}_0^{(l)} +$

Algorithm 1: NC2C Optimization Process

Input: problem description $\mathcal{D}_p$
Output: success flag, execution flag, solution $\mathbf{x}^*$
Initialize success flag $\leftarrow 0$, execution flag $\leftarrow 0$
$\mathcal{P} \leftarrow$ GenerateMathFormulation$(\mathcal{D}_p)$
$\mathcal{P}_c \leftarrow$ ConvertToConvex$(\mathcal{P})$
$C \leftarrow$ GenerateCode$(\mathcal{P}_c)$
for $k = 1$ to K **do**
  $\mathbf{x}^*, \mathbb{I}_e \leftarrow$ ExecuteCode(C)
  if $\mathbb{I}_e = 1$ **then**
    execution flag $\leftarrow 1$; **break**
  else
    $C \leftarrow$ CorrectCode$(C, \mathcal{E}^{(k)}, \mathcal{P}_c)$ {ECL}
  end if
end for
if execution flag $= 1$ **then**
  $\mathbb{I}_f \leftarrow$ CheckFeasibility$(\mathcal{P}, \mathbf{x}^*)$
  if $\mathbb{I}_f = 1$ **then**
    success flag $\leftarrow 1$
  else
    for $l = 1$ to L **do**
      $C, \mathbf{x}^* \leftarrow$ RefineSolution$(\mathcal{P}, \mathcal{P}_c, C, l)$ {FDC}
      if CheckFeasibility$(\mathcal{P}, \mathbf{x}^*) = 1$ **then**
        success flag $\leftarrow 1$; **break**
      end if
    end for
  end if
end if
return success flag, execution flag, $\mathbf{x}^*$

$\alpha^{(l)} \cdot \Delta\mathbf{x}^{(l)}$, where $\alpha^{(l)} \in (0, 1]$ is the step size and $\Delta\mathbf{x}^{(l)}$ is the correction direction computed from constraint violations. The updated initial point is used to re-solve $\mathcal{P}_c$, yielding a new candidate $\mathbf{x}^{*(l+1)}$.

Stage 2: Problem Re-convexification (iterations $l = \lfloor L/2 \rfloor + 1, \ldots, L$). If Stage 1 fails to find a feasible solution, FDC reanalyzes the original problem $\mathcal{P}$ and applies alternative convexification strategies, generating a new convexified problem $\mathcal{P}_c^{(l)} = \text{ReConvexify}(\mathcal{P}, \{\sigma_l'\}_{l=1}^{k'}, \mathbf{x}^{*(l-1)})$, where $\{\sigma_l'\}$ represents alternative strategies.

The FDC process terminates when either a feasible solution is found or the maximum iteration count is reached, returning either the first feasible $\mathbf{x}^{*(l)}$ or the final iterate $\mathbf{x}^{*(L)}$.

Algorithm 1 summarizes the end-to-end NC2C pipeline: mathematical modeling and convexification produce solver-ready code, ECL repairs execution failures, and FDC iteratively restores feasibility when convexification relaxes the original problem.

3.7 Prompt Design

Each stage in Algorithm 1 is driven by stage-specific prompts that interface natural-language problem descriptions with structured model outputs. NC2C maintains separate prompt templates for modeling, convexification, code generation, execution diagnosis, and feasibility checking.

Mathematical modeling. The modeling prompt instructs the LLM to construct a complete formulation—objective, constraints, auxiliary variable definitions, and parameter values—from $\mathcal{D}_p$. It also requires an explicit optimization flag (`[Optimization Flag: 1]` for maximization, `0` for minimization) so downstream stages know the problem direction. A consistency-validation prompt is invoked iteratively until the generated formula aligns with the original description.

Convexification. Convex conversion uses a few-shot system prompt with worked examples of non-convex-to-convex transformations (change of variables, logarithmic reformulation, binary relaxation) together with a user query that supplies the current mathematical formulation. The model must identify every non-convex component, justify the chosen strategy from Σ, derive the convexified formulas step by step, and return the final problem $\mathcal{P}_c$ including any supplementary variables.

Code generation and execution. A code-generation prompt maps $\mathcal{P}_c$ to executable Python using an appropriate solver library, with few-shot examples demonstrating objective definition, constraint setup, and solver invocation. When execution fails, ECL feeds the error traceback back to the LLM together with the current code and formulation, requesting a minimally edited fix. When execution succeeds but feasibility checking fails, FDC prompts guide initial-value adjustment or alternative re-convexification.

Feasibility validation. The candidate solution is substituted back into the original formulation $\mathcal{P}$ and each original constraint is evaluated; a solution counts as feasible only if all equality and inequality constraints are satisfied within numerical tolerance. This closes the loop between the relaxed convex solution and the original non-convex problem.

4 Experimental Setup

4.1 Dataset

We evaluate NC2C on four diverse benchmarks that cover different aspects of optimization problem solving. For each problem we conduct $N = 10$ independent evaluations using both GPT-5.1 and Qwen3-235B-A22B under identical conditions.

NL4Opt (Ramamonjison et al., 2023) is drawn from the NL4Opt competition and is designed to evaluate the ability to automatically convert natural-language descriptions of optimization problems into solver-ready code. The original dataset contains 289 instances, primarily Linear Programming (LP) and Mixed Integer Linear Programming (MILP) problems. After filtering for instances that involve non-convex components requiring convexification, we retain 214 well-formed instances for evaluation.

NLP4LP (Ahmaditeshnizi et al., 2024) is provided by the OptiMUS benchmark and contains 344 linear and integer programming problems with verbose natural-language descriptions and multi-dimensional parameters. The dataset is particularly challenging due to its lengthy descriptions and complex constraint structures. After filtering for non-convex instances that require transformation, we retain 178 instances for evaluation.

ComplexOR (Xiao et al., 2024) initially contains 37 challenging operations-research problems drawn from production scenarios, including combinatorial optimization tasks with tightly coupled variables. After validation we use 18 instances that involve non-convex components requiring convexification. This dataset stresses the model's reasoning and problem-solving capabilities in complex contexts.

WireOpt (Peng et al., 2025) is curated by the authors specifically for non-convex to convex transformation tasks in wireless communications and signal processing. We extract 50 non-convex resource-allocation problems from authoritative sources including optimization textbooks (Hou et al., 2014), research papers (Shen & Yu, 2018), and standard solver libraries (The MathWorks, Inc., 2022), then apply LLM-based data augmentation to generate 50 additional structurally similar instances, yielding 100 evaluation problems in total. WireOpt complements the existing benchmarks by stressing formulations encountered in radio resource management, beamforming, and convex relaxations in communications.

The four datasets jointly cover linear programming, mixed-integer programming, combinatorial optimization, and communication-domain non-convex problems, providing a broad testbed for NC2C.

4.2 Baselines

We compare NC2C against state-of-the-art baselines that represent different optimization approaches with LLMs.

Reflexion (Shinn et al., 2023) leverages the LLM's self-reflection capability. After each failed attempt the model critiques its own output, identifies the cause of failure, and regenerates a corrected formulation. Reflexion-style self-reflection has been reported to raise accuracy on NL4Opt substantially; we adapt its reflection prompt to check the validity of convexification strategies in addition to code-level debugging.

Chain-of-Experts (Xiao et al., 2024) decomposes a complex reasoning task across a chain of specialized expert models. Each expert focuses on one aspect (mathematical modeling, convexification, code generation), enabling collaborative decomposition. Prior work reports 68.8% success rate on NL4Opt and 40.5% on ComplexOR; we instantiate experts specialized in mathematical modeling, convexification, and code debugging, mirroring NC2C's pipeline but without iterative error correction or feasibility verification.

OptiMUS (Ahmaditeshnizi et al., 2024) is an LLM-based agent designed to formulate and solve Mixed Integer Linear Programming problems from natural-language descriptions. It can develop mathematical models, write and debug solver code, and evaluate generated solutions. OptiMUS improves over prior methods by more than 12% on easy datasets and more than 8% on challenging datasets including NLP4LP; we extend its prompting to handle non-convex problems by adding a high-level convexification step so that it competes on the same task as NC2C.

