RWTH Aachen

Department of Computer Science
Technical Report

Compiler-Generated Subgradient Code for McCormick Relaxations

Callum Corbett, Michael Maier, Markus Beckers, Uwe Naumann, Amin Ghobeity and Alexander Mitsos

ISSN 0935–3232 · Aachener Informatik-Berichte · AIB-2011-25

RWTH Aachen · Department of Computer Science · November 2011

The publications of the Department of Computer Science of *RWTH Aachen University* are in general accessible through the World Wide Web.

http://aib.informatik.rwth-aachen.de/

Compiler-Generated Subgradient Code for McCormick Relaxations

Callum Corbett, Michael Maier, Markus Beckers, Uwe Naumann, Amin Ghobeity, and Alexander Mitsos
beckers@stce.rwth-aachen.de

LuFG Informatik 12: Software and Tools for Computational Engineering

Abstract. McCormick Relaxations are special convex and concave under- and overestimators which are used in the field of nonconvex global optimization. As they are possibly nonsmooth at some points subgradients are used as derivative information. Subgradients are natural extensions of usual derivatives. They may be used to construct linear or piecewise linear relaxations. [Mitsos et al. 2009] developed a corresponding method and the library libMC (written in C++) for propagating relaxations and related subgradients using the tangent-linear mode of Algorithmic Differentiation (also known as Automatic Differentiation) by operator overloading. This paper extends [Mitsos et al. 2009] by providing libMC functionality by source transformation in Fortran. The corresponding Fortran module modMC is described. A research prototype of the NAG Fortran compiler has been extended with domain-specific inlining techniques to enable the generation of tangent-linear McCormick code. Speedups by factors of up to four with respect to the runtime of the respective libMC-based implementation can be observed. These results are supported by a number of relevant applications. To perform the numerical experiments, an interface between tangent-linear McCormick code written in Fortran and the existing C++-implementation of the branch-and-bound algorithm has been established.

1 Motivation and Context

Nonlinear programs (NLPs) with nonconvex objective function or constraints typically result in multiple local extrema some of which are suboptimal. There are various necessary and sufficient criteria to establish a local extremum, see e.g., [6], which are relatively easy to test, and local, gradient-based methods employ these criteria for termination. In other words, at termination, gradient-based methods can identify if a local extremum has been obtained. In contrast, no direct criteria exist for establishing a global optimum. Several heuristics exist to attempt global optimization with local solvers, such as multistart, i.e., repeated application of a local solver from different initial guesses. An alternative method, are gradient-free methods, such as evolutionary algorithms. These methods do not rely on local termination criteria, and as such have the potential of avoiding suboptimal local solutions. However, neither gradient-based methods with heuristics, nor gradient-free methods can guarantee that a global optimum has been obtained, at least not at finite termination. This article focuses on compiler support for *deterministic global optimization*, i.e., methods which can deterministically guarantee a global solution. Throughout the article, round-off error is not considered.

Deterministic global optimization algorithms [28, 15] rely on lower bounds on the optimal objective value, obtained via *relaxations*. A relaxation is an auxiliary problem which is guaranteed to have a better objective value (lower for minimization problems). For a relaxation to be useful for global optimization it must be

easier to solve than the original optimization problem. Popular relaxations of a nonconvex NLP are convex NLPs, linear programs (LPs) and interval-extensions.

1.1 Relaxations

Consider a constrained NLP

$$\begin{aligned} f^* = \min_{\mathbf{x}} & f(\mathbf{x}) \\ \text{s.t. } & \mathbf{g}(\mathbf{x}) \leq \mathbf{0} \\ & \mathbf{x} \in X, \end{aligned} \tag{1}$$

where $X \subset \mathbb{R}^n$ is an continuous bounded nonempty and convex set and $f : X \to \mathbb{R}$ and $\mathbf{g} : X \to \mathbb{R}^m$. Without loss of generality a hyperbox is assumed for the host set, i.e. $X = [\mathbf{x}^L, \mathbf{x}^U]$. Note that the objective function f and constraints $\mathbf{g}$ may be known explicitly or implicitly, e.g., through a computer code. This was demonstrated by [21] for algorithms with an a priori known number of iterations.

