Automatic source code generation for deterministic global optimization with parallel architectures

Robert X. Gottlieb, Pengfei Xu, and Matthew D. Stuber

Process Systems and Operations Research Laboratory, Department of Chemical and Biomolecular Engineering, University of Connecticut, Storrs, CT, USA

ARTICLE HISTORY
Compiled January 27, 2026

ABSTRACT
Trends over the past two decades indicate that much of the performance gains of commercial optimization solvers is due to improvements in x86 hardware. To continue making progress, it is critical to consider alternative/specialized massively parallel computing architectures. In this work, we detail the development of an open-source source code transformation approach built using `Symbolics.jl` to construct McCormick-based relaxations of functions that enables their effective parallelized evaluation. We then apply this approach in a novel parallelized branch-and-bound routine that offloads lower- and upper-bounding problems to a GPU. The effectiveness of this new approach is demonstrated on three nonconvex problems of interest, where it yields convergence time improvements of 22–118x compared to an equivalent serial CPU implementation and in two cases outperforms vanilla branch-and-bound versions of existing state-of-the-art solvers that use tighter bounding techniques. This work exemplifies how deterministic global optimizers using alternative hardware architectures can compete with—or eventually outclass—even the most powerful serial CPU implementations, and to the best of the authors' knowledge, represents the first successful demonstration of deterministic global optimization using a GPU.

KEYWORDS
dynamical systems; parameter estimation; factorable programming; open-source software; McCormick relaxations

AMS CLASSIFICATION
90C26; 90-04; 65G30; 26B25; 65Y05

1. Introduction and motivation

Optimization research over the past several decades has led to an increase in the number and types of solvable problems, but as recently observed by Koch et al. [25], much of the perceived performance improvements of mixed-integer programming solvers were the result of advances in CPU hardware [25]. However, with the death of Moore's Law [37, 56], CPU hardware is not improving at the same high rate as in past decades.

Author's final accepted version. Published version: Gottlieb, R.X., Xu, P., and Stuber, M.D. Automatic Source Code Generation for Deterministic Global Optimization With Parallel Architectures. (2024) pp 1–39. DOI: 10.1080/10556788.2024.2396297

Corresponding author: M.D. Stuber. Email: stuber@alum.mit.edu

This performance slump is especially hard-hitting for deterministic global optimization solvers which are, to the best of our knowledge, exclusively designed to run on CPUs. Other application areas of process systems engineering that were traditionally designed for CPU operation, such as data analytics, simulation, and control, have found improved performance by utilizing alternative hardware architectures, including graphics processing units (GPUs) and application-specific integrated circuits (ASICs) [3, 4]. GPUs in particular are a common, cost-effective, and scalable way to gain massive speedups for specialized computations through parallel computing. The successes of parallelization in a variety of fields are well documented [15], yet to the best of our knowledge, there has been no publication documenting the use of GPUs for deterministic global optimization. With earlier preliminary results from this paper first reported at the AIChE Annual Meeting 2022 [16] and FOCAPO-CPC 2023 [18], this article aims to detail the full development of the method, software implementation, and benchmarking for the first successful implementation, to the best of our knowledge, of a deterministic global optimization algorithm designed to function on a GPU for massive parallelization.

Modern deterministic global optimization solvers (BARON [38], ANTIGONE [31], SCIP [7, 52], MAiNGO [8], EAGO [59], etc.) rely on some variation of spatial branch-and-bound (B&B), where, for each spatial subdomain, a mathematically rigorous lower bound of the objective function must be obtained. One way to obtain these lower bounds is to apply the rules laid out by McCormick [30]. These rules were later formalized for use in a *forward mode* to operate on generic algorithms by Mitsos et al. [32], generalized by Scott et al. [39], extended to implicit functions by Stuber et al. [44], developed for use in a *reverse mode* by Wechsung et al. [55], and modified to ensure differentiability by Khan et al. [23, 24], among other recent developments. Through the McCormick composition theorem provided in Section 2, the McCormick-based approaches enable convex and concave relaxations—and therefore lower and upper bounds, respectively—to be generated for arbitrarily complicated expressions. As observed by Mitsos et al. [32], the automatic construction of convex/concave relaxations via these rules is similar to the concept of automatic differentiation (AD). Due to this similarity, the implementation of automatic McCormick relaxations can be accomplished using the same techniques as AD; namely: operator overloading (OO) and source code transformation (SCT) [20].

Briefly, OO for AD involves the definition of a variable type for floating-point numbers and their associated gradients. Operations such as those for elementary arithmetic can be defined for these variable types to compute the result of the operation applied to the floating-point values as well as their associated gradients [6]. SCT, on the other hand, involves the automated rewriting or generation of source code that computes the derivative of an expression as a subroutine call [6].

AD is critically important for a variety of numerical computing tasks, including machine learning [2], computational fluid dynamics [22], and quantum chemistry [6]. In machine learning, for example, backpropagation and gradient-based optimization are ubiquitous for model training, and consequently several AD software tools have been developed specifically for use in machine learning applications [2]. In this application space, OO-based AD is more common (e.g., autograd [28], PyTorch [36], among others), but recent SCT implementations exist as well (e.g., Tangent [50, 51], etc.). As it pertains to the present work, there have been several successes in parallelizing these methods on GPU hardware that have accelerated the speed of machine learning model training and development [11, 26, 34, 49], although those developments did not directly guide this work. Generally, OO approaches are simpler to implement and can be flexibly used, but incur a runtime overhead time cost, whereas SCT approaches have much higher implementation complexity but can facilitate compiler optimizations that result in faster execution of the generated code [6].

For deterministic global optimization, all existing implementations of the rules of McCormick that we are aware of (e.g., `MC++` [9], `McCormick.jl` [57]) rely on OO, or the analogous multiple dispatch in the Julia language. In `McCormick.jl`, for example, the `MC` structure is defined, and new methods are created for the arithmetic operators and a library of univariate intrinsic functions that apply the McCormick calculation rules when they are dispatched on `MC` objects. Such an approach is adequate for cases where the relaxation calculations do not account for a high proportion of runtime requirements [32], but if relaxations are to be evaluated thousands or millions of times, as may happen in comparatively large global optimization problems, the combined overhead times may be substantial. Additionally, approaches such as `McCormick.jl` are incompatible with general-purpose computing on GPUs (GPGPU). Any implementation of GPGPU of a deterministic global optimizer that utilizes McCormick-based relaxations would therefore need an alternative method of evaluating those relaxations.

In this paper, we employ the SCT approach to interpret mathematical expressions and automatically generate static functions that represent convex underestimators, concave overestimators, and interval extensions of the original expressions. By design, these generated functions are compatible with GPU operation, enabling the parallelized evaluation of convex relaxations at multiple points. Additionally, a custom B&B routine has been developed to exploit this parallelism to solve lower-bounding problems of the B&B framework entirely on a GPU.

The remainder of this paper is structured as follows. Section 2 describes the mathematical conventions used in this paper and provides a brief overview of convex analysis and the rules of McCormick. In Section 3, we describe the implementation of the SCT approach capable of generating convex evaluator functions and describe how this approach can be used within deterministic global optimization contexts. Subsequently, in Section 4, we present the numerical results arising from a benchmark comparison between the SCT-based product of this work and the multiple dispatch approach of `McCormick.jl`, and then provide several numerical examples of optimization problems solved using this approach. Lastly, in Section 6, we reflect on the technical challenges associated with this new method and software implementation, and we present our plans for future improvements.

2. Mathematical background

2.1. Interval arithmetic

The following notation will be used throughout this paper. Scalar quantities are denoted by lower-case letters (e.g., a) and vectors are denoted by boldface lower-case letters (e.g., $\mathbf{a}$). A nonempty compact set $A = \left[\mathbf{a}^L, \mathbf{a}^U\right]$ represents an n-dimensional interval defined as $A = \left\{\mathbf{a} \in \mathbb{R}^n : \mathbf{a}^L \leq \mathbf{a} \leq \mathbf{a}^U\right\}$ with $\mathbf{a}^L$ and $\mathbf{a}^U$ the lower and upper bounds of the interval, respectively, and A_i represents the i-th component of the interval A. A set B^n is defined as the Cartesian product $B^n \equiv B \times B \times \ldots \times B$ for $B \subset \mathbb{R}$. Additionally, let $\mathbb{IR}^n$ be the set of all n-dimensional intervals and for any $D \subset \mathbb{R}^n$, $\mathbb{I}D = \{X \in \mathbb{IR}^n : X \in D\}$ is the set of all interval subsets of D. Furthermore, let the *diameter* of an interval $X \in \mathbb{IR}^n$ be defined as $\mathrm{diam}\,(X) = \mathbf{x}^U - \mathbf{x}^L$ and the *radius* be given by $\mathrm{rad}\,(X) = \mathrm{diam}\,(X)\,/2$. An *inclusion monotonic interval extension* [33, Sections 3.2–3.3] of the real-valued mapping $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ on the interval $X \in \mathbb{IR}^n$ will be denoted by $F(X) = [\mathbf{f}^L(X), \mathbf{f}^U(X)]$. The image of $D \subset \mathbb{R}^n$ under the mapping $\mathbf{f}$ will be denoted $\hat{\mathbf{f}}(D)$. From the *Fundamental Theorem of Interval Analysis* [33, Theorem 3.1], $\hat{\mathbf{f}}(X) \subset F(X)$.

2.2. Convex and concave relaxations

Definition 2.1 (Convex and Concave Relaxations [32])**.** Given a convex set $Z \subset \mathbb{R}^n$ and a function $f : Z \to \mathbb{R}$, a convex function $f^{cv} : Z \to \mathbb{R}$ is a *convex relaxation* of f on Z if $f^{cv}(\mathbf{z}) \leq f(\mathbf{z})$ for every $\mathbf{z} \in Z$. A concave function $f^{cc} : Z \to \mathbb{R}$ is a *concave relaxation* of f on Z if $f^{cc}(\mathbf{z}) \geq f(\mathbf{z})$ for every $\mathbf{z} \in Z$.

Note that convex and concave relaxations of vector-valued functions $\mathbf{f} : Z \to \mathbb{R}^m$ are defined by applying the inequalities in Definition 2.1 componentwise [44].

Definition 2.2 (Convex and Concave Envelope [60])**.** Let $f : Z \to \mathbb{R}$ where $Z \subset \mathbb{R}^n$ is a nonempty convex set. The *convex envelope* of f on Z is the convex relaxation $f^{cv,env} : Z \to \mathbb{R}$ such that $f^{cv}(\mathbf{z}) \leq f^{cv,env}(\mathbf{z})$ holds for all $\mathbf{z} \in Z$ and every convex relaxation f^{cv} of f on Z. Similarly, the *concave envelope* of f on Z is the concave relaxation $f^{cc,env} : Z \to \mathbb{R}$ such that $f^{cc}(\mathbf{z}) \geq f^{cc,env}(\mathbf{z})$ holds for all $\mathbf{z} \in Z$ and every concave relaxation f^{cc} of f on Z.

Definition 2.3 (Univariate Intrinsic Function [39])**.** The function $u : B \subset \mathbb{R} \to \mathbb{R}$ is a *univariate intrinsic function* if, for any $A \in \mathbb{I}B$, the following are known and can be evaluated computationally:

- an inclusion monotonic interval extension of u on A,
- a convex relaxation of u on A,
- a concave relaxation of u on A.

Definition 2.4 (Factorable Function [39])**.** A function $\mathscr{F} : Z \subset \mathbb{R}^n \to \mathbb{R}$ is *factorable* if it can be expressed in terms of a finite number of factors $v_1, \ldots, v_m$, such that given $\mathbf{z} \in Z$, $v_i = z_i$ for $i = 1, \ldots, n$, and v_k is defined for $n < k < m$ as either

- $v_k = v_i + v_j$, with $i, j < k$, or
- $v_k = v_i v_j$, with $i, j < k$, or
- $v_k = u_k(v_i)$, with $i < k$, where $u_k : B_k \to \mathbb{R}$ is a univariate intrinsic function,

and $\mathscr{F}(\mathbf{z}) = v_m(\mathbf{z})$ for every $\mathbf{z} \in Z$. A vector-valued function is factorable if each of its components is a factorable function.

Definition 2.5 (Cumulative Mapping [39])**.** Let the *cumulative mapping* v_k be the mapping $v_k : Z \to \mathbb{R}$ defined for each $\mathbf{z} \in Z$ by the value $v_k(\mathbf{z})$ when the factors of $\mathscr{F}$ are computed recursively, as per Definition 2.4, beginning from $\mathbf{z}$.

Definition 2.6 (Composite Relaxations [44])**.** Let $D \subset \mathbb{R}^n$, $Z \in \mathbb{I}D$, and $P \in \mathbb{IR}^{n_p}$. Define the mapping $\mathcal{G} : D \times P \to \mathbb{R}^{n_\mathcal{G}}$. The functions $\mathbf{u}_\mathcal{G}, \mathbf{o}_\mathcal{G} : \mathbb{R}^n \times \mathbb{R}^n \times P \to \mathbb{R}^{n_\mathcal{G}}$ are called *composite relaxations* of $\mathcal{G}$ on $Z \times P$ if for $\boldsymbol{\psi}^{cv}, \boldsymbol{\psi}^{cc} : P \to \mathbb{R}^n$, the functions $\mathbf{u}_\mathcal{G}(\boldsymbol{\psi}^{cv}(\cdot), \boldsymbol{\psi}^{cc}(\cdot), \cdot)$ and $\mathbf{o}_\mathcal{G}(\boldsymbol{\psi}^{cv}(\cdot), \boldsymbol{\psi}^{cc}(\cdot), \cdot)$ are, respectively, convex and concave relaxations of $\mathcal{G}(\mathbf{q}(\cdot), \cdot)$ on P for any function $\mathbf{q} : P \to Z$ and any pair of convex and concave relaxations $(\boldsymbol{\psi}^{cv}, \boldsymbol{\psi}^{cc})$ of $\mathbf{q}$ on P.

The concept of composite relaxations is introduced to formalize the notion of constructing convex and concave relaxations of composite functions on the domain of the inner function. It makes explicit the relationship of convex and concave relaxations of the outer function on its domain with respect to convex and concave relaxations of the inner function on its domain. This is somewhat nuanced but is useful herein because convex and concave relaxations of the inner function on its domain may be known a priori or calculated by some other algorithmic procedure, and, through the notion of the composite relaxation, can simply be treated as convex and concave relaxations of a cumulative mapping on its domain and used to define convex and concave relaxations of a composite function involving the cumulative mapping. This property is explicitly referenced and utilized in Section 3.3.

2.3. McCormick relaxations

Definition 2.7 (McCormick Relaxations [32])**.** Relaxations of factorable functions that are formed from the recursive application of univariate composition, binary multiplication, and binary addition from convex and concave relaxations of univariate intrinsic functions, without the introduction of auxiliary variables, are referred to as *McCormick relaxations.*

The developments of Section 3 rely on *generalized* McCormick relaxations—under which the relaxations of Definition 2.7 are a special case—that were developed by Scott et al. [39]. Summarily, Scott et al. [39] formalized a more general definition and construction framework than the relaxations defined by McCormick [30], which made it possible to construct composite relaxations described by Definition 2.6. The reader is directed to [39] for complete definitions and analyses of both the *standard* and *generalized* McCormick relaxations.

A well-known alternative to the construction of McCormick relaxations for use in a B&B algorithm is the use of the *auxiliary variable method* [30, 41, 45, 47], in which new variables are created to replace univariate compositions and binary operations. That is, each v_k in Definition 2.4 is maintained as a new problem variable. This method raises the dimensionality of the problem, which enables some mathematical simplifications and allows for potentially tighter relaxations of the problem than can typically be obtained from McCormick relaxations [46]. However, the introduction of auxiliary variables increases the dimensionality of the problem, and for certain problems this may not be beneficial. The auxiliary variable method is utilized to great effect by the commercial solvers BARON [38] and ANTIGONE [31].