Vanilla applies the underlying base LLM—GPT-5.1 (OpenAI, 2025) or Qwen3-235B-A22B (Yang et al., 2025a)—directly to the optimization problem with a single prompt, without any external framework or iterative refinement. This baseline measures the raw capability of the underlying LLM on non-convex to convex transformation and isolates the contribution of NC2C's pipeline.

4.3 Evaluation Metrics

We adopt two standard metrics (Ahmaditeshnizi et al., 2024). For each problem $\mathcal{P} \in \mathcal{D}$, we conduct $N = 10$ independent evaluations and average the results across the dataset:

$$\text{SR} = \frac{1}{|\mathcal{D}|} \sum_{\mathcal{P} \in \mathcal{D}} \left(\frac{1}{N} \sum_{i=1}^{N} \mathcal{F}_{\mathcal{P}} \right), \tag{7}$$

$$\text{ER} = \frac{1}{|\mathcal{D}|} \sum_{\mathcal{P} \in \mathcal{D}} \left(\frac{1}{N} \sum_{i=1}^{N} \mathcal{E}_{\mathcal{P}} \right), \tag{8}$$

where $\mathcal{F}_{\mathcal{P}} = 1$ if the returned solution satisfies all original constraints of $\mathcal{P}$ within numerical tolerance and $\mathcal{E}_{\mathcal{P}} = 1$ if the generated code executes successfully. Both metrics are reported in percent.

5 Results

5.1 Main Experiments

Table 1 summarizes the comprehensive results across the four datasets. NC2C consistently outperforms all baseline methods on both GPT-5.1 and Qwen3-235B-A22B. On the NL4Opt dataset with GPT-5.1, NC2C achieves a 94.5% SR and 96.7% ER, surpassing OptiMUS (90.2%/94.0%), Chain-of-Experts (88.3%/92.5%), Reflexion (87.5%/92.8%), and Vanilla (85.0%/90.2%). On Qwen3-235B-A22B, NC2C still achieves 91.2% SR / 94.4% ER on NL4Opt, outperforming all baselines. The performance gap is most pronounced on the more challenging ComplexOR dataset: NC2C with GPT-5.1 reaches 72.2% SR versus OptiMUS at 63.9%, and on Qwen3-235B-A22B reaches 66.7% versus OptiMUS at 55.6%, demonstrating NC2C's effectiveness in complex non-convex scenarios. On the verbose NLP4LP dataset, NC2C with GPT-5.1 obtains a 90.5% SR versus OptiMUS at 85.4%, and on WireOpt NC2C achieves 87.0% / 93.0% (SR/ER), evidencing generalizability across model architectures and problem domains.

5.2 Ablation Studies

We assess the contribution of NC2C's key components through ablation: (1) the full **NC2C** framework; (2) **o-convex**, in which convexification is removed and the model directly handles non-convex

Table 1: Main experimental results on NL4Opt, NLP4LP, ComplexOR, and WireOpt. We report Success Rate (SR, %) and Execution Rate (ER, %) for each dataset; results are averaged over 10 independent runs per problem. The best result in each model block per column is bolded.

	NL4Opt		NLP4LP		ComplexOR		WireOpt	
Method	SR	ER	SR	ER	SR	ER	SR	ER
GPT-5.1								
Vanilla	85.0	90.2	78.5	85.3	52.8	60.0	73.0	80.0
Chain-of-Experts	88.3	92.5	82.0	88.2	58.3	66.7	77.0	84.0
OptiMUS	90.2	94.0	85.4	90.5	63.9	72.2	80.0	87.0
Reflexion	87.5	92.8	81.2	88.0	55.6	64.4	75.0	83.0
NC2C (ours)	**94.5**	**96.7**	**90.5**	**94.2**	**72.2**	**80.6**	**87.0**	**93.0**
Qwen3-235B-A22B								
Vanilla	78.5	85.2	72.3	80.1	45.0	52.8	68.0	75.0
Chain-of-Experts	82.3	88.5	76.5	83.2	50.0	58.3	72.0	79.0
OptiMUS	85.0	90.5	80.2	87.0	55.6	63.9	75.0	82.0
Reflexion	80.8	88.0	74.5	82.5	48.3	56.1	70.0	78.0
NC2C (ours)	**91.2**	**94.4**	**86.5**	**91.0**	**66.7**	**75.0**	**82.0**	**89.0**

problems; (3) **o-ECL**, in which no error correction is performed; and (4) **o-FDC**, in which no re-solving is performed for infeasible solutions.

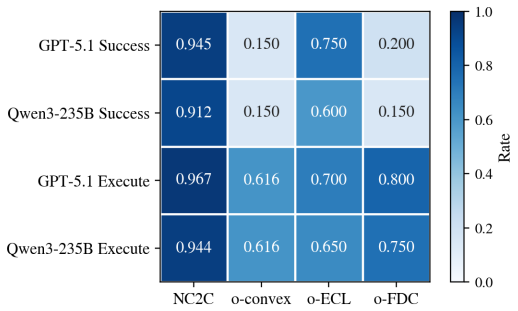

Figure 2: Impact of omitting key components of the NC2C framework on success and execution rates across GPT-5.1 and Qwen3-235B-A22B.

As shown in Figure 2, *o-convex significantly reduces both metrics, highlighting the critical importance of convexification*. GPT-5.1's success rate drops from 0.945 to 0.15 (an 84.1% decline) and execution rate from 0.967 to 0.616; with Qwen3-235B-A22B, the SR drops from 0.912 to 0.15 (83.6% decline). This confirms that convexification is essential to convert non-convex problems into solvable forms.

o-ECL primarily harms execution rate, which subsequently lowers success rate. For GPT-5.1, ER drops from 0.967 to 0.70 (27.6% decrease) and SR from 0.945 to 0.75 (20.6% reduction); for Qwen3-235B-A22B, ER drops from 0.944 to 0.65 (31.1%) and SR from 0.912 to 0.60 (34.2%).

o-FDC primarily decreases success rate, emphasizing its role in ensuring solution feasibility. Without FDC, GPT-5.1's SR drops to 0.20 (78.8% reduction) while Qwen3-235B-A22B's falls to 0.15 (83.6% decrease). Although GPT-5.1's ER remains at 0.80, the absence of FDC leaves many solutions outside feasible regions.

5.3 Running Time Analysis

As shown in Figure 3, NC2C delivers superior performance with competitive efficiency. With GPT-5.1, NC2C's average running time is 22.5 s per problem, faster than OptiMUS (28.5 s), Reflexion

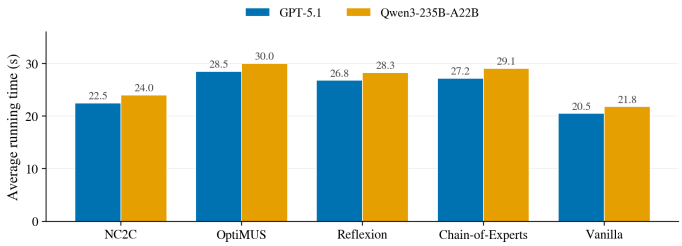

Figure 3: Average running time of NC2C and baseline methods on GPT-5.1 and Qwen3-235B-A22B.

(26.8 s), and Chain-of-Experts (27.2 s), and only slightly slower than Vanilla (20.5 s). On Qwen3-235B-A22B, NC2C is similarly the most efficient non-trivial baseline. The marginal overhead relative to Vanilla is justified by substantially higher SR and ER, demonstrating that NC2C's integrated convexification, error correction, and feasibility verification mechanisms are both effective and efficiently implemented. A component-level breakdown of this time is provided in Appendix D, and a detailed end-to-end case study on a resource-allocation instance, covering mathematical modeling, non-convexity detection, convexification, code repair, and feasibility validation, is provided in Appendix H.