A relaxation can be obtained by constructing the *convex relaxations* of the objective function and constraints:

$$\begin{aligned} f^{u,*} = \min_{\mathbf{x}} & f^u(\mathbf{x}) \\ \text{s.t. } & \mathbf{g}^u(\mathbf{x}) \leq \mathbf{0} \\ & \mathbf{x} \in X. \end{aligned} \tag{2}$$

The new functions f^u and $\mathbf{g}^u$ are convex relaxations of f and $\mathbf{g}$, if they are convex on X and satisfy $f^u(\mathbf{x}) \leq f(\mathbf{x})$, $\mathbf{g}^u(\mathbf{x}) \leq \mathbf{g}(\mathbf{x})$ for all $\mathbf{x} \in X$. A similar definition holds for concave relaxations. Any feasible point of (1) (a point that satisfies the constraints) is also feasible in (2) and therefore $f^{u,*} \leq f^*$.

The construction of the convex relaxations is challenging. They must be converging, i.e., for any sequence $(X^i)_{i\in\mathbb{N}}$ of subsets of the host set X with

$$X^i = [\mathbf{x}^{i,L}, \mathbf{x}^{i,U}] \quad \text{and} \quad ||\mathbf{x}^{i,U} - \mathbf{x}^{i,L}|| \to 0$$

we must have

$$\begin{aligned} \max_{\mathbf{x}\in X^i} |f^u(\mathbf{x}) - f(\mathbf{x}))| &\to 0 \\ \max_{\mathbf{x}\in X^i} \max_{j\in\{1,2,\ldots,m\}} \left|g_j^u(\mathbf{x}) - g_j(\mathbf{x}))\right| &\to 0 \end{aligned}$$

Moreover convex relaxations should be cheap to evaluate.

A popular method is relaxation via the introduction of auxiliary variables and subsequent linearization; this method is used in the global solver BARON [28, 24, 29]. Another method are the $\alpha-$BB and $\gamma-$BB relaxations [1, 2] which are applicable to twice-continuous differentiable functions. This article focuses on McCormick relaxations [17, 18].

McCormick relaxations are applicable to *factorable functions*, i.e., functions defined by a finite recursive composition of binary sums, binary products and a given library of univariate intrinsic functions [21]. Assuming that the library of intrinsic functions has known convex and concave relaxations as well as function enclosures, these relaxations are propagated by rules for the sum, product and composition of functions.

1.2 Branch-and-Bound

In the absence of simple global optimality criteria, deterministic global optimization algorithms must in some sense search the set of the optimization variables. Unlike mixed-integer linear programs, explicit enumeration is not possible. Moreover, brute-force methods are not tractable for all but the simplest problems, and this has lead to the development of sophisticated algorithms, such as branch-and-bound (B&B) [14], branch-and-reduce [23] and outer approximation for mixed-integer nonlinear programs [16]. A thorough description of these algorithms is given in various textbooks, e.g., [28, 15] and here only the basic concept of a simple B&B is given.

B&B algorithms typically rely on an upper bound $UBD \geq f^*$ and a lower bound $LBD \leq f^*$ and convergence is achieved when the difference between the lower and upper bound is within a user-defined absolute tolerance $0 \leq UBD - LBD \leq \varepsilon_a$ and/or relative tolerance $0 \leq \frac{UBD-LBD}{|UBD|} \leq \varepsilon_r$. Note that the optimal objective value f^* is not known during the iterations of the algorithm, nor after termination.

Since convergence of the convex relaxations is only achieved at the limit, B&B represents the host set X via a *partition*, i.e., a collection of subsets $X^i \subset X$ that do not overlap ($X^i \cap X^k = \emptyset$) and cover the host set ($\cup X^i = X$). The subsets are termed *nodes*, and represented in the B&B tree. The first element of the B&B algorithm is *branching*, i.e., the refinement of the partition by subdividing the selected node into children nodes. Typically, this is performed by bisecting a variable. Note that in some algorithms multiple sets are branched simultaneously, or multiple branches are created from a single set at one iteration. There are several selection criteria, such as best-bound (selection of node with lowest lower bound LBD^i), breadth-first, or depth-first.