Definition 2.8 (Mid Function [32])**.** Given three numbers $\alpha, \beta, \gamma \in \mathbb{R}$, the mid function is defined as

$$\operatorname{mid}\{\alpha,\beta,\gamma\} = \begin{cases} \alpha & \text{if } \beta \le \alpha \le \gamma \text{ or } \gamma \le \alpha \le \beta, \\ \beta & \text{if } \alpha \le \beta \le \gamma \text{ or } \gamma \le \beta \le \alpha, \\ \gamma & \text{if } \alpha \le \gamma \le \beta \text{ or } \beta \le \gamma \le \alpha. \end{cases}$$

Proposition 2.9 (Relaxations of Sums [32])**.** *Let $Z \subset \mathbb{R}^n$ be a nonempty convex set, and $g, g_1, g_2 : Z \to \mathbb{R}$ such that $g(\mathbf{z}) = g_1(\mathbf{z}) + g_2(\mathbf{z})$. Let $g_1^{cv}, g_1^{cc} : Z \to \mathbb{R}$ be a convex and concave relaxation of g_1 on Z, respectively. Similarly, let $g_2^{cv}, g_2^{cc} : Z \to \mathbb{R}$ be a convex and concave relaxation of g_2 on Z, respectively. Then, $g^{cv}, g^{cc} : Z \to \mathbb{R}$, such* *that*

$$g^{cv}(\mathbf{z}) = g_1^{cv}(\mathbf{z}) + g_2^{cv}(\mathbf{z}), \quad g^{cc}(\mathbf{z}) = g_1^{cc}(\mathbf{z}) + g_2^{cc}(\mathbf{z}),$$

are, respectively, a convex and concave relaxation of g on Z.

Proposition 2.10 (Relaxations of Products [32])**.** *Let $Z \subset \mathbb{R}^n$ be a nonempty convex set, and $g, g_1, g_2 : Z \to \mathbb{R}$ such that $g(\mathbf{z}) = g_1(\mathbf{z})g_2(\mathbf{z})$. Let $g_1^{cv}, g_1^{cc} : Z \to \mathbb{R}$ be a convex and concave relaxation of g_1 on Z, respectively. Similarly, let $g_2^{cv}, g_2^{cc} : Z \to \mathbb{R}$ be a convex and concave relaxation of g_2 on Z, respectively. Furthermore, let $g_1^L, g_1^U, g_2^L, g_2^U \in \mathbb{R}$ such that*

$$g_1^L \le g_1(\mathbf{z}) \le g_1^U \quad \text{for all } \mathbf{z} \in Z \quad \text{and} \quad g_2^L \le g_2(\mathbf{z}) \le g_2^U \quad \text{for all } \mathbf{z} \in Z.$$

Consider the following intermediate functions, $\alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2, \delta_1, \delta_2 : Z \to \mathbb{R}$:

$$\begin{aligned}
\alpha_1(\mathbf{z}) &= \min\left\{g_2^L g_1^{cv}(\mathbf{z}), g_2^L g_1^{cc}(\mathbf{z})\right\}, & \alpha_2(\mathbf{z}) &= \min\left\{g_1^L g_2^{cv}(\mathbf{z}), g_1^L g_2^{cc}(\mathbf{z})\right\},\\
\beta_1(\mathbf{z}) &= \min\left\{g_2^U g_1^{cv}(\mathbf{z}), g_2^U g_1^{cc}(\mathbf{z})\right\}, & \beta_2(\mathbf{z}) &= \min\left\{g_1^U g_2^{cv}(\mathbf{z}), g_1^U g_2^{cc}(\mathbf{z})\right\},\\
\gamma_1(\mathbf{z}) &= \max\left\{g_2^L g_1^{cv}(\mathbf{z}), g_2^L g_1^{cc}(\mathbf{z})\right\}, & \gamma_2(\mathbf{z}) &= \max\left\{g_1^U g_2^{cv}(\mathbf{z}), g_1^U g_2^{cc}(\mathbf{z})\right\},\\
\delta_1(\mathbf{z}) &= \max\left\{g_2^U g_1^{cv}(\mathbf{z}), g_2^U g_1^{cc}(\mathbf{z})\right\}, & \delta_2(\mathbf{z}) &= \max\left\{g_1^L g_2^{cv}(\mathbf{z}), g_1^L g_2^{cc}(\mathbf{z})\right\}.
\end{aligned}$$

Then, α_1, α_2, β_1, β_2 are convex on Z, while γ_1, γ_2, δ_1, and δ_2 are concave on Z. Moreover, $g^{cv}, g^{cc} : Z \to \mathbb{R}$, such that

$$\begin{aligned}
g^{cv}(\mathbf{z}) &= \max\left\{\alpha_1(\mathbf{z}) + \alpha_2(\mathbf{z}) - g_1^L g_2^L,\ \beta_1(\mathbf{z}) + \beta_2(\mathbf{z}) - g_1^U g_2^U\right\},\\
g^{cc}(\mathbf{z}) &= \min\left\{\gamma_1(\mathbf{z}) + \gamma_2(\mathbf{z}) - g_1^U g_2^L,\ \delta_1(\mathbf{z}) + \delta_2(\mathbf{z}) - g_1^L g_2^U\right\},
\end{aligned}$$

are, respectively, a convex and concave relaxation of g on Z.

Theorem 2.11 (McCormick's Composition Theorem [30, 32])**.** *Let $Z \subset \mathbb{R}^n$ and $X \subset \mathbb{R}$ be nonempty convex sets. Consider the composite function $g = F \circ f$, where $f : Z \to \mathbb{R}$ is continuous, $F : X \to \mathbb{R}$, and let $f(Z) \subset X$. Suppose that a convex relaxation $f^{cv} : Z \to \mathbb{R}$ and a concave relaxation $f^{cc} : Z \to \mathbb{R}$ of f on Z are known. Let $F^{cv} : X \to \mathbb{R}$ be a convex relaxation of F on X, let $F^{cc} : X \to \mathbb{R}$ be a concave relaxation of F on X, let $x^{\min} \in X$ be a point at which F^{cv} attains its infimum on X, and let $x^{\max} \in X$ be a point at which F^{cc} attains its supremum on X. Then, $g^{cv} : Z \to \mathbb{R}$,*

$$g^{cv}(\mathbf{z}) = F^{cv}\left(\operatorname{mid}\left\{f^{cv}(\mathbf{z}), f^{cc}(\mathbf{z}), x^{\min}\right\}\right)$$

is a convex relaxation of g on Z, and $g^{cc} : Z \to \mathbb{R}$,

$$g^{cc}(\mathbf{z}) = F^{cc}\left(\operatorname{mid}\left\{f^{cv}(\mathbf{z}), f^{cc}(\mathbf{z}), x^{\max}\right\}\right)$$

is a concave relaxation of g on Z. By definition $f^{cv}(\mathbf{z}) \le f^{cc}(\mathbf{z})$, and therefore,

$$\operatorname{mid}\left\{f^{cv}(\mathbf{z}), f^{cc}(\mathbf{z}), x^{\min}\right\} = \begin{cases} f^{cv}(\mathbf{z}) & \text{if } x^{\min} < f^{cv}(\mathbf{z}),\\ f^{cc}(\mathbf{z}) & \text{if } x^{\min} > f^{cc}(\mathbf{z}),\\ x^{\min} & \text{otherwise,} \end{cases}$$

$$\operatorname{mid}\{f^{cv}(\mathbf{z}), f^{cc}(\mathbf{z}), x^{\max}\} = \begin{cases} f^{cv}(\mathbf{z}) & \textit{if } x^{\max} < f^{cv}(\mathbf{z}), \\ f^{cc}(\mathbf{z}) & \textit{if } x^{\max} > f^{cc}(\mathbf{z}), \\ x^{\max} & \textit{otherwise.} \end{cases}$$

2.4. Spatial branch-and-bound

Consider a nonconvex NLP:

$$\begin{aligned} f^* = f(\mathbf{x}^*) = &\min_x f(\mathbf{x}) \\ &\text{s.t. } \mathbf{x} \in \mathcal{F} \subset \mathbb{R}^n, \end{aligned}$$

where $f : D \to \mathbb{R}$ is the objective function and $\mathcal{F} \subset D \subset \mathbb{R}^n$ is the *feasible set.* Assuming that f^* exists and it is possible to compute suitable lower bounds for $\min \hat{f}(M)$ for at least some sets $M \subset D$, a B&B method can be applied [21]. As laid out by Horst and Tuy [21], the basic idea of the spatial B&B framework is summarized in Algorithm 1. The reader is directed to Wilhelm and Stuber [59, Alg. 3.1] for a detailed B&B algorithm as it is implemented in the open-source solver EAGO. Summarily, EAGO follows the framework described by Algorithm 1 where upper bounds for partition elements are obtained by passing subproblems to an NLP solver such as EAGO's default IPOPT [53] and lower bounds for partition elements are determined by generating subtangent hyperplanes of McCormick relaxations (Def. 2.7) to create a linear programming (LP) problem, which is then passed to an LP solver such as EAGO's default Cbc [14]. EAGO also has several pre- and post-processing features that are described in greater detail by Wilhelm and Stuber [59].

Algorithm 1 Spatial Branch-and-Bound (B&B)

(1) Start with a relaxed feasible set $M^{(0)} \supset \mathcal{F}$, such that $M^{(0)} \in \mathbb{I}D$ and partition $M^{(0)}$ into finitely many subsets $M^{(i)}, i = 1, \ldots, n_\sigma$.

(2) For each i, determine lower and (if possible) upper bounds LBD_i and UBD_i, respectively, satisfying:

$$LBD_i \leq \inf \hat{f}(M^{(i)} \cap \mathcal{F}) \leq UBD_i.$$

Then $LBD := \min_i LBD_i$, $UBD := \min_i UBD_i$ are *global* bounds, meaning that

$$LBD \leq \min \hat{f}(\mathcal{F}) \leq UBD.$$

(3) If $UBD = LBD$ (or, for some prescribed $\varepsilon_{\text{abs}}, \varepsilon_{\text{rel}} > 0$, we have $UBD - LBD \leq \varepsilon_{\text{abs}}$ or $|UBD - LBD|/\max(|UBD|, |LBD|) \leq \varepsilon_{\text{rel}}$), then stop.

(4) Otherwise, select some of the subsets $M^{(i)}$ and partition these chosen subsets to obtain a refined partition of $M^{(0)}$. Return to Step (2) to determine new, hopefully tighter global bounds using the refined partition.

Step 4 of Algorithm 1 involves partitioning an element $M^{(i)}$, which is a process referred to as "branching." Branching is most commonly done by bisecting an existing element in one dimension of the decision variables (i.e., the variable is "branched on"). This partitioning strategy is commonly visualized as an undirected graph and as a B&B *tree* whose nodes are elements of the partition. When an element of the current partition is branched, the depth of the B&B tree increases with the addition of two child nodes. All terminal nodes are referred to as "leaves" and collectively comprise a partition of $M^{(0)}$. Because the B&B algorithm relies on further partitioning existing elements of the partition, this method is particularly susceptible to the *curse of dimensionality.* That is, as the problem dimensionality grows, the search space becomes exponentially larger, and in the worst case, exponentially more branches are required to obtain small nodes and tight relaxations.

One common way to obtain LBD_i for $\min \hat{f}(\mathcal{F} \cap M^{(i)})$ or $\min \hat{f}(M^{(i)})$ in Algorithm 1 is by minimizing a suitable convex relaxation of f on $M^{(i)}$, as described in Definition 2.1. Existing implementations of B&B operate by selecting an individual element of the partition $M^{(i)}$, creating and solving a convex subproblem to obtain LBD_i, and further partitioning element $M^{(i)}$ before repeating the process. As indicated in the process laid out by Horst and Tuy [21], in principle, multiple elements of the partition $M^{(i)}$ may be selected in each iteration. However, in practice, multiple elements are not addressed simultaneously, and Steps 2 through 4 are iterated on for each i. One of the novel contributions of this paper is a B&B algorithm capable of handling multiple partition elements simultaneously by exploiting computing hardware designed to accelerate such parallelized processes.

3. Software development

One key product of this work is a GPU-compatible SCT approach to construct McCormick-based relaxations. These relaxations are then used in a novel B&B routine to *process* (i.e., solve lower- and upper-bounding problems) multiple partition elements (nodes) in parallel, which is described at the end of this section. Previous methods of creating relaxations by applying McCormick arithmetic include `MC++` [9], written in C++ and used in the MAiNGO solver [8], and `McCormick.jl` [57], written in Julia and used in the EAGO solver [59]. Both of these approaches comprise intrinsic function libraries and an OO-like scheme to construct relaxations in a functionally and mathematically equivalent way, just in different programming languages.

Similar to the ways that forward-mode AD is accomplished, it is also possible to implement the rules of McCormick via SCT. A first implementation of this new approach is currently, at the time of writing, under active development within the `SourceCodeMcCormick.jl` (`SCMC`) package, which is available publicly on GitHub [17]. As a brief summary of its operation: `SCMC` uses the Julia package `Symbolics.jl` [19] to decompose mathematical expressions, applies the generalized McCormick relaxation rules [39, 57] and interval bounding rules based on interval arithmetic and natural interval extensions [33], creates source code that computes McCormick relaxations and inclusion monotonic interval extensions associated with the original expression, and then compiles the source code into functions that return inclusion monotonic interval extensions and pointwise values of convex and concave relaxations of the original input expression. The intended use for these evaluator functions is to evaluate relaxations at a large number of points on potentially different domains within a single function call, and thereby exploit the parallel computing capabilities of GPUs.

`SCMC` relies on several features of the `Symbolics.jl` package to operate. Primarily, `Symbolics.jl` is used as a symbolic algebra system that enables McCormick relaxations to be expressed symbolically. These relaxations are then composed together to create expressions representing composite relaxations using the

Listing 1 A truncated version of the SCMC convex_evaluator function is presented. convex_evaluator accepts a mathematical expression composed of Symbolics.jl-type variables and returns a function that evaluates a convex relaxation of the input mathematical expression, as well as an ordered list of variables to inform the user as to the input format of the generated function.

```julia
function convex_evaluator(term::Symbolics.Num)
    step_1 = apply_transform(McCormickIntervalTransform(), term)
    step_2 = shrink_eqs(step_1)
    ordered_vars = pull_vars(step_2)
    @eval new_func = $(build_function(step_2[3].rhs, ordered_vars...))
    return new_func, ordered_vars
end
```

Symbolics.substitute function, and the final expressions are converted into callable functions using the Symbolics.build_function routine. Further details of how these utilities are employed by SCMC are presented in the remainder of this section.

A truncated version of convex_evaluator, one of the main user-facing utilities in SCMC, is presented in Listing 1. This utility creates an evaluator function for the convex relaxation of a math expression. The following subsections will detail the mechanism of this function to describe how an input expression is converted to an evaluator function.

3.1. Factorization and computational graph generation

The SCMC package exploits the factorability of functions of interest to construct McCormick-based relaxations. As such, we must make the practical and generally non-restricting assumption that all functions of interest are factorable. When the user inputs a mathematical expression into an evaluator generation function such as convex_evaluator shown in Listing 1, the expression is automatically factored into binary and univariate terms to generate a primal trace of the expression, from which a computational graph can be inferred. The generation of a primal trace can be seen for a representative expression in Listing 2, with the resulting primal trace and its corresponding computational graph shown in Figure 1 (left and center, respectively). Elements of the primal trace of the expression are represented in the graph as nodes, which are joined by labeled arcs that represent binary addition, multiplication, or division (i.e., inversion of the denominator multiplied by the numerator [32, 39, 57, 61]), or are individually operated on by a univariate intrinsic function. This is analogous to the auxiliary variable method of generating relaxations in that each node can be thought of as an auxiliary variable representing an intermediate term in the calculation of the original expression. However, unlike in the auxiliary variable method, when McCormick-based relaxations of the original expression are ultimately generated, these auxiliary variables are eliminated as intermediates and do not appear in the final calculation. Internally, SCMC represents the trace as a vector with elements of type Symbolics.Equation, where the left-hand sides are the auxiliary variables and the right-hand sides are the corresponding factored expressions. The left- and right-hand sides of these Equations are maintained as type SymbolicUtils.BasicSymbolic{Real}, where SymbolicUtils.jl is the foundation for Symbolics.jl.