5.4 Component Time Breakdown

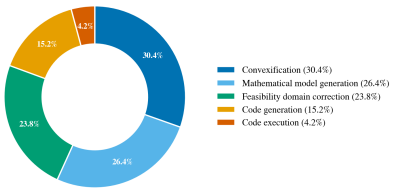

Figure 4: Internal time distribution of NC2C across its main processing stages, computed on GPT-5.1 with NL4Opt.

To identify where NC2C spends its budget, we profile the per-stage wall-clock time on NL4Opt with GPT-5.1 (Figure 4). Convex-problem transformation consumes the largest share at roughly 30.4%, driven by complex derivations over function structure and constraints. Mathematical model generation accounts for about 26.4% and feasibility verification adds another 23.8%, reflecting the multi-pronged check that occasionally triggers re-solve attempts under FDC. The remaining 19.4% is split among code generation, code execution, and inter-stage I/O. This distribution is consistent with the ablation results: removing convexification (*o-convex*) collapses SR from 94.5% to 15%, confirming that the time spent on transformation is indispensable despite its cost. Likewise, the overhead of modeling and feasibility checking buys reliability gains that far exceed their time cost.

6 Conclusion

We have presented NC2C, an LLM-based framework that automatically transforms generic non-convex optimization problems into convex forms. Across two leading LLMs and four benchmarks, NC2C consistently outperforms state-of-the-art baselines, with the integrated convexification, error correction, and feasibility verification stages each contributing measurably to its gains. As language models that incorporate real-time feedback become more capable, NC2C offers a practical path to deploying convex solvers in previously intractable domains.

AI USE STATEMENT

In this work, we used generative AI tools for research tasks with *required* disclosure, including helping to develop and refine the conceptual framework, providing feedback on the research methodology and experimental design, implementing methods, and interpreting results. We also used generative AI tools for tasks with *recommended* disclosure, including creating or editing software code, creating or modifying scientific figures, drafting and editing parts of the manuscript, brainstorming, identifying relevant literature, and summarizing existing literature. We did not use generative AI tools for the other tasks with required disclosure (for example, generating synthetic datasets, translating text, or cleaning and reformatting datasets), and those tasks are not applicable to this work. We have reviewed all AI-assisted work: LLM-generated code was executed and tested for correctness by the authors, AI-assisted literature summaries and references were verified against the original sources, and all claims, results, and artifacts were checked by the authors. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

We specify the complete NC2C pipeline, including the error-correction loop (ECL) and the two-stage feasibility-domain correction (FDC) schedule, in Algorithm 1. The Experimental Setup section describes the four evaluation datasets (including the provenance and construction of the WireOpt benchmark), the baseline implementations, the success-rate and execution-rate metrics, and the protocol of $N = 10$ independent evaluations per problem. We report the maximum iteration counts for ECL and FDC in Appendix E.

REFERENCES

Ali Ahmaditeshnizi, Wenzhi Gao, and Madeleine Udell. OptiMUS: Scalable optimization modeling with (MI)LP solvers and large language models. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 577–596. PMLR, 2024. URL https://proceedings.mlr.press/v235/ahmaditeshnizi24a.html.

Jiangjie Chen, Wenxiang Chen, Jiacheng Du, Jinyi Hu, Zhicheng Jiang, Allan Jie, Xiaoran Jin, Xing Jin, Chenggang Li, Wenlei Shi, et al. Seed-Prover 1.5: Mastering Undergraduate-Level Theorem Proving via Learning from Experience, 2025a.

Luoxin Chen, Jinming Gu, Liankai Huang, Wenhao Huang, Zhicheng Jiang, Allan Jie, Xiaoran Jin, Xing Jin, Chenggang Li, Kaijing Ma, et al. Seed-Prover: Deep and Broad Reasoning for Automated Theorem Proving, 2025b.

Steven Diamond and Stephen Boyd. CVXPY: A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research*, 17(83):1–5, 2016.

Simone Garatti and Marco C. Campi. Non-Convex Scenario Optimization. *Mathematical Programming*, 209(1):557–608, 2025.

Y. Thomas Hou, Yi Shi, and Hanif D. Sherali. *Convex Programming and Applications*. Cambridge University Press, 2014.

Caigao Jiang, Xiang Shu, Hong Qian, Xingyu Lu, Jun Zhou, Aimin Zhou, and Yang Yu. LLMOPT: Learning to define and solve general optimization problems from scratch. In *Proceedings of the 13th International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=9OMvtboTJg.

Chengchang Liu, Chaowen Guan, Jianhao He, and John Lui. Quantum Algorithms for Non-Smooth Non-Convex Optimization. In *Advances in Neural Information Processing Systems*, volume 37, pp. 35288–35312, 2024.

Yanming Liu, Xinyue Peng, Jiannan Cao, Shi Bo, Yuwei Zhang, Xuhong Zhang, Sheng Cheng, Xun Wang, Jianwei Yin, and Tianyu Du. Tool-Planner: Task planning with clusters across multiple

tools. In *Proceedings of the 13th International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=dRz3cizftU.

V. S. Mikhalevich, A. M. Gupal, and V. I. Norkin. Methods of Nonconvex Optimization, 2024.

OpenAI. GPT-5.1: A smarter, more conversational ChatGPT. https://openai.com/index/gpt-5-1/, 2025. Accessed: 2025-11-12.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training Language Models to Follow Instructions with Human Feedback, 2022.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive APIs. *Advances in Neural Information Processing Systems*, 37:126544–126565, 2024.

Xinyue Peng, Yanming Liu, Yihan Cang, Chaoqun Cao, and Ming Chen. LLM-OptiRA: LLM-driven optimization of resource allocation for non-convex problems in wireless communications, 2025. URL https://arxiv.org/html/2505.02091v1.

Cheng Qian, Emre Can Acikgoz, Qi He, Hongru Wang, Xiusi Chen, Dilek Hakkani-Tür, Gokhan Tur, and Heng Ji. ToolRL: Reward is All Tool Learning Needs. In *Proceedings of the 39th Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=eOLdGbXT6t.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. ToolLLM: Facilitating Large Language Models to Master 16000+ Real-World APIs. In *Proceedings of the 12th International Conference on Learning Representations*, 2024.

Rindranirina Ramamonjison, Timothy Yu, Raymond Li, Haley Li, Giuseppe Carenini, Bissan Ghaddar, Shiqi He, Mahdi Mostajabdaveh, Amin Banitalebi-Dehkordi, Zirui Zhou, et al. NL4Opt competition: Formulating optimization problems based on their natural language descriptions. In *NeurIPS 2022 Competition Track*, pp. 189–203. PMLR, 2023.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–68551, 2023.

Kaiming Shen and Wei Yu. Fractional programming for communication systems—part I: Power control and beamforming. *IEEE Transactions on Signal Processing*, 66(10):2616–2630, 2018.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652, 2023.

Xiaomeng Sui, Zehao He, Daping Chu, and Liangcai Cao. Non-Convex Optimization for Inverse Problem Solving in Computer-Generated Holography, 2024.

The MathWorks, Inc. MATLAB version 9.13.0 (r2022b). https://www.mathworks.com, 2022.

Raghav Thind, Youran Sun, Ling Liang, and Haizhao Yang. OptimAI: Optimization from natural language using LLM-powered AI agents, 2025.

Shuhe Wang, Xiaofei Sun, Xiaoya Li, Rongbin Ouyang, Fei Wu, Tianwei Zhang, Jiwei Li, and Guoyin Wang. GPT-NER: Named entity recognition via large language models, 2023.