The second element of the B&B algorithm is *bounding*, i.e., the calculation of an upper and lower bound of the optimization problem with the variables restricted to the chosen node, UBD^i and LBD^i respectively. The lower bound is obtained by the solution of the relaxation, while the upper bound can be calculated by the local solution of the original optimization problem with the variables restricted to the chosen node. At branching LBD^i is inherited to the children nodes.

The overall upper bound (or incumbent) can be updated every time a better solution is found. Since a feasible point in a node is also feasible for the original problem we have $UBD = \min_i UBD^i$. The overall lower bound is the best bound $LBD = \min_i LBD^i$.

The third element of B&B is *fathoming*, i.e., elimination of the nodes with a lower bound which is higher than the incumbent. If $LBD^i \geq UBD$ then node i cannot contain an optimal solution and need no longer be considered. There are also heuristics employed, such as fathoming of nodes with a small diameter [25]. Another common trick is to fathom all the nodes with $LBD^i > UBD - \varepsilon_a$, since they cannot contain a significantly better solution than the incumbent.

In the worst case B&B algorithms have exponential complexity. Moreover, since the relaxations may become weaker with increasing dimensionality of the problem, the worst-case complexity can be even worse than exponential. Actual computational times depend a lot on the properties of the instance, and in some cases problems of engineering significance can be solved globally, e.g., [20].

In practice, often finding a global minimum is relatively easy and verifying its optimality is computationally expensive. An important point is however, that even before reaching ε_a convergence, the existing LBD is a *certificate of optimality*, since it gives an estimate of how far the solution found could be from the global optimum.

1.3 Subgradients

The application of Branch-and-Bound algorithms in our context include the calculation of the above defined McCormick relaxations as well as some "derivative" information of these. Since convex relaxations are possibly nonsmooth we use subgradients instead of usual derivatives to avoid possible miscalculations based on nondifferentiability. Subgradients are formally defined as follows:

Definition 1. *Let $f : X \to \mathbb{R}$ be function given on a convex set $X \subseteq \mathbb{R}^n$ and $f'(x,d) := \lim_{t \downarrow 0} \frac{f(x+td)-f(x)}{t}$ denote the directional derivative of f at a point x in direction d. Then the set*

$$\partial f(x) := \left\{ s \in \mathbb{R}^n \mid \langle s, d \rangle \leq f'(x,d) \ \forall d \in \mathbb{R}^n \right\}$$

is called the ***subdifferential*** *of f at x. A vector $s \in \partial f(x)$ is called* ***a subgradient*** *of f at x.*

[13] show that the subdifferential of any convex function is nonempty at any point. If the function f is differentiable at x then the only subgradient is given by the usual derivative of f at x. Hence subgradients are a natural extension of usual derivatives. [21] show that the tangent-linear (or forward) mode of Algorithmic Differentiation (AD) [12] can be applied to the propagation of subgradients of McCormick relaxations. The performance of the proposed C++-library libMC turned out to be superior when compared with other state-of-the-art global optimization tools on some test problems with special structure.

The next section gives a short summary of libMC. In Chapter 2 we present the corresponding new Fortran module modMC. Its superiority in terms of computational efficiency is demonstrated in Chapter 4. More significant improvements are due to the transition to source transformation implemented as a special inlining algorithm in a research prototype of the NAG Fortran compiler that has been introduced in a previous ACM TOMS article [22]. The inliner is described in Chapter 3.

1.4 libMC: Tangent-Linear McCormick Relaxations by Overloading in C++

We restrict the description of libMC to the essential facts. Refer to [21] and [9] for more details.

libMC provides the data type (C++ class) `McCormick`, that calculates convex and concave relaxations of factorable functions by means of operator overloading. Subgradients of these relaxations with respect to each optimization variable are propagated in parallel similar to standard forward mode AD.

The number of optimization variables `npar` is set via the `McCormick` member function `np(int npar)`. Optimization variables are typically initialized with the constructor

`McCormick (const double l, const double u, const double c, const int ip=-1).`

It creates a variable with bounds $[l, u]$, value `c`, and unique index `ip` ($0 \leq$ `ip` $<$ `npar`). Intermediate values can either be initialized by the above constructor, the constructor `McCormick (const double c)` (assigning the value `c` to the variable), or by assigning a double or McCormick variable using the $=$ operator. Intermediate variables' indices are set to -1 indicating that they are not optimization variables.