For ease of organizing the necessary data and computed values, we define the following data structure, which is analogous to the tuples that are often defined in AD libraries (e.g., $(f(x), f'(x))$).

Listing 2 An example showing how `SCMC` can be used to create a primal trace of a mathematical expression. Note that this step is for demonstration and analysis purposes and is not necessary for standard use cases of `SCMC`.

```julia
using SourceCodeMcCormick, Symbolics
@variables x, y
expr = exp(x/y) - (x*y^2)/(y+1)
trace = factor(expr)
```

$$
\begin{aligned}
v_1 &= x\\
v_2 &= y\\
v_3 &= v_1/v_2\\
v_4 &= \exp(v_3)\\
v_5 &= v_2^2\\
v_6 &= v_1 v_5\\
v_7 &= -v_6\\
v_8 &= v_2 + 1.0\\
v_9 &= v_7/v_8\\
v_{10} &= v_4 + v_9
\end{aligned}
$$

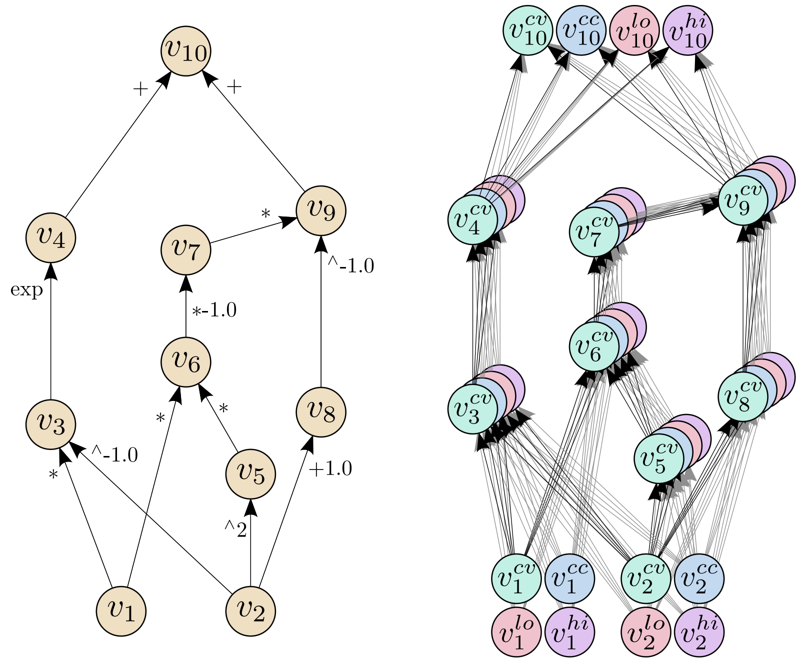

Figure 1. (left) The primal trace of the expression $\exp(x/y) - xy^2/(y+1)$, (center) the associated computational graph, and (right) the expanded computational graph. The nodes in the original computational graph are split into their McCormick tuples, creating the expanded computational graph. For nodes that were connected in the original graph, their McCormick tuples are connected in the expanded graph. Although the density of interrelationships between nodes in the expanded graph is high, the overall structure is comparable to the original computational graph.

Definition 3.1 (McCormick Tuple)**.** Consider a cumulative mapping $v_k : Z \in \mathbb{IR}^n \to \mathbb{R}$. A *McCormick tuple* will be the collection of four symbolic elements: `(vk_cv, vk_cc, vk_lo, vk_hi)` that represents $\{v_k^{cv}(\cdot), v_k^{cc}(\cdot), v_k^L(Z), v_k^U(Z)\}$. A McCormick tuple of a base variable `z` will be `(z_cv, z_cc, z_lo, z_hi)`.

Note that this definition of a McCormick tuple is very closely related to the septuple defined by Mitsos et al. [32] and the McCormick *object* defined by Ye and Scott [61] with a few exceptions. First, the septuple of Mitsos et al. [32] includes the cumulative mapping v_k and subgradient information. Subgradients are currently not included in the proposed approach, as discussed in Section 5. The McCormick objects of Ye and Scott [61] are functionally equivalent (despite differences in organization) to Definition 3.1 with the nuance that McCormick objects are defined with real-valued components (i.e., pointwise with respect to the independent variable for McCormick relaxations), whereas the McCormick tuple of Definition 3.1 contains the actual functions representing McCormick relaxations.

Listing 3 Output from the `apply_transform` function applied to the expression $x + y$. The variables x and y are extended to their McCormick tuples, and elements of the McCormick tuple of the resulting expression are labeled as `result_*`. The tilde character (˜) is used to represent equality in `Symbolics.Equations`.

```
julia> apply_transform(McCormickIntervalTransform(), x+y)
  4-element Vector{Equation}:
     result_lo ~ x_lo + y_lo
     result_hi ~ x_hi + y_hi
     result_cv ~ x_cv + y_cv
     result_cc ~ x_cc + y_cc
```

Once a function is factored (a computational graph is prepared), `SCMC` proceeds by creating a McCormick tuple of each base variable (e.g., `x`). If a base variable is being branched on in a B&B scheme, its interval bounds will be the lower and upper bounds of the B&B node for this variable, and its convex and concave relaxations will simply take the value `z` where the relaxation of the primal expression is desired (i.e., `x_cv = x_cc = z`) [39, Def. 9, Step 2]. The computational graph is then processed in a forward mode as follows. Starting from the children of the root nodes, each node is extended into its McCormick tuple by applying the McCormick rule(s) corresponding to the operation that fed into the node, with the McCormick tuple(s) of the parent node(s) as inputs. An example of the process of extending base variables and factors into their McCormick tuples is shown in Figure 1. While a basic computational graph that represents the primal trace of a function has $m - n$ nodes (not including the base variable nodes), where each has at most 2 inputs, the result of the application of McCormick rules is a larger computational graph with $4(m - n)$ nodes, each having at most 8 inputs. In this new graph, each node represents one part of its origin node's McCormick tuple.

The process described so far, including the factorization step and the application of McCormick rules, occurs in Line 2 of Listing 1. Specifically, the `apply_transform` function combines the factorization and McCormick rule application steps to output a vector with elements of type `Equation`. In this vector, the left-hand sides are variables of type `BasicSymbolic{Real}`, representing individual elements of the McCormick tuples of the auxiliary variables, and the right-hand sides are the symbolic representations of the appropriate McCormick rule(s), stored as type `BasicSymbolic{Real}`. A trivial example of this structure is displayed in Listing 3, which shows the vector of `Equations` that result from applying the `apply_transform` function to the expression $x + y$, where x and y are `Symbolics.jl` variables (wrapped in the `Symbolics.Num` type).

Following the application of McCormick rules, the computational graph is collapsed via edge contraction to eliminate auxiliary variables until only the four deepest nodes—the McCormick tuple of the original full expression—remain. This step occurs in Line 3 of Listing 1, using the utility `shrink_eqs`. The function works by selecting the first `Equation` in the input vector and replacing all instances of the `Equation`'s left-hand side with its right-hand side for all other `Equations` in the vector using the `SymbolicUtils.substitute` subroutine. That is, it replaces instances of elements of McCormick tuples of auxiliary variables with the expressions representing their associated McCormick rule(s). This process is repeated iteratively to eliminate the McCormick tuples of all auxiliary variables. The end result is a vector with four components of type `Equation`, where the left-hand sides are `Symbolics.jl` variable representations of the McCormick tuple of the original expression, and the right-hand sides are the symbolic compositions of all the McCormick rules that made up the original expression. By eliminating the auxiliary variables and composing rules starting from the base variables, these four expressions are only functions of the McCormick tuples of the base variables.

One key property of McCormick relaxations, as illustrated by Proposition 2.10 and Theorem 2.11, is that the mathematical structure of $f^{cv}(\mathbf{x})$ depends not only on the expression of f, but also on the bounds of the domain of $\mathbf{x}$ (i.e., $X = [\mathbf{x}^L, \mathbf{x}^U]$). As an aside, the development of the generalized McCormick relaxation [39] made this dependence explicit. `McCormick.jl` handles this type of bounds dependence through the use of control flow in its OO-like scheme. That is, different software functions will be called depending on logic statements based on the bounds of the input variable(s). In the novel source code generation approach of `SCMC`, this bounds dependence is accounted for using the Julia `ifelse` function. By deeply nesting these `ifelse` functions within one another, all possible mathematical structures of a relaxation that arise from composing McCormick rules together can be contained within a single code expression, where the output of the expression depends on the same type of logic statements used in typical OO schemes.

An example of how the `ifelse` function is used within `SCMC` can be seen in Listing 4. This listing shows a simplified version of the McCormick rule that `SCMC` applies for the multiplication of two terms, truncated to show only the convex relaxation part of the function. By using `ifelse` functions, this expression captures all possible realizations of the convex relaxation arising from a multiplication step within a single `Equation`. Because the input McCormick tuples are treated symbolically, during the edge contraction step, these symbolic variables can be replaced by full symbolic expressions of the McCormick tuples of the inputs thereby resulting in a more deeply nested `ifelse` expression that encodes the combined rules of multiple mathematical operations.

A key benefit of this approach is that, because `ifelse` is a function, `SCMC` can continue to work in the environment of the `Symbolics.jl` symbolic algebra system, which is important for the source code generation process, while still utilizing the control decisions necessary for building McCormick relaxations. As discussed in Section 3.4, in practice, the complexity of representable factorable functions that are represented in this manner is limited due to the rapidly increasing length of such expressions. That is, the output from the `shrink_eqs` function contains four right-hand-side terms that must capture all possible dependencies of the final McCormick tuple on the dependent variables' domains.

3.2. Source code generation

By defining and applying the McCormick rules as single-line code expressions, the deepest nodes of the expanded (and then edge-contracted) computational graph are already in a form that, if written out, could be interpreted by Julia as source code expressions that are the McCormick tuples of the original function. For example, if the right-hand sides of the four `Equation`s output by `shrink_eqs` were pasted into a separate Julia file, with McCormick tuples of the base variables assigned numerical values, the pasted expressions would return evaluations of the McCormick tuple of the original mathematical expression. However, by remaining within the `Symbolics.jl` algebra system throughout the application of McCormick-based relaxation rules, `SCMC` can further automate the step of interpreting the source code expressions and substituting in numerical values through the use of `Symbolics.build_function`.

When a symbolic expression is passed to `Symbolics.build_function`, it generates

Listing 4 A simplified and truncated version of the transform_rule for the multiplication of two symbolic variables, x and y. The apply_transform function automatically extends x and y into their McCormick tuples and then applies the appropriate transform_rule with those tuples as inputs, such as this version of the function for multiplication. This transform_rule function returns Equations where the left-hand sides (first argument) are the symbolic variables corresponding to the convex and concave relaxations of the product, and the right-hand sides (second argument) are the symbolic expressions of the applied McCormick rule.

```julia
function transform_rule(::McCormickTransform, ::typeof(*), zL, zU, zcv, zcc,
                                         xL, xU, xcv, xcc, yL, yU, ycv, ycc)
  rcv = Equation(zcv,
      ifelse(xL >= 0.0,
          ifelse(yL >= 0.0, max(-xU*yU + xU*ycv + xcv*yU,
                                -xL*yL + xL*ycv + xcv*yL),
              ifelse(yU <= 0.0, -min(xU*yU - xU*ycv - xcc*yU,
                                     xL*yL - xL*ycv - xcc*yL),
                  max(-xU*yU + xU*ycv + xcv*yU,
                      -xL*yL + xL*ycv + xcc*yL))),
          ifelse(xU <= 0.0,
              ifelse(yL >= 0.0, -min(xL*yL - xL*ycc - xcv*yL,
                                     xU*yU - xU*ycc - xcv*yU),
                  ifelse(yU <= 0.0, max(-xL*yL + xL*ycc + xcc*yL,
                                        -xU*yU + xU*ycc + xcc*yU),
                      -min(xL*yL - xL*ycc - xcc*yL,
                           xU*yU - xU*ycc - xcv*yU))),
              ifelse(yL >= 0.0, max(-xU*yU + xU*ycv + xcv*yU,
                                    -xL*yL + xL*ycc + xcv*yL),
                  ifelse(yU <= 0.0, -min(xL*yL - xL*ycc - xcc*yL,
                                         xU*yU - xU*ycv - xcc*yU),
                      max(-xU*yU + xU*ycv + xcv*yU,
                          -xL*yL + xL*ycc + xcc*yL)))))
  [...]
```

code for a numeric function that allows the substitution of numerical input values in place of the BasicSymbolic{Real}-typed variables. build_function is used in Line 5 of Listing 1, where the third element of the vector output from shrink_eqs (i.e., the term representing the convex relaxation of the original expression) is passed as an argument, along with a sorted list of BasicSymbolic{Real}-typed variables used in the relaxation, generated in Line 4 using the SourceCodeMcCormick.pull_vars function. The code output from build_function is then wrapped in an @eval statement to compile the code into a callable Julia function. Although not utilized by SCMC, build_function is also capable of building functions compatible with other languages such as C, Stan, and MATLAB [19]. This function, and the ordered list of BasicSymbolic{Real}-typed variables to use as a reference, are the outputs of the convex_evaluator function in Listing 1. An additional user-facing function, all_evaluators, returns four functions representing the elements of the McCormick tuple as well as the ordered variable list. In sum, because the McCormick rules were applied within the Symbolics.jl algebra system, we are allowed to use build_function to construct static evaluation functions that return inclusion monotonic interval extensions and convex and concave relaxation values of the original function at respective input values.

One important distinction between using build_function and simply pasting the top-level node expressions into a separate file is the fate of the ifelse function. Although this function is useful for remaining in the Symbolics.jl algebra system, if used as-is, the resulting code will perform very poorly. The ifelse function works by evaluating both conditional paths and then returning the appropriate result based on the conditional statement. This operation would waste significant computing time, particularly because the nature of McCormick-based relaxations means that there would be many potential conditional branches. The `build_function` utility, however, internally substitutes `ifelse` with normal if-else control flow, which only evaluates the branch corresponding to the conditionally selected outcome. This type of substitution does not occur for functions such as `min` or `max`, for which both arguments must be evaluated and compared to return the correct result.

The distinction between `ifelse` and `min` or `max` can be seen more clearly by referring again to Listing 4. By Proposition 2.10, the relaxations of a product are comprised of `min`/`max` functions for the calculations of both intermediate terms and the relaxations themselves. If these rules were implemented using only `min` and `max`, every multiplication operation in the intermediate terms would need to be evaluated prior to performing the relaxation evaluations. Instead, we recognize that the decisions in the intermediate functions can be simplified to checks on the bounds of g_1 and g_2. For example, for α_1, since $g_1^{cv}(\mathbf{z}) \leq g_1^{cc}(\mathbf{z})$ by definition, the correct output can be ascertained by checking if $g_2^L \geq 0$. In this case, replacing `min` with `ifelse` eliminates the need to perform both multiplication operations of $g_2^L g_1^{cv}(\mathbf{z})$ and $g_2^L g_1^{cc}(\mathbf{z})$, and also eliminates the need to evaluate a relaxation of g_1 that is not ultimately chosen, which may be expensive if the expression is deeply nested. A similar substitution of `min`/`max` for `ifelse` cannot easily be made for the calculation of g^{cv} or g^{cc} since a simpler conditional check is not readily available. Thus, the implementation of multiplication in `SCMC`, as shown in Listing 4, proceeds by checking the bounds of the factors using `ifelse` conditional checks with each branch terminating in a `min` or `max` function with the correct results of the intermediate functions interpolated. It is worth noting that, by this construction, the only code branches required for multiplication are those that check the signs of the bounds of the factors—`min` and `max` do not result in branches in the code since both arguments are always evaluated to determine the output, regardless of the result.