Ziyang Xiao, Dongxiang Zhang, Yangjun Wu, Lilin Xu, Yuan Jessica Wang, Xiongwei Han, Xiaojin Fu, Tao Zhong, Jia Zeng, Mingli Song, et al. Chain-of-Experts: When LLMs meet complex operations research problems. In *Proceedings of the 12th International Conference on Learning Representations*, 2024.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report, 2025a.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large Language Models as Optimizers. In *Proceedings of the 12th International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=Bb4VGOWELI.

Zhicheng Yang, Yiwei Wang, Yinya Huang, Zhijiang Guo, Wei Shi, Xiongwei Han, Liang Feng, Linqi Song, Xiaodan Liang, and Jing Tang. OptiBench meets ReSocratic: Measure and improve LLMs for optimization modeling. In *Proceedings of the 13th International Conference on Learning Representations*, 2025b.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *Proceedings of the 11th International Conference on Learning Representations*, 2023.

Junjie Ye, Guanyu Li, Songyang Gao, Caishuang Huang, Yilong Wu, Sixian Li, Xiaoran Fan, Shihan Dou, Tao Ji, Qi Zhang, et al. ToolEyes: Fine-grained evaluation for tool learning capabilities of large language models in real-world scenarios. In *Proceedings of the 31st International Conference on Computational Linguistics*, pp. 156–187, 2025.

Bowen Zhang and Pengcheng Luo. OR-LLM-Agent: Automating modeling and solving of operations research optimization problem with reasoning large language model, 2025.

Qihuang Zhong, Kang Wang, Ziyang Xu, Juhua Liu, Liang Ding, Bo Du, and Dacheng Tao. Achieving> 97% on GSM8K: Deeply Understanding the Problems Makes LLMs Perfect Reasoners. In *Proceedings of the 12th International Conference on Learning Representations*, 2024.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language Agent Tree Search Unifies Reasoning, Acting, and Planning in Language Models. In *Proceedings of the 41st International Conference on Machine Learning*, pp. 62138–62160, 2024.

Yuchen Zhuang, Xiang Chen, Tong Yu, Saayan Mitra, Victor Bursztyn, Ryan A. Rossi, Somdeb Sarkhel, and Chao Zhang. ToolChain*: Efficient action space navigation in large language models with A* search. In *Proceedings of the 12th International Conference on Learning Representations*, 2024.

A Broader Impact And Limitations

Broader Impact. The proposed NC2C method significantly improves success and execution rates in generic non-convex optimization by introducing automated non-convex to convex transformations and multi-stage solution strategies. We believe that by integrating more optimization techniques, such as reinforcement learning and online learning, large-scale models can further enhance their solving capabilities in complex scenarios. In particular in dynamic environments, models that combine real-time feedback could achieve more efficient optimization. Furthermore, the current model can be further integrated with information retrieval and incremental learning techniques to handle more complex tasks with a large amount of noise or uncertainty in practical applications. We believe that the fusion of these technologies will provide high-quality solutions for the broader optimization field, advancing the development and application of automated convexification techniques.

B Baselines

Reflexion (Shinn et al., 2023) leverages LLMs' self-reflection capabilities to improve reasoning performance. The method enables the model to critique its own outputs, identify errors, and regenerate corrected formulations through iterative refinement. Reflexion has demonstrated significant improvements in optimization tasks, achieving accuracy improvements from 78.8% to 92.3% on the NL4Opt benchmark. We adapt Reflexion for non-convex to convex transformation by incorporating reflection steps that evaluate the correctness of convexification strategies.

Chain-of-Experts (Xiao et al., 2024) employs a multi-expert architecture where different expert models are chained together to handle complex reasoning tasks. Each expert specializes in different aspects of the problem-solving process, enabling collaborative problem decomposition and solution generation. This approach has shown effectiveness in complex operations research scenarios, achieving 68.8% success rate on NL4Opt and 40.5% on ComplexOR datasets. We configure Chain-of-Experts with experts specialized in mathematical modeling, convexification, and code generation.

OptiMUS (Ahmaditeshnizi et al., 2024) is an LLM-based agent designed to automatically formulate and solve Mixed Integer Linear Programming (MILP) problems from natural language descriptions. The framework can develop mathematical models, write and debug solver code, and evaluate generated solutions. OptiMUS has demonstrated superior performance, achieving over 12% improvement on easy datasets and over 8% on challenging datasets including NLP4LP compared to existing state-of-the-art methods. We extend OptiMUS to handle non-convex problems by incorporating convexification steps.

Vanilla represents the direct application of the underlying the base LLM models (GPT-5.1 (OpenAI, 2025) or Qwen3-235B-A22B (Yang et al., 2025a)) to solve optimization problems without additional frameworks or specialized prompting strategies. This baseline evaluates the raw capability of advanced LLMs in handling non-convex to convex transformation tasks, providing a strong comparison point for our framework.

C Dataset

NL4Opt (Ramamonjison et al., 2023) is a benchmark dataset from NL4Opt competition, designed to evaluate the ability to automatically convert natural language descriptions of optimization problems into solver-ready code. The original dataset contains 289 instances, primarily consisting of Linear Programming (LP) and Mixed Integer Linear Programming (MILP) problems. After data cleaning and validation, we use 214 instances that contain well-formed non-convex optimization problems requiring convexification.

NLP4LP (Ahmaditeshnizi et al., 2024) is provided by the OptiMUS benchmark, containing 344 linear and integer programming problems with verbose descriptions and multi-dimensional parameters. The dataset is particularly challenging due to its lengthy problem descriptions and complex constraint structures. After filtering for non-convex problems that require transformation, we retain 178 instances for evaluation.

ComplexOR (Xiao et al., 2024) initially contains 37 challenging optimization problems involving complex operations research scenarios, including combinatorial optimization tasks. This dataset focuses on evaluating LLMs' reasoning and problem-solving capabilities in complex optimization contexts. After data validation, we use 18 instances that involve non-convex components requiring convexification.

WireOpt (Peng et al., 2025) is specifically designed for non-convex to convex transformation tasks. We extract 50 non-convex optimization problems from authoritative sources including textbooks, research papers, and optimization libraries (Hou et al., 2014; Shen & Yu, 2018; The MathWorks, Inc., 2022). These problems span various domains including mathematical programming, engineering design, and scientific computing in a total of 100 instances. This dataset serves as a complementary evaluation set to assess NC2C's performance on diverse non-convex optimization challenges.

D Running time of each component within the NC2C framework

Building on the running time analysis in last subsection, we continue to analyze the time distribution and contributions of the internal components within the NC2C framework.

When the NC2C framework runs, convex problem transformation, mathematical model generation, and feasibility verification each take up more than 23% of the total time. The convex problem transformation part, accounting for about 30.4% of the time, involves in-depth analysis and conversion of complex non-convex problems into convex ones, entailing numerous complex mathematical operations and logical derivations when handling complex function structures and constraints. The mathematical model generation part, consuming around 26.4% of the time, demands precise under-

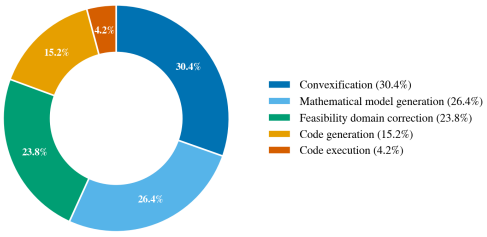

Figure 5: Pie chart of the time proportion of each part.

standing of input problems and strict transformation of natural language descriptions into mathematical models. The pursuit of accuracy in identifying entities, relationships, and semantics in natural language for constructing accurate models leads to significant time investment. The feasibility verification part, taking approximately 23.8% of the time, has a strict verification mechanism that requires multiple solution checks and possible adjustments. The detailed evaluation of various aspects of the solutions to ensure they meet constraints and practical needs results in increased time consumption.

As can also be seen from the ablation study in the main text (Figure 2), in the absence of convex problem transformation (o-convex), the success rate of GPT-5.1 plummets from 0.945 to 0.15, a significant drop of 84.1%, and the execution rate decreases from 0.967 to 0.616. This vividly demonstrates the crucial role of convex problem transformation in converting non-convex problems into solvable forms, and also explains why, despite the significant time consumption, this part is indispensable in the NC2C framework.