The binary operators addition, subtraction, multiplication and division as well as the intrinsic functions exponential, logarithmic, power, square root, and absolute value are implemented. As an example, the multiplication ($*$) of a positive scalar `s` with a McCormick variable `MCArg` is discussed here. The corresponding complete source code is available through [9]:

```
1  McCormick operator* ( const double s, const McCormick &MCArg ) {
2    if( MCArg._sizep != MCArg._np ) throw MC_Excp( MC_Excp::SIZE );
3    McCormick MCRes;
4    if ( s >= 0 ){
5
6      MCRes._l = s * MCArg._l;
7      MCRes._u = s * MCArg._u;
8
9      MCRes._cv = s * MCArg._cv;
10     MCRes._cc = s * MCArg._cc;
11
12     for( int ip=0; ip<MCRes._np; ip++ ){
13       MCRes._dcvdp[ip] = s * MCArg._dcvdp[ip];
14       MCRes._dccdp[ip] = s * MCArg._dccdp[ip];
15     }
16   }
17   else{
18     ...
19     }
20   }
21   return MCRes;
22 }
```

Listing 2.1. Excerpt from the scalar times `McCormick` operator

The operator ensures that the McCormick variable is acting on the correct number of optimization variables in line 2. An exception is thrown otherwise. Two cases must be distinguished: Multiplication with a positive scalar (**`if`**-branch) and with a negative scalar (**`else`**-branch, omitted here) inverting the upper and lower bounds. The McCormick variable returned as the result of the multiplication has the interval bounds `_l` and `_u` (lines 6 and 7), the values of the relaxations `_cv` and `_cc` (lines 9 and 10) and associated subgradients `_dcvdp[ip]` and `_dccdp[ip]` with respect to the set of optimization variables (lines 12–15). The respective member variables of `MCArg` are simply multiplied with `s` in this simple case. Refer to [9] for more complex examples.

2 modMC: Tangent-Linear McCormick Relaxations and Subgradients by Overloading in Fortran

This section covers modMC, a Fortran implementation of libMC. Relaxations are calculated by overloading arithmetic operators and intrinsic functions for the *derived type* `McCormick`. Subgradients are propagated along with the previously introduced bounds and relaxations due to potential non-smoothness of the latter. Support subroutines are provided for initialization, error handling, and to produce human-readable type instance printouts for debugging and documentation.

2.1 Compilation

modMC is distributed under the ECLIPSE PUBLIC LICENSE[1]. The latest version can be downloaded as a compressed tar-archive containing the complete source code from `http://wiki.stce.rwth-aachen.de/modMC/`. The build process complies with the GNU Code Standards[2] and is based on the GNU Autotools[3]. To compile modMC upon successful download, the following steps are required:

1. Extract package contents using the **`tar`** command-line utility:
   ```
   tar -xvj modmc-1.0.tar.bz2
   ```
2. Change into the directory created during package extraction. Note, that the version number 1.0 may differ for the latest version available:
   ```
   cd modmc-1.0
   ```
3. Run the `configure` command to determine local system settings and to select an appropriate compiler:
   ```
   ./configure
   ```
   Note: `./configure --help` lists available options.
4. Compile modMC. Binaries are placed inside the `src` directory:
   ```
   make
   ```
   The following files are created: `libmodmc.a` contains the binary object code and `mccormick.mod` contains the Fortran-specific interface description of modMC. It is possible to compile a Fortran project that uses modMC without actually installing modMC. The file `mccormick.mod` needs to be placed inside the project directory. The static library `libmodmc.a` must be provided only at link time.
5. *Optional.* To use modMC in multiple projects (all using the same Fortran compiler), a local installation can be generated:
   ```
   su
   make install
   exit
   ```
   `su` opens a new subshell with system administrator privileges; **`make`** `install` performs a local installation of modMC; **`exit`** closes the previously spawned subshell. If the **`sudo`** command is available, then the previous three lines can be combined into