It is also important to address the complexity of these source-code-generated functions. The aim of `SCMC` is to create single compiled functions that produce McCormick relaxations on any interval domain, and, by design, the mathematical structure of McCormick relaxations depends on the domain. Therefore, any function that tries to encapsulate all possible domains will necessarily have many different potential calculations to perform, separated in `SCMC`-generated functions by `if-else` statements. Table 1 shows the number of conditional branches contained in convex relaxations for a set of common mathematical expressions. There are more branches than might be expected from a naïve implementation of the rules in, e.g., Definition 2.10 due to an added intersection between relaxations and the inclusion monotonic interval extensions. This operation is realized in [39, Def. 9, Step 6] and referred to as the *cut* operation, which is formalized in Ye and Scott [61, Def. 11]. Due to how `SCMC` composes McCormick-based relaxations together, the total number of conditional branches grows rapidly, even for visually simple expressions. Although the compiled versions of these functions return evaluations quickly (see Subsection 3.4), combining more than a few variables together, such as for multiplication or division, results in lengthy expressions that can consume significant memory resources of the computing system during compilation. If expressions that are too complicated are passed through `SCMC`, it may cause Julia to stall in the function creation step. For this reason, a warning system is implemented in `SCMC` that attempts to determine a priori whether an expression will be too long to handle in a reasonable time frame. If so, `SCMC` will abort the calculation and inform the user.

Table 1. The total numbers of conditional branches that are required to express a convex relaxation equation using the SCT approach are reported for commonly encountered multi-term/multi-factor equation forms (e.g, sums and products). Cases involving summations exhibit a number of conditional branches that are independent of the number of terms. Cases involving products (and divisions) exhibit a combinatorial dependence of the number of conditional branches on the number of factors in the operation. Dashes in the table indicate expressions that were too long to evaluate.

Equation	Cardinality of Terms/Factors			
Form	**2**	**3**	**4**	**5**
$\sum x_i$	0	0	0	0
$\exp(\sum x_i)$	6	6	6	6
$(\sum x_i)^2$	6	6	6	6
$\prod x_i$	16	736	20,608	—
$\exp(\prod x_i)$	262	10,390	281,014	—
$(\prod x_i)^2$	310	11,206	294,118	—
$x_1/(\prod_{i=2}^{m} x_i)$	314	32,426	321,578	—

3.3. Utilities

In this section, details are provided on how to use two main `SCMC` utilities to evaluate the bounds and relaxations of mathematical expressions of interest: `convex_evaluator` and `all_evaluators`.

The `SCMC` function `convex_evaluator` returns a source-code-generated function that provides evaluations of convex relaxations of the mathematical expression of interest. This utility is designed for situations where only convex relaxations of an expression are desired, which may be the case with relatively simple constraints or an objective function in an optimization problem. Listing 5 provides an example of how `convex_evaluator` can be used to generate an evaluator function, as well as how the generated function can be used. Specifically, the generated function takes numerical values corresponding to the McCormick tuples of the dependent variables, grouped by tuple and sorted alphabetically according to the `var_order` output, and returns a convex relaxation of the original expression on the desired domain, evaluated at the desired point in that domain. Similarly, the generated function can be broadcast over vectors of numerical values to return evaluations of a convex relaxation of the original expression at multiple locations, with no requirement that the domains of the dependent variables be similar for different elements of the input vectors.

The function `all_evaluators` returns four functions corresponding to the McCormick tuple of the input expression, i.e. a convex relaxation, a concave relaxation, and the lower and upper bounds of an inclusion monotonic interval extension. Having access to all elements of the McCormick tuple may be useful for analysis purposes, and when dealing with cumulative mappings such as when the original expression is too complicated to handle in a single `convex_evaluator` call due to memory limitations. Listing 6 shows an example of how `all_evaluators` can be used to calculate relaxations for a complicated expression that could not otherwise be handled in one step. This approach exploits the notion that, when a computational graph is contracted, we are applying the concepts of composite relaxations (Def. 2.6) and cumulative mappings (Def. 2.5). Namely, observe that all nodes with index $k > n$ (i.e., factors) only rely on the McCormick tuples corresponding to the connected nodes immediately prior in the graph. For example, in Figure 1, the source nodes correspond to factors v_1 and v_2,

Listing 5 An example is provided to demonstrate how the `convex_evaluator` function can be used to obtain convex relaxations of a desired expression. The variables in the `var_order` vector, with the McCormick tuples sorted alphabetically, are shown in the order in which they should be passed into the evaluator function. Note that the created `expr_cv_eval` function may also be broadcast over vectors of values to obtain relaxations of multiple points simultaneously.

```
using SourceCodeMcCormick, Symbolics
@variables x, y
expr = exp(x/y) - (x*y^2)/(y+1)
expr_cv_eval, var_order = convex_evaluator(expr)
xcv, xcc, xlo, xhi = 1.0, 1.0, 0.5, 3.0
ycv, ycc, ylo, yhi = 0.7, 0.7, 0.1, 2.0

julia> @show var_order;
  var_order = Num[x_cv, x_cc, x_lo, x_hi, y_cv, y_cc, y_lo, y_hi]

julia> convex_relaxation = expr_cv_eval(xcv, xcc, xlo, xhi, ycv, ycc, ylo, yhi)
  0.22836802303235837
```

which were defined to be the independent (base) variables. However, for more complicated expressions, v_1 and v_2 could instead represent cumulative mappings, with their corresponding relaxations in the McCormick tuple calculated as composite relaxations. Although the process shown in Listing 6 currently requires user-defined functions, future work will explore the automation of this step so that complicated expressions can be passed seamlessly to `SCMC` without any external combination steps such as described.

3.4. GPU compatibility

A key design objective of `SCMC` was compatibility with GPGPU. This was a goal primarily due to the potential for substantial speed benefits if the software can exploit the massive parallelization of GPUs, but also because a successful implementation would allow the speed of the associated global optimization algorithm to scale based on available hardware resources rather than CPU clock speed. An example is presented in Listing 7 to illustrate how the generated `SCMC` functions can be used to perform evaluations of relaxations on a GPU. Thanks to the development of `CUDA.jl` by Besard et al. [5] (a Julia wrapper for the CUDA language [27] for GPGPU on NVIDIA GPUs), it is simple to accelerate these functions by broadcasting them over CUDA arrays instead of standard arrays. `SCMC` functions are designed to only employ functions internally that are compatible with GPGPU, and thus no additional considerations are required in terms of problem formulation or use of `SCMC` function generators.

Benchmarking results are presented in Table 2 for convex evaluator functions created for a set of commonly encountered multi-term equation forms. Given that the intent of `SCMC` is to make use of massive parallelization, the benchmark times in this table represent the time to calculate one million relaxation values for randomly selected points from the variable domain of $X = [0.9, 1.1]^n$. Three times are shown for each cell, representing the time to perform the calculations using the multiple dispatch approach of `McCormick.jl`, the time using a `SCMC`-generated function using one core of an Intel Xeon W-2195 CPU (2.3 GHz/4.3 GHz base/turbo), and the time using a `SCMC`-generated function using an NVIDIA Quadro GV100 GPU. Due to memory limitations, evaluator functions were only generated for expressions with fewer than 10^5 conditional branches (see Table 1). It should also be noted that the `McCormick.jl`

Listing 6 An example of how an expression that is too complicated for `SCMC` to handle in a single function evaluation can be separated into intermediate expressions and recombined with the use of the `all_evaluators` function in user-defined code. Arbitrarily complicated expressions can be handled in this manner, exploiting the notions of composite relaxations (Def. 2.6) and cumulative mappings (Def. 2.5).

```julia
# Expression: f = (x*y*z - 3*(x*y) + 4*(z/y)) / (x^2 + y^2 + z^2)
using SourceCodeMcCormick, Symbolics

@variables x, y, z, temp1, temp2

num_cv, num_cc, num_lo, num_hi, _ = all_evaluators(x*y*z - 3*(x*y) + 4*(z/y))
den_cv, den_cc, den_lo, den_hi, _ = all_evaluators(x^2 + y^2 + z^2)
f_cv, _ = convex_evaluator(temp1/temp2)

function div_cv(xcv, xcc, xlo, xhi, ycv, ycc, ylo, yhi, zcv, zcc, zlo, zhi)
  input = [xcv, xcc, xlo, xhi, ycv, ycc, ylo, yhi, zcv, zcc, zlo, zhi]
  temp1_cv = num_cv.(input...)
  temp1_cc = num_cc.(input...)
  temp1_lo = num_lo.(input...)
  temp1_hi = num_hi.(input...)
  temp2_cv = den_cv.(input...)
  temp2_cc = den_cc.(input...)
  temp2_lo = den_lo.(input...)
  temp2_hi = den_hi.(input...)
  division_cv = f_cv.(temp1_cv, temp1_cc, temp1_lo, temp1_hi,
                      temp2_cv, temp2_cc, temp2_lo, temp2_hi)
  return division_cv
end
```

Listing 7 An example of how relaxation evaluator functions can be used with CUDA arrays to calculate relaxations on a GPU. In this example, the domain of x is left unchanged for all points for simplicity, but this is not a requirement. Points on any domain may be passed within the same function evaluation, all of which will be evaluated in parallel. Note also that double-precision floating-point numbers must be specified, as CUDA arrays will default to single-precision floating-point values.

```julia
  using SourceCodeMcCormick, Symbolics, CUDA
  @variables x
  expr = (x-0.5)^2+x
  f_cv, order = convex_evaluator(expr)

  xcv_GPU = CUDA.rand(Float64, 1000000)
  xcc_GPU = copy(xcv_GPU)
  xlo_GPU = CUDA.zeros(Float64, 1000000)
  xhi_GPU = CUDA.ones(Float64, 1000000)
  convex_relaxations = f_cv.(xcv_GPU, xcc_GPU, xlo_GPU, xhi_GPU)
```

method, by using the MC type and multiple dispatch, simultaneously calculates the following for the desired expression: convex and concave relaxations, inclusion monotonic interval extensions, and corresponding subgradients. Notably, the SCMC-generated functions evaluated using a CPU are considerably faster than the McCormick.jl approach in all cases, by a factor of 1.1–25.4x. Using a GPU, however, the SCMC functions are consistently nearly three orders of magnitude faster than the McCormick.jl approach. This result showcases the speed that can be gained by utilizing a parallel computing approach.

Table 2. The total times to evaluate convex relaxations of the equation of the listed form with one million randomly selected points in the domain $X = [0.9, 1.1]^n$ are tabulated for three different methods: using the McCormick.jl approach (MC), the SCMC approach operating on one core of an Intel Xeon W-2195 CPU (SCMC (CPU)), and the SCMC approach on an NVIDIA Quadro GV100 GPU (SCMC (GPU)). The SCMC method, when performed on the CPU, calculates relaxations considerably faster than the McCormick.jl method in every case. When the SCMC approach is paired with a GPU, the evaluation times are nearly three orders of magnitude shorter than the McCormick.jl method. Note that the McCormick.jl method, by design, is also computing a concave relaxation, inclusion monotonic interval extension, and subgradients of the relaxations.

Equation	McCormick		Cardinality of Terms/Factors			
Form	Style	Unit	**2**	**3**	**4**	**5**
	MC	ms	15.923	28.457	42.387	61.937
$\sum x_i$	SCMC (CPU)	ms	0.851	1.292	1.861	2.434
	SCMC (GPU)	ms	0.049	0.063	0.076	0.091
	MC	ms	79.753	92.130	99.210	117.020
$\exp(\sum x_i)$	SCMC (CPU)	ms	6.817	9.528	12.755	30.842
	SCMC (GPU)	ms	0.082	0.113	0.142	0.171
	MC	ms	53.276	67.735	76.703	98.066
$(\sum x_i)^2$	SCMC (CPU)	ms	9.941	14.412	22.479	28.354
	SCMC (GPU)	ms	0.109	0.152	0.194	0.236
	MC	ms	33.154	65.497	93.542	127.982
$\prod x_i$	SCMC (CPU)	ms	7.666	21.990	77.212	—
	SCMC (GPU)	ms	0.110	0.154	0.211	—
	MC	ms	97.459	132.791	161.305	201.233
$\exp(\prod x_i)$	SCMC (CPU)	ms	25.075	90.600	—	—
	SCMC (GPU)	ms	0.113	0.209	—	—
	MC	ms	77.450	109.112	141.369	179.004
$(\prod x_i)^2$	SCMC (CPU)	ms	26.583	96.794	—	—
	SCMC (GPU)	ms	0.113	0.216	—	—
	MC	ms	83.475	121.933	154.147	198.200
$x_1/(\prod_{i=2}^{m} x_i)$	SCMC (CPU)	ms	9.837	41.361	—	—
	SCMC (GPU)	ms	0.113	0.167	—	—

Designing an algorithm to be GPU compatible requires acknowledgment of the limitations of GPU architectures. While CPUs are built with a small number of fully independent cores that can execute independent tasks simultaneously, GPUs are designed with a much larger number of cores that, in general, operate in parallel by following a single instruction that is applied to multiple data points (SIMD). For tasks where the same instructions are executed a large number of times, SIMD operation can be significantly faster than operation on x86 hardware, by virtue of the greater number of cores in typical GPUs as compared to typical CPUs. This concept is illustrated in Figure 2 for the evaluation of a convex relaxation $f^{cv} : X \to \mathbb{R}$ at several different points on a single core of a CPU versus a GPU. However, if the instructions differ depending on the data, there can be significant time penalties that impact the overall computation time.

Consider, for example, GPUs produced by NVIDIA. To execute a GPU function—called a kernel—on a collection of data points, the task will be handled by a collection of threads, which can be thought of as individual execution units. NVIDIA GPUs issue

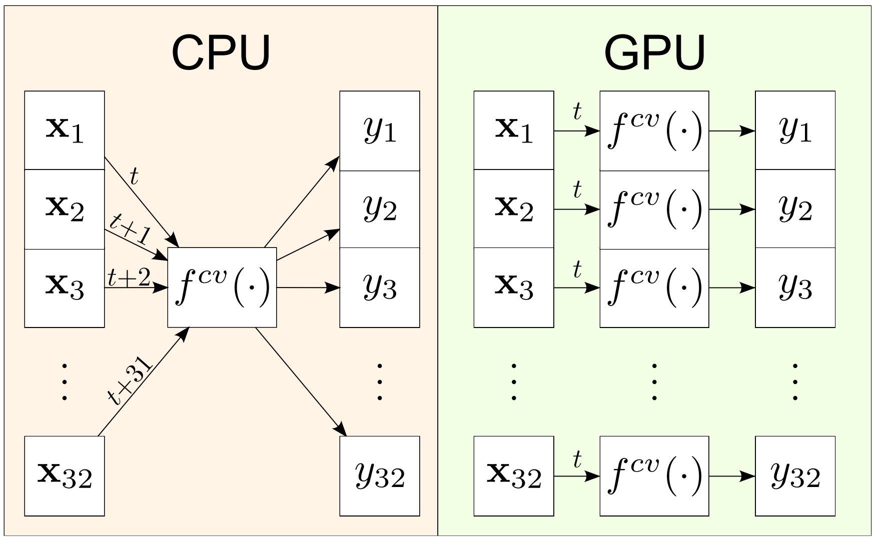

Figure 2. This figure illustrates a high-level simplification of the computing differences between CPU and GPU architectures. A convex relaxation $f^{cv}: X \to \mathbb{R}$ is evaluated at 32 points with reference to an instruction set sequence time index t. A CPU executes the instruction sets on the data points sequentially (ignoring multithreading). In contrast, the architecture of the GPU enables instruction sets to be executed on many data points simultaneously. 32 points $\mathbf{x}_i \in X$ and function evaluations are illustrated to coincide with a single warp on the GPU.

instructions to *warps*, which are groups of 32 threads, as illustrated in Figure 2. In the ideal case, where the same instruction applies to every data point, each thread in a warp will operate in parallel to compute the total of 32 results. However, if the instructions differ between threads in a warp—a phenomenon called warp divergence—the GPU will be unable to issue the different instructions to different threads simultaneously. One way GPUs can handle warp divergence is by first activating only those threads in the warp that will execute one set of instructions, then activating the threads that will execute an alternate set of instructions, and so on. This is inherently slower than the ideal case because the massive parallelization made possible by the GPU architecture is not being used to the fullest extent. In the worst case, where each thread in a warp executes a different instruction, the total calculation time may increase by a factor of approximately 32. On the other hand, if there would be significant warp divergence but the data points can be sorted prior to calculation to minimize the number of unique instructions per warp, the computation time can be greatly reduced.