Similarly, the high requirements of the mathematical model generation part for understanding and transforming input problems, as well as the strict verification mechanism of the feasibility verification part, result in both of these parts consuming a relatively large amount of time within the framework. However, these components work together. Although they increase the running time, they bring about a performance improvement that far exceeds the time cost for the framework, ensuring the efficiency and reliability of NC2C in solving complex problems.

In summary, different in time-consumption, these parts collaborate. They achieve performance gains far beyond the time cost, showing the framework's effectiveness and superiority in solving complex non-convex optimization problems.

E Maximum Iteration Counts for ECL and FDC

Figure 6 reports the effect of varying the maximum iteration counts K (ECL) and L (FDC). *ECL lifts SR and ER in early iterations:* on NL4Opt with GPT-5.1, SR rises from 70% to 94.5%; with Qwen3-235B-A22B, from 60% to 91.2%. *FDC yields steady gains in SR with mild ER growth:* on ComplexOR, GPT-5.1 climbs from 45%/75% to 72.2%/80.6% and Qwen3-235B-A22B from 35%/70% to 66.7%/75%. These gains trace to FDC's two-stage schedule—initial-value adjustment before $\lfloor L/2 \rfloor$, then problem re-convexification.

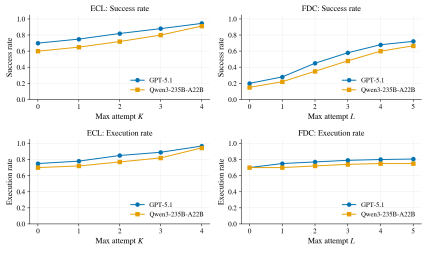

Figure 6: Success and execution rates for NC2C under different maximum iteration counts for ECL and FDC.

F PSEUDOCODE AND EXPLANATION OF NC2C OPTIMIZATION PROCESS

In the domain of optimization problem-solving, the NC2C optimization process offers an efficient approach. The following pseudocode outlines the core steps of this process, followed by a brief explanation of each significant step.

Algorithm 2: NC2C Optimization Process (detailed)

Input: problem description $\mathcal{D}_p$, max ECL iterations K, max FDC iterations L
Output: success flag, execution flag, solution $\mathbf{x}^*$
Modeling: $\mathcal{P} \leftarrow \text{GenerateMathFormulation}(\mathcal{D}_p)$
Convexification: $\mathcal{P}_c \leftarrow \text{ConvertToConvex}(\mathcal{P})$
Code generation: $C \leftarrow \text{GenerateCode}(\mathcal{P}_c)$
for $k = 1$ to K **do**
  $\mathbf{x}^*, \mathbb{I}_e \leftarrow \text{ExecuteCode}(C)$
  if $\mathbb{I}_e = 1$ **then**
    execution flag $\leftarrow 1$; **break**
  else
    $C \leftarrow \text{CorrectCode}(C, \mathcal{E}^{(k)}, \mathcal{P}_c)$
  end if
end for
if execution flag $= 1$ **then**
  $\mathbb{I}_f \leftarrow \text{CheckFeasibility}(\mathcal{P}, \mathbf{x}^*)$
  if $\mathbb{I}_f = 1$ **then**
    success flag $\leftarrow 1$
  else
    for $l = 1$ to L **do**
      $C, \mathbf{x}^* \leftarrow \text{RefineSolution}(\mathcal{P}, \mathcal{P}_c, C, l)$
      if $\text{CheckFeasibility}(\mathcal{P}, \mathbf{x}^*) = 1$ **then**
        success flag $\leftarrow 1$; **break**
      end if
    end for
  end if
end if
return success flag, execution flag, $\mathbf{x}^*$

F.1 EXPLANATION OF THE PSEUDOCODE

The entire process aims to solve the optimization problem effectively and obtain a feasible solution, with the status flags indicating the success of different stages of the process.

1. **Initialization**:
   - `success_flag` and `execute_flag` are initialized to 0. These flags will be updated later to indicate the success of the overall process and the code execution respectively.
2. **Problem Formulation**:
   - `GenerateMathFormulation(problem)`: Transforms the input natural-language optimization problem into a mathematical formula.
   - `ConvertToConvex(math_formulation)`: Converts the obtained mathematical formula into a convex optimization form, which is easier to solve.
   - `GenerateCode(convex_formulation)`: Generates executable code based on the convex optimization formula.
3. **Code Execution and Solution Checking**:
   - `ExecuteCode(code)`: Runs the generated code and returns the solution and the execution status.

- `CheckFeasibility(math_formulation, solution)`: Checks if the obtained solution is feasible according to the original mathematical formulation.

4. **Iterative Refinement**:
   - If the initial solution is not feasible, `RefineCode` is called up to `MaxAttempts` times to improve the code and try to find a feasible solution.

The effectiveness of these functions **mainly relies on well-designed prompts**, which will be detailed in the next section of the appendix.

G Details of Key Prompt Sections

In the process of solving optimization problems using LLMs, prompts play a crucial role. They serve as the interface between user problems and model outputs. Well-designed prompts can guide the model to generate expected results. This appendix will elaborate on the key prompts used in each stage from problem input to convex optimization and then to solution. By presenting and analyzing the original text of the prompts, we will reveal their design ideas and working mechanisms.

G.1 Overall Process Overview

The entire optimization problem-solving process mainly includes the following key steps:

1. **Problem Input**: Users provide descriptions of actual optimization problems, which will be used as the basic input for subsequent stages.
2. **Mathematical Formula Generation**: Utilize carefully designed prompts to guide the LLM to transform the problem described in natural language into a mathematical formula and clarify the optimization objective (maximization or minimization).
3. **Convex Optimization Conversion**: Through specific prompts, prompt the LLM to convert the generated non-convex mathematical formula into a convex form for subsequent solution.
4. **Code Generation**: Based on the convex optimization formula, use corresponding prompts to let the LLM generate executable code.
5. **Code Execution and Solution**: Execute the generated code to obtain the solution of the optimization problem.

G.2 Details of Prompts in Each Stage

G.2.1 Mathematical Formula Generation Stage

1. **Prompt File Overview**
   The main prompt file used in this stage is `math_query.txt`. Its core function is to guide the large language model (LLM) to understand and analyze the input optimization problem, generate the corresponding mathematical formula, and clarify the optimization objective. Additionally, it requests the LLM to provide formulas for other variables mentioned in the main formula and their corresponding values.
2. **Original Text and Analysis of `math_query.txt`**
   The original text content of `math_query.txt` is as follows:

`math_query.txt`

> Based on this optimization problem, construct a complete mathematical formula, including the objective function and constraints. Please also provide formulas for some of the other variables mentioned in the formula, such as channel condition h, and the values corresponding to these variables.
> Input: $input$

Detailed analysis of the original text:

- **Problem Description Placeholder**: The `$input$` in the original text is a placeholder, which will be replaced by the specific optimization problem description in actual use. This design makes the prompt highly versatile, enabling it to handle different optimization problems. For instance, if the optimization problem is about resource allocation in a network system, the `$input$` will be replaced with the detailed description of this resource-allocation problem, including parameters like available resources, entity demands, etc.
- **Comprehensive Formula Requirement**: The prompt asks the LLM to construct a complete mathematical formula, covering the objective function and constraints. This ensures that the generated mathematical representation fully captures the essence of the optimization problem. Moreover, it specifically requests formulas for other relevant variables (e.g., channel condition h) and their corresponding values. This detailed requirement helps to build a more accurate and practical mathematical model.
- **Optimization Goal Specification in Code**: Although the original prompt text in `math_query.txt` does not explicitly mention the optimization goal specification format, in the Python code `generate_math`, an additional instruction is added to the user prompt. The line `user_prompt += "Please specify whether to maximize or minimize, using the format '[Optimization Flag: 1]' for maximize and '[Optimization Flag: 0]' for minimize."` forces the LLM to clearly indicate the optimization objective in the generated mathematical formula. This clear requirement helps the model output results that meet the specifications and facilitates subsequent processing and analysis.