[1] `http://www.eclipse.org/legal/epl-v10.html`
[2] `http://www.gnu.org/prep/standards/`
[3] `http://www.gnu.org/software/autoconf/`

```
sudo make install
```

After installation modMC is available to all users. The Fortran interface description file `mccormick.mod` is installed into the `include` directory defined during package configuration. Let `program.f90` use the McCormick data type and suppose that the GNU Fortran compiler **`gfortran`** is used. Then

```
gfortran program.f90 -lmodmc -I/usr/include/modmc
```

is a feasible build instruction. The compiler option `-lmodmc` advises **`gfortran`** to link with the `libmodmc.a` library. The `-I` parameter defines an include path to locate `mccormick.mod`. Default search paths may differ between machines. The installation path of modMC is set via the `--prefix` command line argument of the `configure` script.

2.2 Usage

Given a function implemented in Fortran, McCormick relaxations are easily obtained by replacing floating-point types for optimization, output and intermediate variables with `McCormick` and initializations of optimization variables with calls to the `mccormick_init()` subroutine, setting boundaries and a unique index for each variable. Relaxations are automatically propagated through the program flow.

```
program example
    implicit none

    integer, parameter :: n = 5
    integer i
    double precision, dimension(n) :: x
    double precision y

    do i = 1, n
      x(i) = dble(i)
    enddo

    y = 0.0d0

    do i = 1, n - 1
      y = y + x(i) ** 2 - x(i + 1)
    enddo

    write (*,*) y
end program
```

Listing 2.2. Simple example program

Listing 2.2 shows a simple sample program implementing a multivariate scalar function $f : \mathbb{R}^n \to \mathbb{R}$ defined as

$$y = f(x) \equiv \sum_{i=1}^{n-1} \left(x_i^2 - x_{i+1} \right)$$

for $n = 5$ and evaluated at $x_i = i$. Execution yields

```
 16.000000000000000
```

as output. Consider the following boundaries for the optimization variables x:

$$x_i \in \left[\frac{i}{2}, 2n\right], i = 1, \ldots, n\,.$$

Listing 2.3 shows the modifications required to augment the original computation with bounds, McCormick relaxations, and their subgradients with respect to the optimization variables x_i.

```
program example_mc
    ! Load McCormick module.
    use mccormick
    implicit none

    integer, parameter :: n = 5
    integer i
    ! Change type of parameters and objective.
    type(mccormick_type), dimension(n) :: x
    type(mccormick_type) y

    ! Set the absolute number of parameters used
    ! before initialization.
    call mccormick_set_np (n)

    ! Initialize parameters with lower and upper bounds;
    ! set their initial values and assign unique index (here: i).
    do i = 1, n
      call mccormick_init (x(i), &
            &   dble(i) / 2.0d0, dble(2*n), dble(i), i)
    enddo

    ! The function implementation itself remains unchanged.
    y = 0.0d0

    do i = 1, n - 1
      y = y + x(i) ** 2 - x(i + 1)
    enddo

    ! Call special output subroutine for McCormick variables.
    call write (y)
end program
```

Listing 2.3. McCormick-enabled version of Listing 2.2

Modifications to the original program are only applied to type definitions and initializations, including the specification of parameter bounds. The implementation of the function evaluation itself remains unchanged; McCormick relaxations and their subgradients are implicitly propagated through the execution of the function. The following output is generated:

```
 %MC-Value
 \interval-extension[
-3.2500000000000000E+01,  3.9300000000000000E+02
 ]
 \relaxations[
 1.6000000000000000E+01,  5.1000000000000000E+01
 ]
 \subgradient[
 2.0000000000000000E+00,  1.0500000000000000E+01
```

```
 3.0000000000000000E+00,  1.0000000000000000E+01
 5.0000000000000000E+00,  1.0500000000000000E+01
 7.0000000000000000E+00,  1.1000000000000000E+01
-1.0000000000000000E+00, -1.0000000000000000E+00
 ]
```

The output contains the calculated relaxations of f, as well as the corresponding subgradients. These values are accessible via the standard Fortran derived type component separator:

```
  ! Get the convex underestimator.
  write (*,*) y%cv
  ! Get the concave overestimator.
  write (*,*) y%cv
  ! Get the subgradients with respect to the 2nd variable.
  write (*,*) y%dcvdp(2), y%dccdp(2)
```

Listing 2.4. Access to objective components in **write**(mccormick_type y)

Indices conform to Fortran standards and start with 1 (libMC indices start with 0 conforming to C++ standards). Indices within subgradients correspond to the unique index assigned during parameter initialization and range between 1 and n.