This architecture limitation is critical to understand and account for when using `SCMC`, since the relaxation evaluator functions are composed of deeply nested `if-else` statements. Particularly, if worst-case GPU performance comes from warp divergence resulting in 32 unique instructions per warp, and the `SCMC` functions have hundreds to tens of thousands of unique branches, intuitively these functions should practically always result in very poor computational performance. Fortunately, there are three important factors that work to the benefit of `SCMC`:

(1) The number of conditional branches is large because every possible control-flow case must be considered. However, conditional branches are not equally likely to be encountered in practice. For example, 57 paths from the 314 conditional branches in the expression representing a convex relaxation of x/y stem from checks that the division operation is defined on the given domain. If the user is working on variable domains where division always returns a real value, these

conditional branches will never be encountered.

(2) The substitution process of `SCMC` results in nested `ifelse` statements but does not currently check for conflicting conditional statements within the tree. Consequently, many conditional branches may be unreachable.

(3) Functions such as x/y fall victim to warp divergence and performance degradation in the case where threads in the same warp try to access different conditional branches. For McCormick relaxations, different conditional branches might be accessed when inputs or intermediate values switch from positive to negative, or vice versa. In a typical use case of B&B, which is the expected application space of `SCMC`, the SCT-generated functions will likely be used to evaluate multiple points within individual B&B nodes, which in most cases represent comparatively small regions of the overall search space. Within a single node, it is unlikely that multiple conditional branches will need to be accessed, since the points to evaluate will be numerically similar. If points from multiple nodes are to be evaluated and the nodes are far from one another, it is possible that different conditional branches will need to be accessed for points associated with the two nodes, but this is far from the worst-case warp divergence scenario of 32 unique branch paths. This point is further helped by the problem scale. As detailed in Subsection 3.5, the current application method requires the evaluation of $2n+1$ points per node, where n is the problem dimensionality. For problems involving higher dimensions, larger numbers of numerically similar points will be grouped together, which will further reduce the incidence of warp divergence.

In summary, although these functions have a large number of conditional branches, the existence of these conditional branches is not necessarily problematic. The key factor is warp divergence, which is likely to be low in practice within typical optimization problems that will be solved using B&B.

One critical aspect to address is how the speed of the evaluator functions is affected by warp divergence and how the functions can be expected to perform in a spatial B&B algorithm. To address the large potential impact of warp divergence, the speeds of `SCMC` convex evaluator functions for several forms of mathematical expressions are presented in Table 3 for a set of potential computational scenarios. These range from the most ideal scenario, in which all the input points and domains are numerically similar—such as the case in Table 2—to the worst-case scenario, where the input points and domains are numerically dissimilar and randomized. There are additional intermediate cases meant to represent how these functions may be expected to behave in a B&B implementation. Notably, in optimization problems where the branching variables are constrained to numerically similar values. For example, positive values over a range that is not expected to change the sign of the resulting calculation, the `SCMC` functions should operate at speeds close to the ideal case, since there is expected to be low warp divergence. This is also true for multiplication, which is implemented to only have warp divergence when the signs of the operands change (see discussion in Section 3.2). In more complicated cases, such as optimization problems where the variables are all close to zero and can take on positive or negative values, the `SCMC` functions perform slower than the ideal case, but still substantially faster than the worst case. Generally, these results indicate that although warp divergence is a critical factor affecting performance, the degree of warp divergence is likely to be low in typical problems, and thus the generated functions should perform similarly to their ideal speeds in practice.

Table 3. The total times to evaluate convex relaxations of the equation of the listed form with $\mathbf{x} \in X \in \mathbb{IR}^3$ are tabulated. These timings are for 350k evaluation points—equivalent to 50k B&B nodes using the blackbox sampling method of Song et al. [43]—using `SCMC` source-code-generated functions on a GV100 GPU. The Ideal Case represents a potentially unrealistic scenario with numerically similar inputs, where all evaluation points are randomly selected from a small, identical domain of $X = [0.9, 1.1]^3$. The Typical B&B Case represents a commonly encountered B&B scenario in which we evaluate 7 points per node and the 50k nodes are randomly selected non-overlapping subdomains of $X = [0, 3]^3$. The Complicated B&B Case represents a B&B scenario where we evaluate 7 points per node and the 50k nodes are randomly selected non-overlapping subdomains of $X = [-1.5, 1.5]^3$ (i.e., spanning negative and positive values). In the two B&B cases, the order of the nodes is randomized to emulate a more realistic B&B situation. The Worst Case represents a situation where we evaluate 1 point per node and the 350k nodes are randomly selected (possibly overlapping) subdomains of $X = [-1.5, 1.5]^3$. In the complicated B&B and worst case scenarios, the $\exp(\Pi x_i)$, $(\prod x_i)^2$, and division formulations exhibit significant time penalties from warp divergence.

Equation Form	Unit	Ideal Case	Typical B&B Case	Complicated B&B Case	Worst Case
$\sum x_i$	ms	0.029	0.029	0.029	0.029
$\exp(\sum x_i)$	ms	0.047	0.047	0.047	0.047
$(\sum x_i)^2$	ms	0.060	0.060	0.060	0.060
$\prod x_i$	ms	0.063	0.063	0.239	0.713
$\exp(\prod x_i)$	ms	0.093	0.094	3.801	12.665
$(\prod x_i)^2$	ms	0.097	0.099	3.696	15.776
$x_1/(\prod_{i=2}^{m} x_i)$	ms	0.072	0.090	1.823	2.078

3.5. Global optimization with SourceCodeMcCormick.jl

So far, we have focused on the details of how the `SCMC` approach can be used to quickly obtain a large number of pointwise evaluations of convex relaxations of mathematical expressions. In this section, we develop a novel parallelized B&B framework that can exploit these fast pointwise evaluations of relaxations in parallel. Since this parallel pointwise evaluation approach does not align naturally with how global optimizers like EAGO normally function, careful considerations are required to effectively exploit this approach. Specifically, EAGO and global optimizers like EAGO most commonly assume that only one B&B node is evaluated at a time. Since the `SCMC` approach makes the best use of parallelization when a large number of points are evaluated, and because it is capable of evaluating different variable domains (nodes) simultaneously, a B&B method that uses `SCMC` should have the capability to solve lower- and upper-bounding problems for multiple B&B nodes in parallel rather than in series. Additionally, modern global optimization routines that utilize McCormick-based relaxations also typically use subgradients of their corresponding convex/concave relaxations for domain reduction and accelerating lower-bounding problems, which are currently not computed by `SCMC`. Although the inclusion of subgradients could be beneficial, solvers like EAGO typically use them to generate subtangent hyperplanes used for LP lower-bounding problems, which are then sent to a separate LP solver to obtain a lower bound. If subgradients are to be incorporated into `SCMC`, additional developments are necessary to obtain batched solutions to large numbers of LPs on a GPU so that this process does not become the bottleneck for parallelization. This is a future area of research.

Instead, for the choice of lower-bounding method, recent work of Song et al. [43] provided a method for calculating a mathematically rigorous lower bound of a convex function via pointwise evaluations. This method starts by considering an interval $X \in \mathbb{IR}^n$ and a convex function $f : X \to \mathbb{R}$. We define the midpoint $\mathbf{w}^{(0)}$ of X, and for each index $i \in \{1, \dots, n\}$, we define two vectors $\mathbf{w}^{(\pm i)}$, as follows:

$$\begin{aligned}\mathbf{w}^{(0)} &:= \frac{1}{2}(\mathbf{x}^L + \mathbf{x}^U),\\ \mathbf{w}^{(\pm i)} &:= \mathbf{w}^{(0)} \pm \frac{\alpha_i}{2}(x_i^U - x_i^L)\mathbf{e}^{(i)},\end{aligned}$$

with $\mathbf{e}^{(i)}$ as the i-th unit coordinate vector in $\mathbb{R}^n$ and with a predetermined step length $\alpha_i \in (0, 1]$. We continue by obtaining the values of the convex function at each of these points (i.e., $y_0 := f(\mathbf{w}^{(0)})$ and $y_{\pm i} := f(\mathbf{w}^{(\pm i)})$) and, using the evaluations at these points, we calculate a scalar lower bound as:

$$f^{LBD} := y_0 - \sum_{i=1}^{n} \left(\frac{\max(y_{+i}, y_{-i}) - y_0}{\alpha_i} \right).$$

The full details of the proof may be found in Song et al. [43].

This method is especially convenient for the `SCMC` approach because a known number of pointwise evaluations $(2n + 1)$ is required regardless of the complexity of the underlying convex function. Furthermore, since the locations of these points can be determined a priori based on the domain of a B&B node, it is simple to determine which points the convex relaxations must be evaluated at for multiple nodes before any calculations are completed. In effect, by applying this method, the points and domains that will need to be evaluated can be gathered prior to calculation, and then all the pointwise evaluations that are needed can be calculated in a single step for an arbitrary number of nodes, limited mainly by VRAM capacity.

To address the new parallel node evaluation paradigm, an extension of the EAGO solver was created to handle multiple nodes simultaneously. Pseudocode showing the overall algorithm is provided by Algorithm 2, which is also illustrated in Figure 3. An important distinction between this algorithm and that of the default EAGO solver is how upper-bounding problems are handled. In the default EAGO B&B method, the upper-bounding problem is solved for every node shallower in the tree than a predetermined threshold, and then it is solved depending on a depth-based probability distribution for deeper nodes. This approach works well when nodes are individually evaluated, as the number of nodes after several iterations is still low, on the order of 10^1. With this small number of nodes, solvers such as IPOPT [53], which may take $10^1 - 10^2$ ms to run, are only called a small number of times. In Algorithm 2, by the time the first iteration begins, the problem domain has already been partitioned into $n_\sigma = q$ subdomains/nodes by the preprocessing step. Depending on the machine and user settings, q may be large: preliminary testing of some of the numerical examples in Section 4 found that $q = 2^{13} = 8192$ worked well. Although IPOPT is a highly performant solver, running it on all $n_\sigma = q$ nodes would be detrimental to the speed of the overall algorithm, and choosing a subset of the n_σ nodes on which to use IPOPT may be no better than random guessing. For this reason, IPOPT is only used once, at the root node before the initial partitioning step, to obtain a reasonable starting upper bound for global optimization. Instead, for each iteration of the main algorithm, we note that pointwise evaluations of the objective function can indeed function as valid upper bounds. The main power of `SCMC` lies in its speed of performing parallelized pointwise evaluations, so the addition of one point per node to function as a valid upper bound is computationally inexpensive.

Although the lower bounds generated for each individual node could be made tighter

Algorithm 2 Parallel-Evaluation B&B (ParBB)

```
(1) Solve the original program to local optimality at the root node on the domain
    $X \in \mathbb{IR}^n$ using IPOPT (or some other NLP solver)
(2) Partition $X$ into $n_\sigma := q$ similar-sized intervals $X^{(i)}$ such that $X = \cup_{i=1}^{q} X^{(i)}$,
    and push them to the main B&B stack
(3) While the problem is not converged (or the stack is nonempty):
    (a) Perform fathoming (pop and delete $d$ nodes from the stack if their lower
        bounds preclude them from containing a problem solution). $n_\sigma := n_\sigma - d$
    (b) Pop the $n_\sigma^{\text{sub}} := \min\{n_\sigma, q\}$ nodes with the lowest lower bounds in the main
        B&B stack to the parallel evaluation substack
    (c) Perform in parallel, for each node in the substack:
        (i) Calculate the values of the $2n+1$ points to be evaluated and add them
            to the evaluation queue
        (ii) Calculate the node's midpoint and add it to the evaluation queue; eval-
             uating the objective function at this point returns an upper bound
        (iii) Obtain convex relaxation values and a pointwise evaluation of the ob-
              jective function by passing the evaluation queue to the SCMC convex
              evaluator for the objective function
        (iv) Apply the blackbox sampling method to calculate the node lower bound
             based on the $2n+1$ evaluation points
        (v) Apply the lower bound result to the node in the substack
        (vi) Apply the upper bound result to the node in the substack
    (d) Branch each node in the substack and push all new partitions to the main
        B&B stack. $n_\sigma := n_\sigma + n_\sigma^{\text{sub}}$
    (e) Check for problem convergence
```

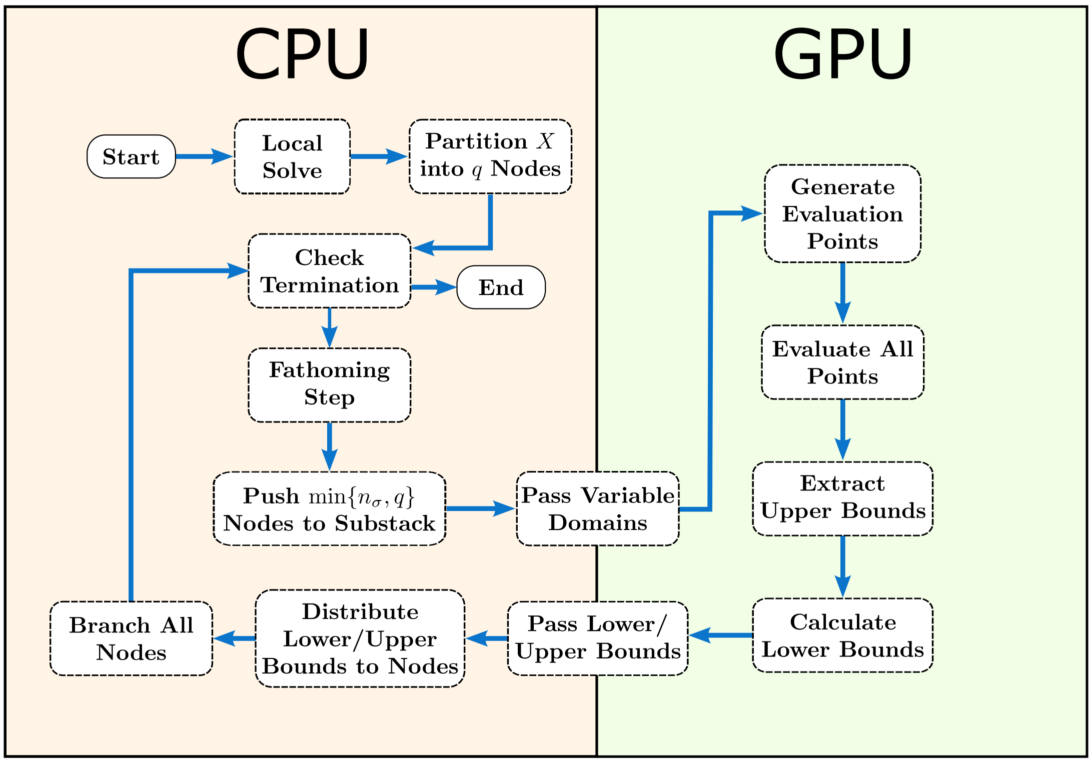

Figure 3. A block flow diagram representation of the parallel B&B algorithm (Algorithm 2), indicating the hardware on which the algorithm steps are executed. Operations corresponding to node storage and management are kept on the CPU, whereas numerical operations are offloaded to the GPU. Steps involving the transfer of information between the CPU and GPU are located at the intersection of the CPU and GPU regions.

by utilizing subgradient information, this is compensated for by the much greater node throughput and the consequently smaller subdomains that these calculations are being performed on for each iteration of the algorithm. One important consequence of this algorithm, however, is related to the large number of nodes generated in the main B&B stack. While the base implementation of B&B would require 10^5 iterations to reach a stack size of 10^5 nodes (without fathoming), with $n_\sigma = q$ set to 2^{13}, this number of nodes is reached after just twelve iterations of Algorithm 2. Storing such a large number of nodes may be highly memory intensive. This issue is exacerbated by the decreased tightness associated with each node due to the choice of lower-bounding method, which results in fewer nodes being fathomed than if the same nodes were evaluated using a tighter method. Consequently, while many more nodes can be processed using this algorithm than the more typical single-node implementations, the abundant use of memory quickly becomes a limiting factor in the overall algorithm speed. However, if subgradient calculations can be incorporated into the `SCMC` functionality in the future, this issue may be ameliorated to a large extent.