3. **Code-based Process Explanation**
The `generate_math` function implements the process of generating a mathematical formula based on the optimization problem. Here is a step-by-step explanation:
   - **Initialization**: The function initializes an empty string `math` to store the generated mathematical formula and a variable `optimization_flag` to record the optimization objective.
   - **Prompt Preparation**: It reads the content of `math_query.txt` as the base user prompt. Then, it replaces the `$input$` placeholder with the actual optimization problem description.
   - **Optimization Goal Instruction Addition**: An instruction about specifying the optimization goal is appended to the user prompt.
   - **Iterative Generation and Validation**: The function enters a loop where it calls the GPT model using the prepared user prompt to generate a mathematical formula. Then, it constructs a validation prompt to check if the generated formula is consistent with the problem description. If the validation result indicates consistency, the loop is terminated.
   - **Optimization Flag Extraction**: After obtaining a valid mathematical formula, the function extracts the optimization flag from the formula. If `[Optimization Flag: 1]` is found, it sets `optimization_flag` to 1 (maximize); if `[Optimization Flag: 0]` is found, it sets `optimization_flag` to 0 (minimize).
   - **Return Result**: Finally, the function returns the generated mathematical formula and the optimization flag.

4. **Example of Generating a Mathematical Formula**
Suppose the optimization problem is: "In a resource allocation problem, we have N entities. The goal is to maximize the total utility of all entities. The utility of each entity i is given by $R_i = \log_2(1 + \frac{p_i h_i}{\sigma^2})$, where p_i is the resource allocated to entity i, h_i is the utility gain parameter of entity i, and σ^2 is a base constraint parameter. The total available resources is P_{total}, so $\sum_{i=1}^{N} p_i \leq P_{total}$, and $p_i \geq 0$ for all i." After replacing the `$input$` in the prompt and adding the optimization goal instruction, the GPT model might generate the following mathematical formula:

$$\max_{p_1, p_2, \cdots, p_N} \sum_{i=1}^{N} \log_2 \left(1 + \frac{p_i h_i}{\sigma^2}\right)$$

subject to

$$\begin{cases} \sum_{i=1}^{N} p_i \leq P_{total} \\ p_i \geq 0, \quad i = 1, 2, \cdots, N \end{cases}$$

and the formula might also contain `[Optimization Flag: 1]` to indicate maximization.

G.2.2 Convex Optimization Conversion Stage

1. **Prompt File Overview**
   In this stage, two prompt files `convex_example.txt` and `convex_query.txt` are used. `convex_example.txt` provides comprehensive examples of converting non-convex problems into convex problems, serving as references and learning templates for the large language model (LLM). These examples include detailed analyses of non-convex parts, convexification approaches, formula derivations, and the final convex optimization problems. `convex_query.txt` contains the input information of the specific non-convex mathematical formula to be convexly converted, guiding the model to perform the conversion operation by specifying clear requirements for the output.
2. **Original Text and Analysis of `convex_example.txt` and `convex_query.txt`**

- **Original Text and Analysis of `convex_example.txt`**
  The original text content of `convex_example.txt` is as follows:

`convex_example.txt`

Remember your job: 1. Following the input and output format, aligning with the length and context.
Example:
1: Input:

(a) **Mathematical Formulation: Utility Maximization Problem**

(b) **Objective Function** The goal is to maximize the total utility across all entities:

$$\max_{p,\alpha} \sum_i \log \left(1 + \frac{|h_{ii}|^2 p_i}{\sum_{j \neq i} |h_{ij}|^2 p_j + \sigma^2}\right)$$

where:

- p_i: Resource allocated to entity i (continuous).
- h_{ii}: Utility gain parameter for entity i.
- h_{ij}: Interaction parameter from entity j to entity i (continuous).
- σ^2: Base constraint parameter.

(c) **Constraints**

i. **Resource Constraints**:

A. Individual entity resource limits:

$$0 \leq p_i \leq P_{\text{max}}, \quad \forall i$$

B. Total resource budget:

$$\sum_i p_i \leq P_{\text{total}}$$

ii. **Minimum Requirement Constraints**: Each entity must satisfy the minimum requirement:

$$\frac{|h_{ii}|^2 p_i}{\sum_{j \neq i} |h_{ij}|^2 p_j + \sigma^2} \geq \gamma_{\text{min}}, \quad \forall i$$

iii. **Allocation Constraints**: Each entity can only be assigned to one category:

$$\alpha_{ik} \in \{0, 1\}, \quad \forall i, \quad \sum_k \alpha_{ik} = 1$$

(d) **Additional Variable Definitions**

i. **Utility Gain Parameter Definitions**:

A. **Scaling Function**: $L(d)$, which can be expressed as:

$$L(d) = L_0 + 10n \log_{10}\left(\frac{d}{d_0}\right)$$

where L_0 is the reference value at distance d_0, n is a scaling exponent, and d is a distance parameter.

B. **Gain Model**: The utility gain parameters can be modeled as:

$$h_{ii} = \sqrt{g_{ii}} L(d_{ii})$$

$$h_{ij} = \sqrt{g_{ij}} L(d_{ij})$$

where g_{ii} and g_{ij} are random variables representing variability, typically modeled as complex Gaussian variables.

Output:

(a) **Identification of Non-Convex Parts** The optimization problem presented contains several non-convex components due to the following:

i. **Logarithmic Function**: The objective function includes a logarithmic term, which is concave, but the overall maximization creates a non-convex problem.

ii. **Ratio Constraints**: The ratio constraints can also introduce non-convexity because they involve ratios of quadratic forms.

iii. **Binary Variables**: The use of binary variables α_{ik} creates a mixed-integer programming (MIP) problem, which is generally non-convex.

(b) **Convexification Approach** To address the non-convexities, the following strategies can be employed:

i. **Change of Variables**: Introduce auxiliary variables to reformulate the ratio constraints into a convex form.

ii. **Logarithm Reformulation**: Convert the logarithmic objective to a form suitable for optimization.

iii. **Relaxation of Binary Variables**: Use continuous relaxation for binary variables where appropriate.

(c) **Derivation for Convexification**

i. **Logarithmic Objective Reformulation**: Let $z_i = \frac{|h_{ii}|^2 p_i}{\sum_{j \neq i} |h_{ij}|^2 p_j + \sigma^2}$. Instead of maximizing $\log(1 + z_i)$, we can maximize z_i directly under certain transformations.

ii. **Ratio Constraints**: To make the ratio constraints convex, we can rewrite:

$$|h_{ii}|^2 p_i \geq \gamma_{\text{min}} \left(\sum_{j \neq i} |h_{ij}|^2 p_j + \sigma^2 \right)$$

This can be treated as a convex constraint by rearranging terms to isolate p_i.

iii. **Relaxation of Binary Variables**: Instead of using binary variables α_{ik}, introduce continuous variables $0 \leq \alpha_{ik} \leq 1$ such that:

$$\sum_k \alpha_{ik} = 1 \quad \forall i$$

(d) **Convex Optimization Problem** After applying these transformations, the new optimization problem can be expressed as follows:

i. **Objective Function** Maximize:

$$\max_{p,\alpha} \sum_i z_i$$

where z_i is defined as:

$$z_i = \frac{|h_{ii}|^2 p_i}{\sum_{j \neq i} |h_{ij}|^2 p_j + \sigma^2}$$

ii. **Constraints**

A. **Resource Constraints**:

$$0 \leq p_i \leq P_{\text{max}}, \quad \forall i$$

$$\sum_i p_i \leq P_{\text{total}}$$

B. **Convex Ratio Constraints**:

$$|h_{ii}|^2 p_i \geq \gamma_{\text{min}} \left(\sum_{j \neq i} |h_{ij}|^2 p_j + \sigma^2 \right), \quad \forall i$$

C. **Relaxed Allocation Constraints**:

$$0 \leq \alpha_{ik} \leq 1, \quad \forall i, k$$

$$\sum_k \alpha_{ik} = 1, \quad \forall i$$

iii. **Supplementary Variables**

A. **Utility Gain Parameters**: The utility gain parameters can still be defined as:

$$h_{ii} = \sqrt{g_{ii}} L(d_{ii}), \quad h_{ij} = \sqrt{g_{ij}} L(d_{ij})$$

This reformulated optimization problem is now in a convex form and can be directly addressed using standard convex optimization solvers, allowing for efficient computation and resource allocation.