2.3 Implementation Details

The modMC library defines the new derived type `McCormick` shown in Listing 2.5. All computations are based on instances of this structure and rely on the lower and upper interval bounds `l` and `u`, the relaxations `cv` and `cc` and their subgradients `dcvdp` and `dccdp`.

```
type mccormick_type
    sequence
    ! size of subgradients
    integer sizep
    ! lower bound
    double precision l
    ! upper bound
    double precision u
    ! convex underestimator
    double precision cv
    ! concave overestimator
    double precision cc
    ! subgradient of the convex underestimator
    double precision, dimension(:), allocatable :: dcvdp
    ! subgradient of the concave overestimator
    double precision, dimension(:), allocatable :: dccdp
end type mccormick_type
```

Listing 2.5. `McCormick` data type structure

The `sizep` component stores the number of optimization variables registered prior to the initialization of a `McCormick` type instance. This information is essential for the propagation of subgradients through the evaluation of the program as each `McCormick` type instance needs to store two arrays of size `sizep` for the subgradients of the convex underestimator and the concave overestimator with respect to the optimization variables.

Several computations of relaxations with a different number of optimization variables are possible. However, mixing `McCormick` type instances with subgradients of distinct sizes is not permitted. Therefore each operator or function needs to check the `sizep` component of its arguments and report an error, if they do not match. An example is shown in Listing 2.6. The internal `mccormick_error()` subroutine handles implementation-specific errors and aborts program execution with a detailed error message.

```
1  recursive function times_mc_mc (MCArg1, MCArg2) result (MCRes)
2    type(mccormick_type), intent(in) :: MCArg1, MCArg2
3    type(mccormick_type)             :: MCRes
4
5    if (MCArg1%sizep.ne.MCArg2%sizep) &
6    then
7      call mccormick_error (                  &
8          &   'times_mc_mc',                  &
9          &   'MCArg1%sizep.ne.MCArg2%sizep', &
10         &   'Argument␣sizes␣are␣not␣equal')
11   endif
12
13   ...
14 end function
```

Listing 2.6. Excerpt from the multiplication operator for two arguments of type `McCormick`: The sizes of the subgradients of both arguments need to match.

Listing 2.7 shows an example excerpt from the multiplication operator for a scalar `double precision` variable and a second argument of type `McCormick`. Note the obvious similarity with Listing 2.1. Additionally, modMC takes advantage of Fortran specific features such as array arithmetic in lines 24 and 25.
Subgradients in `McCormick` type instances are declared as **`allocatable`**. Allocatable components are a widely supported extension to the Fortran standard and have no direct impact on the practical use of modMC but yield greatly improved efficiency over standard Fortran pointers. modMC also provides full support for inliner side effects described in Section 3.5 and is able to gain an additional boost by re-using allocated memory of intermediate variables inside of loops and between variable scopes.

```
1  function times_dp_mc (s, MCArg) result (MCRes)
2      double precision    , intent(in) :: s
3      type(mccormick_type), intent(in) :: MCArg
4      type(mccormick_type)             :: MCRes
5
6      ! Initialize storage of subgradient
7      MCRes%sizep = MCArg%sizep
8
9      if (.not.allocated(MCArg%dcvdp)) then
10         allocate(MCRes%dcvdp(MCRes%sizep))
11         allocate(MCRes%dccdp(MCRes%sizep))
12     endif
13
14     if (s.ge.0.0d0) then
15         ! Compute bounds.
16         MCRes%l = s * MCArg%l
17         MCRes%u = s * MCArg%u
18
19         ! Compute McCormick relaxations.
```

```
20          MCRes%cv = s * MCArg%cv
21          MCRes%cc = s * MCArg%cc
22
23          ! Compute subgradient of McCormick relaxations.
24          MCRes%dcvdp = s * MCArg%dcvdp
25          MCRes%dccdp = s * MCArg%dccdp
26
27      else
28          ...
29      endif
30  end function
```