4. Numerical examples

While the parallel B&B algorithm (ParBB), Algorithm 2, may be applied to a range of optimization problems, in this section we will explore the application of the new methods and software to the optimization of a machine learning model (Sec. 4.1) and parameter estimation problems (Sec. 4.2–4.3). The machine learning problem of Section 4.1 seeks to determine optimal inputs to a trained artificial neural network (ANN), formulated generally as

$$\min_{\mathbf{x}\in X\in\mathbb{IR}^n} f^{\text{ANN}}(\mathbf{x}), \tag{1}$$

where f^{ANN} represents a scalar output of the ANN that relates to some system performance metric. The parameter estimation problems of interest are formulated generally as:

$$\mathbf{p}^* \in \arg\min_{\mathbf{p}\in P\subset\mathbb{R}^{n_p}} \sum_{i=1}^{n_d} (y_i(\mathbf{p}) - y_i^{\text{data}})^2 \tag{2}$$

The objective function in (2) is the sum of squared error (SSE) between the predictions of a model of interest $\mathbf{y}(\mathbf{p})$ and a set of experimental data $\mathbf{y}^{\text{data}}$, where $\mathbf{p}$ is the uncertain parameter vector and n_d is the number of experimental data points. In the following sections, two problems of this form will be used for numerical experiments to benchmark and demonstrate the use of the proposed approaches.

An important point to note is that the ParBB algorithm is, at the time of writing, an early-stage implementation to demonstrate the utility of `SCMC` in a GPU-based deterministic global optimization algorithm. ParBB by itself is not yet a mature solver and lacks bounds tightening techniques and features that are common across existing solvers, such as constraint propagation, feasibility-based bounds tightening, and optimization-based bounds tightening, among others. For this reason, in order to showcase a more direct comparison between how ParBB and the other solvers handle the following numerical examples, the other solvers (BARON [38], ANTIGONE [31], SCIP [7, 52], and EAGO [59]) will have their pre- and post-processing techniques deactivated, using the settings presented in Table 4. These versions of the solvers, which rely only on their branching and bounding routines, are referred to in this section as "vanilla" versions of these solvers. For each of the examples, reactivating these processes greatly enhances the ability of the solvers to address the problems. Additionally, to better demonstrate the differences between subgradient-based methods and subgradient-free methods, such as what is used in the ParBB algorithm, a version of EAGO is included in the examples that uses the same "vanilla" settings as previously described in addition to the blackbox sampling method of Song et al. [43] as the lower-bounding method.

All numerical experiments in this work were run on a single thread of an Intel Xeon W-2195 2.30/4.30 GHz (base/turbo) processor with 64 GB of RAM running a Windows 11 Enterprise 22H2 operating system and an NVIDIA Quadro GV100 GPU with 32 GB of HBM2 VRAM (driver v552.22, CUDA v12.4). Julia v1.10.3 was used in conjunction with `BenchmarkTools.jl` v1.5.0 [10], `CUDA.jl` v5.3.3 [5], `EAGO.jl` v0.8.1 [59], `JuMP.jl` v1.21.1 [13], `McCormick.jl` v0.13.4 [57], `SourceCodeMcCormick.jl` v0.3.1, and `Symbolics.jl` v5.28.0 [19]. Vanilla EAGO is run using GLPK as the lower-bounding solver, using `GLPK.jl` v1.2.0 which is a wrapper for GLPK solver v5.0.1 [29]. Each problem was run with a time limit of two hours. For the ParBB algorithm and lower-bounding method, q was set to 2^{13} and α was set to 2×10^{-5} for each problem.

Table 4. Parameters used to obtain "vanilla" versions of common deterministic global optimizers. These versions are meant to represent B&B approaches that do not make use of pre-processing or post-processing techniques, relying instead on lower- and upper-bounding techniques within each B&B node.

Solver	Parameters

BARON (Vanilla)	`LBTTDo = 0`
	`MDo = 0`
	`NumLoc = 0`
	`OBTTDo = 0`
	`PDo = 0`
	`TDo = 0`

ANTIGONE (Vanilla)	`fbbt_improvement_bound = 0`
	`use_obbt = 0`

SCIP (Vanilla)	`presolving/maxrounds = 0`
	`propagating/maxrounds = 0`

EAGO (Vanilla)	`cp_depth = 0`
	`dbbt_depth = 0`
	`fbbt_lp_depth = 0`
	`obbt_depth = 0`

EAGO (Blackbox Sampling)	`cp_depth = 0`
	`dbbt_depth = 0`
	`fbbt_lp_depth = 0`
	`obbt_depth = 0`

4.1. Surrogate ANN model

In Smith et al. [42], a surrogate ANN model of bioreactor productivity was constructed by fitting results from computationally expensive computational fluid dynamics simulations to a small ANN with one hidden layer. The surrogate model was then optimized to obtain ideal processing conditions for the bioreactor [42]. The objective is given by:

$$f^{\text{ANN}}(\mathbf{x}) = b_2 + \sum_{r=1}^{3} w_{2,r} \frac{2}{1 + \exp\left(-2\mathbf{w}_{1,r}^{\text{T}}\mathbf{x} + b_{1,r}\right)}, \tag{3}$$

with the weights vectors $\mathbf{w}_{1,r}$ (for $r = 1, 2, 3$) and $\mathbf{w}_2$, and biases $\mathbf{b}_1, b_2$ available in Smith et al. [42], and $\mathbf{x} \in X = [-1, 1]^8$.

The convergence plot for this problem is shown in Figure 4. The first and most direct comparison to make is between the ParBB algorithm and the blackbox sampling method used with the EAGO solver. Since neither solver uses pre- or post-processing techniques, and the same lower-bounding method is used, differences in the convergence profiles between these two methods are only due to the batched node processing of ParBB and ParBB's use of GPU hardware. While EAGO with the blackbox sampling method is able to converge in 95 s, ParBB converges in 4.3 s, giving a speedup of roughly 22x. ParBB and this version of EAGO required 6.6×10^5 and 8.0×10^5 node evaluations to solve the problem to guaranteed global optimality, respectively. The discrepancy in nodes is due to differences in when nodes were branched on in batch versus serial node processing, but since ParBB is able to process nodes in parallel on faster hardware, the solution was obtained in significantly less time.

In this problem, it is interesting to note that the vanilla and blackbox sampling versions of EAGO give roughly comparable performance, converging in 64 s and 95 s, respectively. This indicates that, for this problem, the improved lower bounds that can be obtained by using subgradient information only slightly outweigh the added computational cost of generating and solving LPs. By parallelizing the blackbox sampling method on a GPU, ParBB is able to achieve considerable performance gains

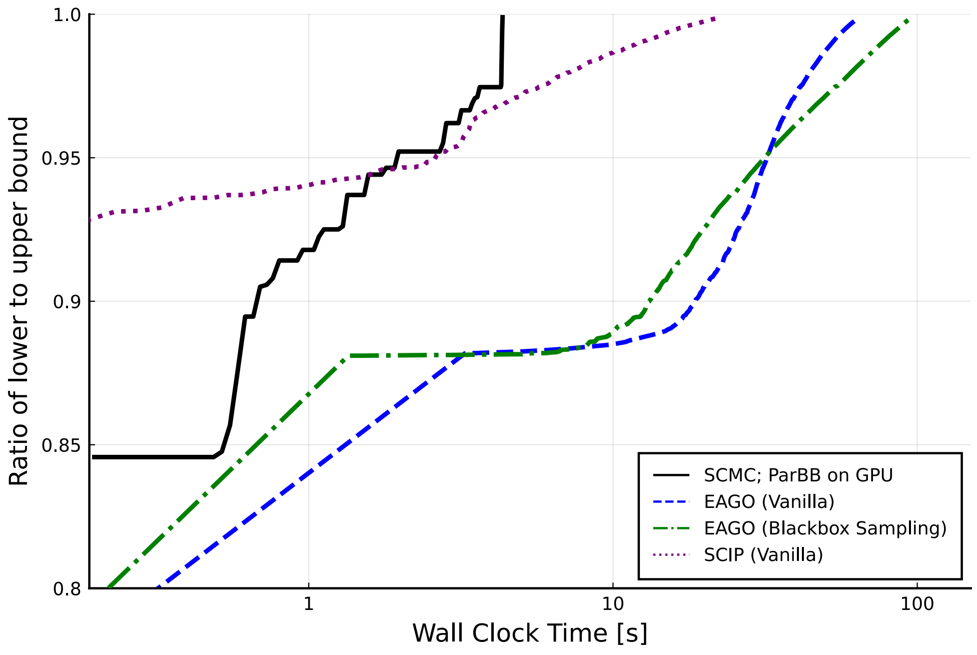

Figure 4. A convergence plot for the surrogate ANN model problem (Sec. 4.1) showing the performance of the novel ParBB algorithm (solid black) against vanilla versions of EAGO (dashed blue) and SCIP (dotted purple) and vanilla EAGO using the same lower-bounding routine as ParBB (dash-dotted green). ParBB converges the fastest in 4.3 s, followed by vanilla SCIP in 22 s, vanilla EAGO in 64 s, and EAGO with the blackbox sampling method in 95 s. As compared to the blackbox method implemented with EAGO, ParBB is roughly 22x faster, and despite not using subgradients, ParBB converges at least 5x faster than vanilla versions of the more mature solvers.

over the same method run on a CPU. Since the performance gained from this GPU parallelization is greater than the switch from subgradient-free to subgradient-based methods on a CPU, this results in ParBB achieving faster convergence than any of the vanilla versions of CPU-based solvers.

Comparing ParBB and the vanilla versions of SCIP and EAGO, for this problem, ParBB is able to reach convergence in only 4.3 s, outperforming the other tested solvers by more than 5x in terms of solution speed. Vanilla SCIP is the next fastest, converging in 22 s, followed by Vanilla EAGO, which solves the problem in 64 s. One curious aspect of these solution profiles is the conspicuous early plateau near 88% convergence identified by vanilla EAGO and EAGO with the blackbox sampling method, which is also present, but at a lower convergence, for ParBB. This plateau corresponds with finding a global solution and a lower bound that corresponds to that of an inclusion monotonic interval extension of the objective function on the domain. For the two versions of EAGO, since their solution methods start at the root node, an inclusion monotonic interval extension provides the first lower bound that is obtained, and improvements are only made once smaller domains with tighter relaxations are reached through branching. ParBB, on the other hand, starts by partitioning the root node into q subdomains, and then applies the blackbox sampling method, which is not guaranteed to provide results tighter than interval extensions. That is, across these subdomains, there are regions where applying the blackbox sampling method to the convex relaxation results in a lower bound that is lower than what can be obtained by applying interval arithmetic to the root node. Consequently, ParBB effectively starts at a lower plateau than what is found by both versions of EAGO, and only after

Table 5. The fitted parameters for (4), as obtained by Álvarez et al. [1], are tabulated for validation with the vapor-liquid equilibrium example (Sec. 4.2).

Parameter	Value
a_0	8.7369
a_1	27.0375
a_2	-21.4172
b_0	-2432.1378
b_1	-6955.3785
b_2	4525.9568

eliminating these regions by further branching is ParBB able to improve the global lower bound. Still, due to its ability to use GPU hardware, ParBB is able to overcome these differences to give overall faster performance than the vanilla versions of more mature solvers.

Finally, it should be noted that vanilla versions of ANTIGONE and BARON were unable to make progress on this problem within 2 hours, likely due to the curse of dimensionality as it applies to the higher dimensional space that these solvers operate in using the auxiliary variable method. With pre- and post-processing routines reactivated, ANTIGONE and BARON are able to solve this problem during preprocessing or at the root node in 0.13 s and 0.13 s, respectively.

4.2. Vapor pressure parameter estimation

The second problem of interest comes from Álvarez et al. [1], in which the authors were studying the vapor-liquid equilibria of electrolyte solutions for triple-effect absorption heat pump systems. Through measurements of the vapor pressure over a range of alkaline nitrate salt concentrations and temperatures, the authors were able to compare the performances of the proposed salt mass fractions for operation in a high-temperature absorption cycle. In this work, we reexamine the experimental vapor pressure data from Álvarez et al. [1] to validate the parameters shown in Table 5, which were obtained by the authors to fit the polynomial expression:

$$\log(\pi^{\text{calc}}) = \sum_{i=0}^{2} a_i w^i + \frac{\sum_{i=0}^{2} b_i w^i}{T}, \tag{4}$$

where π is the vapor pressure in kPa, w is the total mass fraction of salts in the solution, and T is the temperature in K.

For this problem, we use a scaled least-squares method for parameter fitting with the objective function given as:

$$f(\mathbf{p}) = \sum_{i=1}^{N} \left[\frac{(\pi^{\text{calc}}(\mathbf{x}_i, \mathbf{p}) - \pi_i^{\text{exp}})}{\pi_i^{\text{exp}}} \right]^2, \tag{5}$$

where $\mathbf{p} = (a_0, a_1, a_2, b_0, b_1, b_2)$ is the parameter vector, π^{calc} is the vapor pressure calculated by (4) at a given condition $\mathbf{x}_i = (w_i, T_i)$, π_i^{exp} is the experimental vapor pressure data provided by Álvarez et al. [1, Tab. 1] corresponding to the condition

$\mathbf{x}_i$, and N is the number of data points used for fitting, which is 50 in this example. Due to the presence of an exponential term with a complex argument used in the calculation of the objective function, this optimization problem is nonconvex, and with six parameters to fit, this problem is challenging for a B&B approach due to the curse of dimensionality. Bounds tightening techniques are particularly important for problems such as this with nested equation forms and multiple problem dimensions, since significant branching is required to obtain nodes that are sufficiently small in each dimension to provide tight relaxations. Parameter bounds for this problem were selected to be a box of diameter 0.2, centered around the parameters in Table 5.