The examples in this file provide ideas and methods for convex conversion for the LLM. By demonstrating different types of non-convex problems and their convex forms after conversion, the model can learn common convex conversion techniques such as change of variables, logarithm reformulation, and relaxation of binary variables.

- **Original Text and Analysis of `convex_query.txt`**
The original text content of `convex_query.txt` is as follows:

`convex_query.txt`

Please identify if there are any non-convex parts based on the mathematical formula of the optimization problem you built in the previous step and the type of corresponding variables, and if so, choose the algorithm that you think

is most appropriate to make all the non-convex components convex (including mixed integer programming problems) until you can solve them directly with the solver. And tell me the derivation of the formula to make it convex, and finally, give me the optimization problem after the convexity, including some supplementary variable formulas. Input: $input$
Output:

The `$input$` in it is also a placeholder, which will be replaced by the actual non-convex mathematical formula. This prompt directly requires the model to perform a comprehensive convex conversion operation on the input non-convex formula, including identifying non-convex parts, choosing appropriate convexification algorithms, providing formula derivations, and presenting the final convex optimization problem.

3. **Code Explanation: `transfer_convex` Function** The `transfer_convex` function is designed to convert a non-convex mathematical formula into a convex one using the LLM. Here is a step-by-step breakdown of what the function does:
   (a) **Initialization**: The function initializes an empty string `convex` to store the converted convex mathematical formula.
   (b) **Prompt Loading**: It reads the system prompt from `convex_example.txt` and the user prompt from `convex_query.txt` using the `pure_set_prompt` function. These prompts serve as guidelines for the LLM to understand the task and learn from the provided examples.
   (c) **Prompt Modification**: The function replaces the `$input$` placeholder in the user prompt with the actual non-convex mathematical formula (`math`) passed as an argument.
   (d) **Model Call**: It calls the LLM using the `call_gpt_from_sys` function, passing the system prompt and the modified user prompt. The LLM then processes the input and generates the convex version of the mathematical formula.
   (e) **Result Return**: Finally, the function returns the converted convex mathematical formula.

G.2.3 Code Generation Stage

1. **Prompt File Overview**
   In the code generation stage, the prompt files `code_example.txt` and `code_query.txt` are used. `code_example.txt` contains some examples of generating code from mathematical formulas, demonstrating the structure and implementation of the code. `code_query.txt` provides the convex optimization formula for which code needs to be generated, guiding the large language model to generate the corresponding executable code.
2. **Original Text and Analysis of `code_example.txt` and `code_query.txt`**

   - **Original Text of `code_example.txt`**
     The original text content of `code_example.txt` is as follows:

`code_example.txt`

Suppose we have a simple convex optimization problem with the objective of minimizing the objective function $f(x) = x^2+2x+1$, where x is a real-valued variable. Here is a code example using Python and the `scipy.optimize` library to solve this problem:

```
from scipy.optimize import minimize

# Define the objective function
def objective(x):
    return x**2 + 2*x + 1

# Initial guess
```

```
x0 = 0

# Call the optimizer
result = minimize(objective, x0)

# Output the result
if result.success:
    print(f"Optimization succeeded. The optimal
        solution is: {result.x[0]}, and the optimal
        value is: {result.fun}")
else:
    print("Optimization failed")
```

- **Analysis of `codeexample.txt`**
  The examples in `codeexample.txt` provide the large language model with a specific paradigm for translating mathematical formulas into code implementations. It shows how to use a specific Python library (such as scipy.optimize) to solve convex optimization problems. The example details the definition of the objective function, the setting of the initial guess, and the invocation of the optimizer. This helps the model understand the general process of code generation for convex optimization problems, including how to represent mathematical concepts in code, how to choose appropriate libraries and functions, and how to handle the results of the optimization process.
- **Original Text of `codequery.txt`**
  The original text content of `codequery.txt` is as follows:

`codequery.txt`

Please generate Python code to solve the following convex optimization problem. The convex optimization formula is: *input*. Use appropriate optimization libraries (e.g., scipy.optimize). Clearly define the objective function, constraints (if any), and initial guesses. Provide clear comments in the code to explain the key steps.

- **Analysis of `codequery.txt`**
  `codequery.txt` guides the large language model in code generation in multiple ways. Firstly, it directly specifies the input, which is the convex optimization formula, enabling the model to focus on the specific problem. Secondly, it recommends using appropriate optimization libraries, giving the model a clear direction on where to find the necessary tools for solving the problem. Thirdly, by requiring the clear definition of the objective function, constraints, and initial guesses, it ensures that the generated code is complete and accurate. The requirement for clear comments in the code also helps to make the generated code more understandable and maintainable.

G.2.4 Code Execution Stage

1. **Prompt File Overview** In the code execution stage, the prompt file `executecodequery.txt` is used. Its main purpose is to guide the large language model to analyze and handle the execution results of the generated code, including error handling and result extraction.
2. **Original Text and Analysis of `executecodequery.txt`**
   - **Original Text of `executecodequery.txt`** The original text content of `executecodequery.txt` is as follows:

`execute_code_query.txt`

The following is the code generated to solve the convex optimization problem: *code*. Execute this code. If the execution fails, analyze the error message and suggest possible solutions. If the execution is successful, extract the optimal

solution and the optimal value from the output, and present them in a clear format.

- **Analysis of `execute_code_query.txt`**
  This prompt guides the large language model to perform two main tasks. When the code execution fails, it requires the model to analyze the error message, which helps the model understand common error types in code execution for convex optimization problems, such as incorrect function calls, improper parameter settings, or library import issues. Based on the analysis, the model is expected to suggest possible solutions, which promotes the model's ability to troubleshoot and optimize code. When the code execution is successful, the model is required to extract the optimal solution and the optimal value, which trains the model's ability to understand and process the output of the optimization process.

G.2.5 Feasibility Check Stage

1. **Prompt File Overview**
   In the feasibility check stage, the prompt file `feasibilitycheckquery.txt` is used. Its function is to guide the large language model to determine whether the obtained solution of the optimization problem is within the feasible region based on the original problem description and the mathematical formula.

2. **Original Text and Analysis of `feasibilitycheckquery.txt`**

   - **Original Text of `feasibilitycheckquery.txt`**
     The original text content of `feasibilitycheckquery.txt` is as follows:

     `feasibility_check_query.txt`

     The original convex optimization problem description is: $problem_description$. The corresponding mathematical formula is: $mathformula$. The obtained solution of the optimization problem is: $solution$. Determine whether the solution is within the feasible region. If it is, return 1; if not, return 0.

   - **Analysis of `feasibility_check_query.txt`**
     This prompt provides the large language model with all the necessary information for a feasibility check, including the problem description, the mathematical formula, and the obtained solution. By asking the model to determine whether the solution is within the feasible region and return a binary result, it forces the model to understand the constraints and conditions in the convex optimization problem and use logical reasoning to make a judgment. This helps to ensure the quality and validity of the final solution.