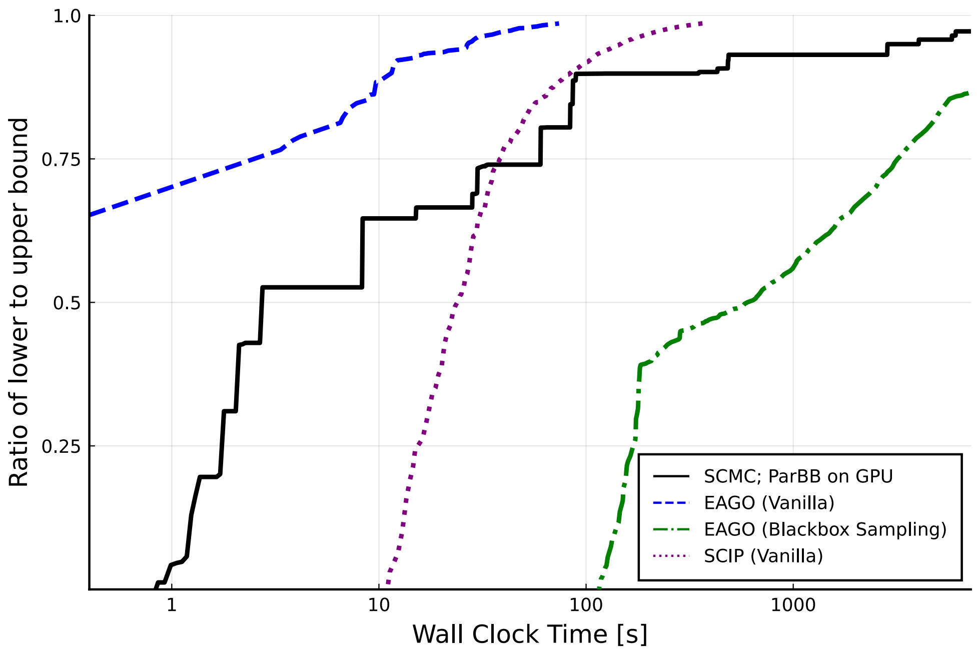

Figure 5. A convergence plot for the vapor pressure parameter estimation problem (Sec. 4.2) showing the performance of the novel ParBB algorithm (solid black) against vanilla versions of EAGO (dashed blue) and SCIP (dotted purple) and vanilla EAGO using the same lower-bounding routine as ParBB (dash-dotted green). While vanilla SCIP and EAGO are able to solve this problem within several hundred seconds due to their tighter relaxation techniques, the ParBB algorithm is unable to solve the problem due to the clustering problem, reaching a final convergence of 97.2% after 7200 s of runtime. The direct-comparison algorithm, EAGO with the blackbox sampling lower-bounding method, is also unable to solve the problem, reaching a final convergence of 86.5% after 7200 s of runtime. This level of convergence is reached by ParBB in 86 s, for a speed improvement of 84x.

As shown in Figure 5, vanilla implementations of EAGO and SCIP are able to converge to within the problem tolerance in 74 s and 360 s, respectively, while the methods that rely on the blackbox sampling method, including ParBB, are unable to converge within 7200 s. Notably, the convergence profile of ParBB shows the apparent behavior of the clustering problem [12, 54], in which an undesirably large number of nodes in the vicinity of a global minimizer must be visited before a solution can be obtained. This problem results from the behavior of objective functions near their global minimizers [12], with the number of nodes having exponential dependence on the problem dimensionality [35]. In particular, and as observed by Wechsung et al. [54], improving the tightness of the relaxations—and by extension improving the tightness of the resulting node lower bounds—is critical to mitigating the clustering problem [54]. While the blackbox sampling method used by ParBB is capable of providing fast lower bounds, the decreased tightness of the technique as compared to subgradient-based lower-bounding methods makes ParBB more susceptible to the clustering problem than solvers that utilize subgradients. This is most visible towards the end of the convergence profile whereby ParBB achieves roughly 90% convergence in 89 s, after which only marginal progress is made for the remaining 7111 s of the problem runtime.

Comparing ParBB to EAGO with the same blackbox sampling method, EAGO's final convergence of 86.5% is reached by ParBB in 86 s, for a direct-comparison performance increase of 84x. Both ParBB and this version of EAGO are outpaced by the vanilla EAGO solver, which is able to exploit subgradient information to calculate tighter lower bounds and thereby mitigate the clustering problem. As a comparison to demonstrate the reduced node requirements due to its tighter lower-bounding method, vanilla EAGO took roughly 5.1×10^4 iterations (5.1×10^4 nodes explored) to reach its solution, whereas ParBB completed 2.4×10^4 iterations (2.0×10^8 nodes explored—over 3 orders of magnitude more than vanilla EAGO) in two hours and only achieved a convergence of 97.2%.

As in the previous example, vanilla versions of ANTIGONE and BARON were unable to make notable progress on this problem within two hours, likely due to the high dimensionality of their auxiliary variable method approaches in addition to their dependence on domain reduction techniques. With pre- and post-processing techniques activated, ANTIGONE and BARON are able to solve this problem in under 5 s each (4.0 s and 3.3 s, respectively).

4.3. Transient absorption kinetics model

The final example of interest, originally described by Taylor [48], concerns the time-evolving concentrations of several chemical species after an initial laser flash pyrolysis. The problem consists of the following system of ODEs:

$$\begin{aligned}
\frac{dx_A}{dt} &= k_1 x_Z x_Y - c_{O_2}(k_{2f}+k_{3f})x_A + \frac{k_{2f}}{K_2}x_D + \frac{k_{3f}}{K_3}x_B - k_5 x_A^2,\\
\frac{dx_B}{dt} &= c_{O_2}k_{3f}x_A - \left(\frac{k_{3f}}{K_3}+k_4\right)x_B,\\
\frac{dx_D}{dt} &= c_{O_2}k_{2f}x_A - \frac{k_{2f}}{K_2}x_D,\\
\frac{dx_Y}{dt} &= -k_{1s}x_Z x_Y,\\
\frac{dx_Z}{dt} &= -k_1 x_Z x_Y,\\
& x_A(0) = x_B(0) = x_D(0) = 0,\ x_Y(0) = 0.4,\ x_Z(0) = 140.
\end{aligned} \tag{6}$$

Here, x_j is the concentration of species $j \in \{A, B, D, Y, Z\}$, and there are known constants of $T = 273$, $K_2 = 46\exp(6500/T - 18)$, $K_3 = 2K_2$, $k_1 = 53$, $k_{1s} = k_1 \times 10^{-6}$, $k_5 = 1.2 \times 10^{-3}$, and $c_{O_2} = 2 \times 10^{-3}$. The uncertain model parameters are $\mathbf{p} = (k_{2f}, k_{3f}, k_4)$ with $k_{2f} \in [10, 1200]$, $k_{3f} \in [10, 1200]$, and $k_4 \in [0.001, 40]$, and experimental data is given in terms of intensity, which has a known dependency on concentrations as $I^{\text{calc}} = x_A + \frac{2}{21}x_B + \frac{2}{21}x_D$, which originates from the Beer-Lambert law for relating measured absorbance to concentration with a correction for multiple species [40]. Thus, the objective function is formulated as:

$$f(\mathbf{p}) = \sum_{i=0}^{N} \left(I^{\text{calc}}(\mathbf{x}_i, \mathbf{p}) - I_i^{\text{exp}} \right)^2 ,$$

where $\mathbf{x}_i = (x_{A,i}, x_{B,i}, x_{D,i})$ represents the relevant discrete-time states at time i at which the intensity function is evaluated.

This problem was also addressed by Mitsos et al. [32], in which the ODE system was first discretized using the explicit Euler method to match the times of the 200 experimental data points, and then a McCormick relaxation library was applied to calculate convex and concave relaxations of the discrete-time state variables on the parameter domain. These relaxations were propagated forward through all time points to obtain a relaxation of the objective function. An alternative method of solving this problem using an implicit Euler discretization was utilized by Stuber et al. [44] and Wilhelm et al. [58], which was then solved using the global optimization of implicit functions approach. In this work, we follow the explicit Euler approach of Mitsos et al. [32]. The ODE system in (6) is discretized using 200 time points, and a convex relaxation of the objective function is obtained by propagating forward McCormick relaxations through the discrete-time states and comparing the resulting ODE trajectories with the experimental data.

Figure 6 shows a convergence plot that compares the performance results between the solvers in Table 4 on this problem. The most apparent aspect of this plot is that the ParBB algorithm converges faster than all other tested solvers. Although none of the solvers were able to converge fully within two hours due to the clustering problem, ParBB reached the highest level of convergence out of any solver, and reached 95% convergence in 0.88 s. This is 24x as fast as vanilla EAGO (21 s), and roughly 90x as fast as EAGO with the blackbox sampling method (79 s). Vanilla SCIP was unable to reach 95% convergence within the two hours of runtime. As another comparison point, EAGO with the blackbox sampling method eventually reached a convergence of 96.9% after two hours. This level of convergence was reached by ParBB in 61 s, giving a speedup of roughly 118x. This problem was too large to address using ANTIGONE or BARON, as the community license limited the number of nonlinear terms in the problem formulation.

This example represents a particularly strong use case for a parallelized B&B approach such as ParBB because evaluating the objective function to obtain an upper bound effectively requires the calculation of a trajectory of the underlying ODE system, and obtaining a lower bound requires generating bounds on the set of all possible trajectories of the ODE system in a given parameter subdomain. These operations are computationally expensive, which makes them a good target for GPU parallelization. With ParBB, this parallelization is accomplished by grouping together calculations for different nodes into batches, and because ParBB uses the bounds on trajectories to calculate node lower bounds on the GPU, there is no overhead required to transfer calculation results between the GPU and CPU. This parallelized processing of nodes, made possible by `SCMC` and the ParBB algorithm, enable the parallelization of this and other expensive calculations that may be needed to solve complex deterministic global optimization problems.

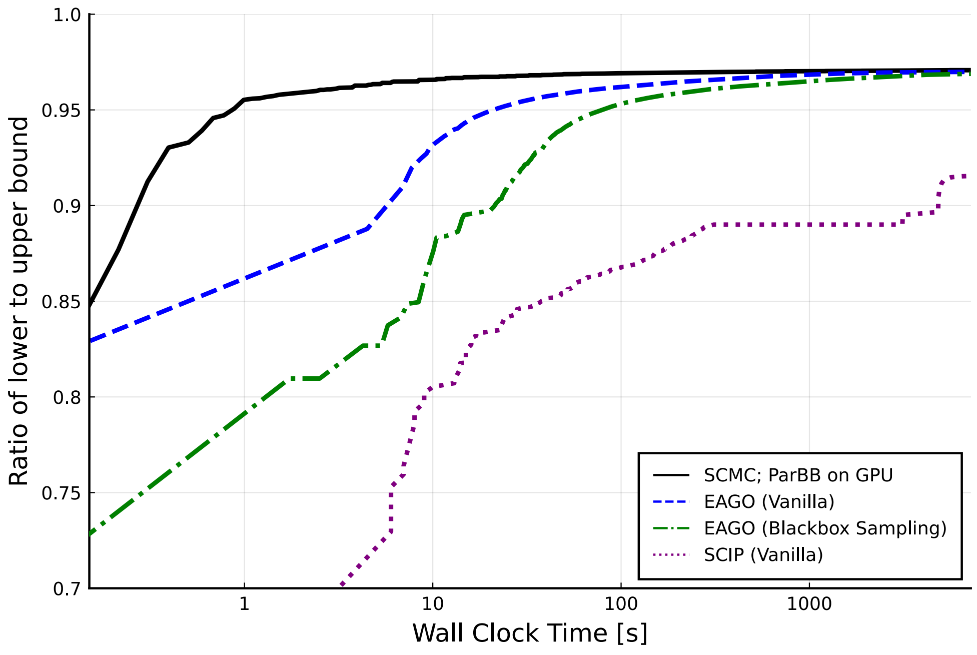

Figure 6. A convergence plot for the transient absorption kinetics problem (Sec. 4.3) showing the performance of the novel ParBB algorithm (solid black) against vanilla versions of EAGO (dashed blue) and SCIP (dotted purple) and vanilla EAGO using the same lower-bounding routine as ParBB (dash-dotted green). The ParBB algorithm shows the strongest performance out of any of the vanilla versions of more mature solvers, reaching 95% convergence in 0.88 s—roughly 24x faster than vanilla EAGO and roughly 90x faster than EAGO with the blackbox sampling method. At the two-hour time limit, EAGO with the blackbox sampling method reached a convergence of 96.9%, which was passed by ParBB in 61 s for a speedup of 118x. ParBB owes its performance to its ability to offload expensive objective function calculations to the GPU, as well as its ability to access a far greater number of nodes than traditional serial solvers.

5. Limitations

The new `SCMC` package, which can calculate pointwise evaluations of the interval extensions and convex and concave relaxations of math expressions in parallel on a GPU, has been shown to be fast and useful for global optimization when used in the correct conditions. However, it comes with several important limitations, some of which are briefly mentioned throughout this paper and many of which are the targets of ongoing research efforts. The most impactful of these limitations, which will likely require significant effort to overcome, are summarized in the following.

- Due to the large number of floating-point operations required to calculate McCormick-based relaxations, it is highly recommended to use double-precision floating-point numbers, including numbers on GPUs. Since most GPUs are designed for single-precision floating-point operation, forcing double-precision will often result in a significant performance hit. GPUs designed for scientific computing, with a higher proportion of double-precision-capable cores, are recommended to obtain the best double-precision floating-point performance for use with `SCMC`.
- Due to the high branching factor for calculating McCormick-based relaxations and the possibility of warp divergence, there will likely be a performance gap between optimization problems with variables covering positive-only domains and variables with mixed domains. Additionally, more complicated expressions

where the structure of a McCormick relaxation changes more frequently with respect to the bounds on its domain will likely perform worse than problems where the structure of the relaxation is more consistent. Some of this performance gap could be fixed by more intelligent ordering of B&B nodes prior to passing calculations to the GPU, but more research is needed to investigate whether the sorting process would result in a net loss or gain of speed.

In addition to these more challenging obstacles, there are several more manageable limitations that we will seek to address in future work, summarized in the following.

- `SCMC` does yet not calculate subgradients of relaxations, which are used in many modern global optimizers. Although it would be possible to implement subgradient calculations in a similar fashion as described in this article, additional advances would be needed to make use of the subgradients in a performant way, such as by creating a GPU-based LP solver that can solve batches of LPs simultaneously. With developments such as this, including subgradients would allow for tighter relaxations for each node, which could increase the number of nodes fathomed in each iteration and thereby ease the memory burden of the parallelized B&B algorithm.
- Complicated expressions may cause significant compilation time if passed through `SCMC` at one time, which can currently be avoided by manually factoring expressions into a few intermediate expressions and combining the results together in a user-defined function. Automating this process with further source code generation capabilities will simplify the use and implementation of `SCMC`.
- Due to limitations in CUDA, functions created with `SCMC` may only accept 32 CUDA arrays as inputs. Thus, functions with more than 8 unique variables will need to be split/factored by the user, similar to how complex expressions are handled, in order to be accommodated. As with the expression complexity limitation, an automated factorization and function generation process would be able to overcome this hurdle by ensuring that the number of inputs for any given intermediate function does not exceed the 32-input limit.
- The current version of `SCMC` is compatible with the elementary arithmetic operations `+`, `-`, `*`, `/`, and the univariate intrinsic functions `^2` and `exp`. Although this library covers a very broad set of nonconvex optimization problems of interest to researchers and practitioners, it represents a very small fraction of the exhaustive library of `McCormick.jl` used by EAGO. Continued development to include additional functions will allow a more diverse set of functions to be used with the parallelized B&B algorithm.

While these listed issues will all positively impact the utility of `SCMC` for deterministic global optimization applications, they were not viewed as fundamental requirements to disseminate the novel methods and software. Instead, they function as near-term attainable targets for future research.

6. Conclusions and future prospects

A new method of calculating McCormick relaxations was developed that is built on a SCT approach inspired by forward-mode AD and implemented as the software package `SourceCodeMcCormick.jl` in the Julia language. This approach enables the parallel evaluation of an arbitrarily large number of interval bounds and convex/concave relaxation points, each on potentially different domains, on a GPU. A deterministic global optimization algorithm was created that makes use of this new approach to solve lower- and upper-bounding problems for a large number of B&B nodes simultaneously. The performance benefits of this new algorithm were demonstrated using GPU parallelization on three challenging nonconvex global optimization problems relevant to researchers and practitioners in the physical sciences and engineering. Future improvements to the `SourceCodeMcCormick.jl` package are underway, including the incorporation of subgradients, which promise to greatly enhance the utility of this approach within deterministic global optimization software.

Acknowledgements

The authors would like to thank Dr. Matt Wilhelm for his fruitful discussions on Julia, EAGO, and McCormick-based relaxation libraries.

Data availability

The software library SourceCodeMcCormick.jl is available in a GitHub repository: `https://github.com/PSORLab/SourceCodeMcCormick.jl`

Disclosure statement

The authors report there are no competing interests to declare.

Funding

Funding is provided in part by the National Alliance for Water Innovation (NAWI) funded by the U.S. Department of Energy (DOE), Office of Energy Efficiency and Renewable Energy (EERE), Advanced Manufacturing Office (AMO), under Funding Opportunity Announcement Number DE-FOA-0001905, the DOE EERE AMO under Funding Opportunity Announcement Number DE-FOA-0002336, and by the National Science Foundation under Grant No. 1932723. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, NAWI, or the Department of Energy.

References

[1] M.E. Álvarez, M. Bourouis, and X. Esteve, *Vapor-liquid equilibrium of aqueous alkaline nitrate and nitrite solutions for absorption refrigeration cycles with high-temperature driving heat*, Journal of Chemical & Engineering Data 56 (2011), pp. 491–496.

[2] A.G. Baydin, B.A. Pearlmutter, A.A. Radul, and J.M. Siskind, *Automatic differentiation in machine learning: a survey*, Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. The Journal of Machine Learning Research, 18(153):1–43, 2018 (2015).

[3] D.E. Bernal, C.D. Laird, S.M. Harwood, D. Trenev, and D. Venturelli, *Impact of emerging computing architectures and opportunities for process systems engineering applications*, in *FOCAPO-CPC 2023*, Jan., San Antonio, TX. 2023.

[4] D.E. Bernal Neira, C.D. Laird, L.R. Lueg, S.M. Harwood, D. Trenev, and D. Venturelli, *Utilizing modern computer architectures to solve mathematical optimization problems: A survey*, Computers & Chemical Engineering 184 (2024), p. 108627.

[5] T. Besard, C. Foket, and B. De Sutter, *Effective extensible programming: Unleashing Julia on GPUs*, IEEE Transactions on Parallel and Distributed Systems (2018).

[6] C. Bischof and H. Bucker, *Computing derivatives of computer programs*, NIC Series 3 (2000).

[7] S. Bolusani, M. Besançon, K. Bestuzheva, A. Chmiela, J. Dionísio, T. Donkiewicz, J. van Doornmalen, L. Eifler, M. Ghannam, A. Gleixner, C. Graczyk, K. Halbig, I. Hedtke, A. Hoen, C. Hojny, R. van der Hulst, D. Kamp, T. Koch, K. Kofler, J. Lentz, J. Manns, G. Mexi, E. Mühmer, M.E. Pfetsch, F. Schlösser, F. Serrano, Y. Shinano, M. Turner, S. Vigerske, D. Weninger, and L. Xu, *The SCIP Optimization Suite 9.0*, Technical report, Optimization Online (2024), URL `https://optimization-online.org/2024/02/the-scip-optimization-suite-9-0/`.

[8] D. Bongartz, J. Najman, S. Sass, and A. Mitsos, *MAiNGO - McCormick-based Algorithm for mixed-integer Nonlinear Global Optimization*, Tech. rep., RWTH-Aachen (2018), URL `https://www.avt.rwth-aachen.de/global/show_document.asp?id=aaaaaaaaabclahw`.

[9] B. Chachuat, B. Houska, R. Paulen, N. Peri'c, J. Rajyaguru, and M.E. Villanueva, *Set-theoretic approaches in analysis, estimation and control of nonlinear systems*, IFAC-PapersOnLine 48 (2015), pp. 981–995.

[10] J. Chen and J. Revels, *Robust benchmarking in noisy environments*, arXiv e-prints (2016), arXiv:1608.04295.

[11] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, *cuDNN: Efficient primitives for deep learning* (2014).

[12] K. Du and R.B. Kearfott, *The cluster problem in multivariate global optimization*, Journal of Global Optimization 5 (1994), pp. 253–265.

[13] I. Dunning, J. Huchette, and M. Lubin, *JuMP: A modeling language for mathematical optimization*, SIAM Review 59 (2017), pp. 295–320.

[14] J. Forrest, T. Ralphs, S. Vigerske, H.G. Santos, J. Forrest, L. Hafer, B. Kristjansson, jpfasano, EdwinStraver, M. Lubin, Jan-Willem, rlougee, jpgoncal1, S. Brito, h-i gassmann, Cristina, M. Saltzman, tosttost, B. Pitrus, F. MATSUSHIMA, and to st, *coin-or/cbc: Release releases/2.10.11* (2023).

[15] M. Garland, S.L. Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, *Parallel computing experiences with CUDA*, IEEE Micro 28 (2008), pp. 13–27.

[16] R.X. Gottlieb and M.D. Stuber, *Global dynamic optimization using hardware-accelerated programming*, in *AIChE Annual Meeting 2022*, Nov., Phoenix, AZ. 2022.

[17] R.X. Gottlieb and M.D. Stuber, *PSORLab/SourceCodeMcCormick.jl* (2023), URL `https://github.com/PSORLab/SourceCodeMcCormick.jl`.

[18] R.X. Gottlieb, P. Xu, and M.D. Stuber, *Automatic source code generation of complicated models for deterministic global optimization with parallel architectures*, in *FOCAPO-CPC 2023*, Jan., San Antonio, TX. 2023.

[19] S. Gowda, Y. Ma, A. Cheli, M. Gwóźdź, V.B. Shah, A. Edelman, and C. Rackauckas, *High-performance symbolic-numerics via multiple dispatch*, ACM Communications in Computer Algebra 55 (2021), pp. 92–96.

[20] A. Griewank, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, no. 19 in Frontiers in Appl. Math., SIAM, Philadelphia, PA, 2000.

[21] R. Horst and H. Tuy, *Global Optimization: Deterministic Approaches*, Springer Berlin Heidelberg, 2013 Nov., URL `https://books.google.com/books?id=Pe_1CAAAQBAJ`.

[22] G.K. Kenway, C.A. Mader, P. He, and J.R. Martins, *Effective adjoint approaches for*

computational fluid dynamics, Progress in Aerospace Sciences 110 (2019), p. 100542.

[23] K.A. Khan, H.A.J. Watson, and P.I. Barton, *Differentiable McCormick relaxations*, Journal of Global Optimization 67 (2016), pp. 687–729.

[24] K.A. Khan, M. Wilhelm, M.D. Stuber, H. Cao, H.A.J. Watson, and P.I. Barton, *Corrections to: Differentiable McCormick relaxations*, Journal of Global Optimization 70 (2018), pp. 705–706.

[25] T. Koch, T. Berthold, J. Pedersen, and C. Vanaret, *Progress in mathematical programming solvers from 2001 to 2020*, EURO Journal on Computational Optimization 10 (2022), p. 100031.

[26] A. Krizhevsky, *Cuda-convnet* (2014), URL `code.google.com/p/cuda-convnet/`.

[27] D. Luebke, *CUDA: Scalable parallel programming for high-performance scientific computing*, in *2008 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro*, May. IEEE, 2008.

[28] D. Maclaurin, D. Duvenaud, and R. Adams, *Gradient-based hyperparameter optimization through reversible learning*, in F. Bach and D. Blei (eds.), *Proceedings of the 32nd International Conference on Machine Learning Proceedings of Machine Learning Research* vol. 37, *Proceedings of Machine Learning Research* vol. 37, 07–09 Jul, Lille, France. PMLR, 2015, pp. 2113–2122, URL `https://proceedings.mlr.press/v37/maclaurin15.html`.

[29] A. Makhorin, *Glpk (gnu linear programming kit)*, http://www.gnu.org/s/glpk/glpk.html (2008).

[30] G.P. McCormick, *Computability of global solutions to factorable nonconvex programs: Part I — convex underestimating problems*, Mathematical Programming 10 (1976), pp. 147–175.

[31] R. Misener and C.A. Floudas, *ANTIGONE: Algorithms for coNTinuous / Integer Global Optimization of Nonlinear Equations*, Journal of Global Optimization 59 (2014), pp. 503–526.

[32] A. Mitsos, B. Chachuat, and P.I. Barton, *McCormick-based relaxations of algorithms*, SIAM Journal on Optimization 20 (2009), pp. 573–601.

[33] R.E. Moore, *Methods and Applications of Interval Analysis*, Society for Industrial and Applied Mathematics, 1979 Jan.

[34] W.S. Moses, V. Churavy, L. Paehler, J. Hückelheim, S.H.K. Narayanan, M. Schanen, and J. Doerfert, *Reverse-mode automatic differentiation and optimization of gpu kernels via enzyme*, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. ACM, SC '21, 2021.

[35] A. Neumaier, *Complete search in continuous global optimization and constraint satisfaction*, Acta Numerica 13 (2004), pp. 271–369.

[36] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, *Automatic differentiation in pytorch*, in *NIPS 2017 Workshop on Autodiff*. 2017, URL `https://openreview.net/forum?id=BJJsrmfCZ`.

[37] D. Reed, D. Gannon, and J. Dongarra, *HPC forecast: Cloudy and uncertain*, Communications of the ACM 66 (2023), pp. 82–90.

[38] N.V. Sahinidis, *BARON: A general purpose global optimization software package*, Journal of Global Optimization 8 (1996), pp. 201–205.

[39] J.K. Scott, M.D. Stuber, and P.I. Barton, *Generalized McCormick relaxations*, Journal of Global Optimization 51 (2011), pp. 569–606.

[40] A.B. Singer, *Global dynamic optimization*, Ph.D. diss., Massachusetts Institute of Technology (2004).

[41] E.M. Smith and C.C. Pantelides, *Global optimisation of nonconvex MINLPs*, Computers & Chemical Engineering 21 (1997), pp. S791–S796.

[42] J.D. Smith, A.A. Neto, S. Cremaschi, and D.W. Crunkleton, *Cfd-based optimization of a flooded bed algae bioreactor*, Industrial & Engineering Chemistry Research 52 (2012), pp. 7181–7188.

[43] Y. Song, H. Cao, C. Mehta, and K.A. Khan, *Bounding convex relaxations of process models from below by tractable black-box sampling*, Computers & Chemical Engineering

153 (2021), p. 107413.

[44] M.D. Stuber, J.K. Scott, and P.I. Barton, *Convex and concave relaxations of implicit functions*, Optimization Methods and Software 30 (2015), pp. 424–460.

[45] M. Tawarmalani and N. Sahinidis, *Convexification and Global Optimization in Continuous and Mixed-Integer Nonlinear Programming: Theory, Algorithms, Software, and Applications*, Nonconvex Optimization and Its Applications, Springer, 2002, URL `https://books.google.com/books?id=MjueCVdGZfoC`.

[46] M. Tawarmalani and N.V. Sahinidis, *Convexification and Global Optimization in Continuous and Mixed-Integer Nonlinear Programming*, Springer US, 2002.

[47] M. Tawarmalani and N.V. Sahinidis, *A polyhedral branch-and-cut approach to global optimization*, Mathematical Programming 103 (2005), pp. 225–249.

[48] J.W. Taylor, *Direct measurement and analysis of cyclohexadienyl oxidation*, Ph.D. diss., Massachusetts Institute of Technology (2005).

[49] Theano Development Team, R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov, A. Belopolsky, Y. Bengio, A. Bergeron, J. Bergstra, V. Bisson, J.B. Snyder, N. Bouchard, N. Boulanger-Lewandowski, X. Bouthillier, A. de Brébisson, O. Breuleux, P.L. Carrier, K. Cho, J. Chorowski, P. Christiano, T. Cooijmans, M.A. Côté, M. Côté, A. Courville, Y.N. Dauphin, O. Delalleau, J. Demouth, G. Desjardins, S. Dieleman, L. Dinh, M. Ducoffe, V. Dumoulin, S.E. Kahou, D. Erhan, Z. Fan, O. Firat, M. Germain, X. Glorot, I. Goodfellow, M. Graham, C. Gulcehre, P. Hamel, I. Harlouchet, J.P. Heng, B. Hidasi, S. Honari, A. Jain, S. Jean, K. Jia, M. Korobov, V. Kulkarni, A. Lamb, P. Lamblin, E. Larsen, C. Laurent, S. Lee, S. Lefrancois, S. Lemieux, N. Léonard, Z. Lin, J.A. Livezey, C. Lorenz, J. Lowin, Q. Ma, P.A. Manzagol, O. Mastropietro, R.T. McGibbon, R. Memisevic, B. van Merriënboer, V. Michalski, M. Mirza, A. Orlandi, C. Pal, R. Pascanu, M. Pezeshki, C. Raffel, D. Renshaw, M. Rocklin, A. Romero, M. Roth, P. Sadowski, J. Salvatier, F. Savard, J. Schlüter, J. Schulman, G. Schwartz, I.V. Serban, D. Serdyuk, S. Shabanian, Étienne Simon, S. Spieckermann, S.R. Subramanyam, J. Sygnowski, J. Tanguay, G. van Tulder, J. Turian, S. Urban, P. Vincent, F. Visin, H. de Vries, D. Warde-Farley, D.J. Webb, M. Willson, K. Xu, L. Xue, L. Yao, S. Zhang, and Y. Zhang, *Theano: A python framework for fast computation of mathematical expressions* (2016).

[50] B. van Merrienboer, D. Moldovan, and A. Wiltschko, *Tangent: Automatic differentiation using source-code transformation for dynamically typed array programming*, in S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (eds.), *Advances in Neural Information Processing Systems* vol. 31, vol. 31. Curran Associates, Inc., 2018, URL `https://proceedings.neurips.cc/paper_files/paper/2018/file/748d6b6ed8e13f857ceaa6cfbdca14b8-Paper.pdf`.

[51] B. van Merriënboer, A.B. Wiltschko, and D. Moldovan, *Tangent: Automatic differentiation using source code transformation in python*, arXiv preprint arXiv:1711.02712 (2017).

[52] S. Vigerske and A. Gleixner, *SCIP: global optimization of mixed-integer nonlinear programs in a branch-and-cut framework*, Optimization Methods and Software 33 (2018), pp. 563–593.

[53] A. Wächter and L.T. Biegler, *On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming*, Mathematical Programming 106 (2006), pp. 25–57.

[54] A. Wechsung, S.D. Schaber, and P.I. Barton, *The cluster problem revisited*, Journal of Global Optimization 58 (2014), pp. 429–438.

[55] A. Wechsung, J.K. Scott, H.A.J. Watson, and P.I. Barton, *Reverse propagation of McCormick relaxations*, Journal of Global Optimization 63 (2015), pp. 1–36.

[56] M.J. White, *Nvidia says falling GPU prices are 'a story of the past'*, Digital Trends (2022), URL `https://www.digitaltrends.com/computing/nvidia-says-falling-gpu-prices-are-over/`.

[57] M.E. Wilhelm, R.X. Gottlieb, and M.D. Stuber, *PSORLab/McCormick.jl* (2020), URL `https://github.com/PSORLab/McCormick.jl`.

[58] M.E. Wilhelm, A.V. Le, and M.D. Stuber, *Global optimization of stiff dynamical systems*, AIChE Journal 65 (2019).

[59] M.E. Wilhelm and M.D. Stuber, *EAGO.jl: easy advanced global optimization in Julia*, Optimization Methods and Software 37 (2022), pp. 425–450.

[60] Y. Yajima, *Convex envelopes in optimization problems*, in *Encyclopedia of Optimization*, Springer US (2001), pp. 343–344.

[61] J. Ye and J.K. Scott, *Extended McCormick relaxation rules for handling empty arguments representing infeasibility*, Journal of Global Optimization 87 (2023), pp. 57–95.