H Case Study

H.1 Problem

H.1.1 Problem Description

Consider a non-convex optimization problem where the objective is to maximize a utility function subject to resource constraints. The problem involves allocating resources to multiple entities to maximize the total utility while ensuring that each entity meets a minimum requirement.
This optimization problem aims to optimize resource allocation to enhance overall performance while maintaining feasibility for all entities.

H.1.2 VARIABLE PARAMETERS

1. **Number of Entities**: 5
2. **Maximum Resource per Entity**: 30 units
3. **Minimum Requirement per Entity**: 10 units
4. **Utility Function Parameters**: The utility gain is modeled as:

$$G_{ij} = \frac{1}{(d_{ij})^{\gamma}} \quad (\gamma = 2.2, d_{ij} \text{ is a distance parameter between entity } i \text{ and } j)$$

5. **Base Constraint Parameter**: $N_0 = 1 \times 10^{-9}$
6. **Total Available Resources**: 120 units

H.2 ANSWER

H.3 MATHEMATICAL FORMULATION

We start by formulating the problem mathematically. Given an optimization problem with the goal of optimizing resource allocation to maximize utility subject to certain constraints.

H.3.1 OBJECTIVE FUNCTION

The objective is to maximize the overall system utility, denoted as R:

$$\text{maximize } R = \sum_{i=1}^{N} \log_2 \left(1 + \frac{P_i |h_i|^2}{N_0}\right)$$

where:

- N is the number of entities.
- P_i is the resource allocated to entity i.
- h_i is the utility gain parameter for entity i.
- N_0 is a base constraint parameter.

H.3.2 CONSTRAINTS

1. **Total resource constraint**:
$$\sum_{i=1}^{N} P_i \leq P_{\text{max}}$$
where P_{max} is the maximum total available resources.
2. **Non-negativity constraints for resources**:
$$P_i \geq 0, \quad \forall i = 1, 2, \ldots, N$$
3. **Minimum requirement for each entity (optional)**:
$$R_i \geq R_{\text{min}}, \quad \forall i = 1, 2, \ldots, N$$
where $R_i = \log_2 \left(1 + \frac{P_i |h_i|^2}{N_0}\right)$ is the utility for entity i and R_{min} is the minimum required utility.

H.3.3 VARIABLES AND FORMULAS

- **Utility Gain Parameter** h is typically modeled as:
$$h_i = \sqrt{G_i} \cdot \exp(j\theta_i)$$
where:
  - G_i represents the gain parameter for entity i (can vary based on problem context).

  - θ_i is a phase parameter (uniformly distributed in $[0, 2\pi]$).
- **Base Constraint Parameter** N_0 can be calculated as:

$$N_0 = kT \cdot B$$

where:

  - k is a constant parameter.
  - T is a scaling factor.
  - B is a bandwidth or capacity parameter.

H.3.4 Optimization Flag

Since the objective is to maximize utility:

[Optimization Flag: 1]

This formulation sets up an optimization problem that can be solved using various optimization techniques and algorithms to find the optimal resource allocation P_i for each entity in the system. The parameters specified in the problem can be adjusted and implemented according to the model being studied.

H.4 Identification of Non-Convex Parts

The optimization problem described contains several non-convex components:

1. **Logarithmic Function**: Although the logarithmic function is concave, maximizing the sum of individual utility terms $R_i = \log_2\left(1 + \frac{P_i|h_i|^2}{N_0}\right)$ creates a non-convex problem due to the interactions between different P_i variables.
2. **Minimum Requirement Constraints**: The constraints $R_i \geq R_{\text{min}}$ involve logarithmic expressions, which can lead to non-convexities, especially when the resource allocation is in the denominator.
3. **Resource Constraints**: The total resource constraint coupled with individual non-negativity constraints does not inherently introduce non-convexity but can complicate the optimization landscape.

H.5 Proposed Algorithm for Convexification

To address these non-convexities, we can use the following strategies:

- **Change of Variables**: Reformulate the problem using auxiliary variables.
- **Convex Relaxation**: Transform existing logarithmic terms into convex equivalent expressions.
- **Use of Sequential Convex Programming (SCP)**: An iterative approach that optimizes a convex approximation of the original problem at each step.

A specific algorithm suitable for this type of problem is the **Alternating Direction Method of Multipliers (ADMM)**, which can be effective for dealing with non-convex optimization problems involving separable structures.

H.6 Derivation for Convexification

1. **Logarithmic Objective Reformulation**: We can rewrite R_i as:

$$R_i = \log_2\left(1 + \frac{P_i|h_i|^2}{N_0}\right)$$

Using a change of variable, let $z_i = P_i|h_i|^2$. Then we have:

$$R_i = \log_2\left(1 + \frac{z_i}{N_0}\right)$$

2. **Handling Minimum Requirement Constraints**: Instead of using the logarithmic form for minimum requirements directly, we can reformulate it to express resources:

$$\frac{P_i|h_i|^2}{N_0} \geq 2^{R_{\min}} - 1$$

Rearranging gives:

$$P_i|h_i|^2 \geq N_0(2^{R_{\min}} - 1)$$

This can be made into a convex constraint by separating P_i from the non-linear aspects.

3. **Relaxation**: The resource constraints remain linear since they are in the form of:

$$\sum_{i=1}^{N} P_i \leq P_{\max}$$

H.7 Convex Optimization Problem

After applying these transformations and simplifications, the new optimization problem can be expressed as follows:

H.7.1 Objective Function

$$\text{maximize } R = \sum_{i=1}^{N} z_i, \quad \text{where } z_i = \log_2\left(1 + \frac{P_i|h_i|^2}{N_0}\right)$$

H.7.2 Constraints

1. **Total Resource Constraint**:

$$\sum_{i=1}^{N} P_i \leq P_{\max}$$

2. **Non-negativity Constraints**:

$$P_i \geq 0, \quad \forall i = 1, 2, \ldots, N$$

3. **Convex Minimum Requirement Constraints**:

$$P_i|h_i|^2 \geq N_0(2^{R_{\min}} - 1), \quad \forall i = 1, 2, \ldots, N$$

H.7.3 Supplementary Variable Formulas

- **Utility Gain Parameter** remains defined as:

$$h_i = \sqrt{G_i} \cdot \exp(j\theta_i)$$

- **Base Constraint Parameter** can be expressed as:

$$N_0 = kT \cdot B$$

This reformulated optimization problem is now in a convex form and can be addressed using standard convex optimization solvers, enabling efficient computation and resource allocation.

H.8 Code Generation

The following is a Python code example using `scipy.optimize.minimize` to solve the optimization problem:

code

```python
import numpy as np
from scipy.optimize import minimize

# Parameters
N = 5  # Number of entities
P_max = 10  # Maximum total resources
R_min = 5  # Minimum requirement
G = np.array([1, 1, 1, 1, 1])  # Utility gain parameters
N0 = 0.001  # Base constraint parameter

# Objective function
def objective(P):
    z = np.log2(1 + (P * G) / N0)  # Calculate z_i
    return -np.sum(z)  # Minimize negative sum for maximization

# Constraints
def constraint_total_power(P):
    return P_max-np.sum(P)

def constraint_qos(P):
    return np.array([P[i] * G[i] - N0 * (2**(R_min) - 1) for i
        in range(N)])

# Initial guess
P0 = np.ones(N)

# Constraints dictionary
constr = [{'type': 'ineq', 'fun': constraint_total_power},
          {'type': 'ineq', 'fun': constraint_qos}]

# Bounds
bounds = [(0, P_max)] * N  # Each P_i must be between 0 and
    P_max

# Solve the optimization problem
result = minimize(objective, P0, bounds=bounds, constraints=
    constr)

# Output results
print("Objective Function Value:", -result.fun)
print("Optimized Resource Allocations (P_i):", result.x)  #
    Optimized resource allocations
```

H.9 Optimization Results

The output of the script shows:

Output

- The value of the objective function is 54.8325272595287.
- Optimized Resource Allocations (P_i) are $[2.2.2.2.2.]$.

The initial solution lies within the feasible region.