Listing 2.7. Excerpt from the multiplication operator for a scalar **double precision** variable and a second argument of type `McCormick`

2.4 Interoperability

In larger projects, different problems are often solved using different programming languages, either to benefit from language-specific features or due to personal preference. *Mixed Language Programming* often encounters several problems as described, for example, in [7]. modMC has been designed to allow utmost compatibility, based on the ability of modern Fortran compilers to generate object code that allows easy interaction with other languages such as C. For example, the symbol name of the overloaded multiplication operator in Listing 2.7 generated by the NAG Fortran compiler is `_mccormick_MP_times_dp_mc`, which is easily callable from other languages.

To enable binary interoperability with other languages, the structure of the `McCormick` type is declared as a **sequence**. This declaration guarantees that the memory layout of `McCormick` type components match the order of their declarations. Otherwise, a Fortran compiler might choose to optimize the type structure by reordering components, leading to possibly incorrect memory accesses from a foreign program. The well-defined memory layout allows the definition of equally structured data types in other programming languages to directly interact with modMC and its derived type instances.

3 Tangent-Linear McCormick Relaxations by Fortran Source Transformation

This section describes extended compiler optimizations for modMC in a research-prototype of the NAG Fortran Compiler based on a technique called *inlining*.

3.1 Inlining

In general, a Fortran program is separated into several modules and source files. A module provides data types and methods shared by other source files. During compilation, all modules are processed and the compiler generates a library and description file for each module. The module description file is used by all other sources to determine data type structures and resolve method calls (e.g. determine the specific method name for overloaded operators). An executable program is generated by linking all sources into a single binary executable that depends on the shared library files. Although such modular design reduces re-

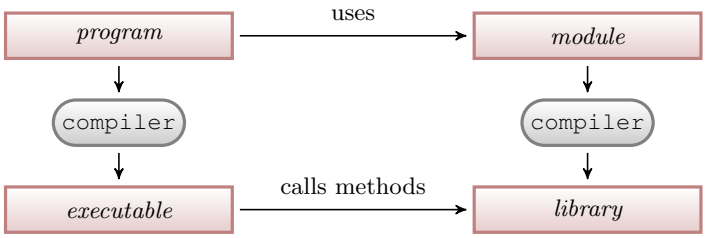

Fig. 1. Default compilation process

dundant code, structures code, improves maintenance and enables techniques like operator overloading, its main drawback is a general loss in runtime speed due the overhead generated by calls to methods. Operator overloading in particular is based on adding method calls for each operator used and results in a massive decrease of runtime efficiency.

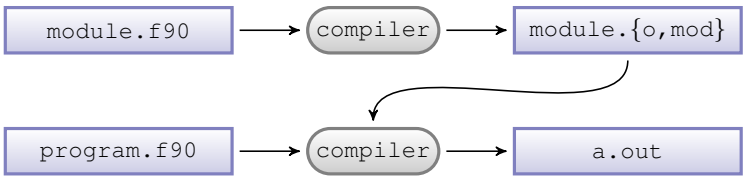

Fig. 2. File usage during default compilation process

The NAG Fortran compiler is a multiple phase Fortran to C compiler. It builds up a parse tree from the Fortran source and performs operations and transformations directly on the obtained parse tree. A special inliner has been developed to replace all occurrences of calls to known functions or procedures by their original implementation. Overloaded operators are internally represented by subroutine and function calls and are therefore included in the process. The inliner itself provides two modes of operation and does not interfere with the compilers' default procedure.

3.2 Collector Mode

The collector mode scans source files and stores metadata required by the propagation mode (see Section 3.3). It is usually applied once per module or source file of each subroutine or function. All supported statements[4] are stored in an intermediate XML-based file format referred to a *Inter Storage Module* (ISM). ISM files are recreated each time the module or subroutine source file is changed. They are interchangeable between different platforms. ISM files contain a hierarchical and complete description of all processed subroutines and functions. Unsupported method calls are left in place and do not result in an error.

[4] A supported statement is a call to a subroutine whose body the inliner is able to store and reconstruct.

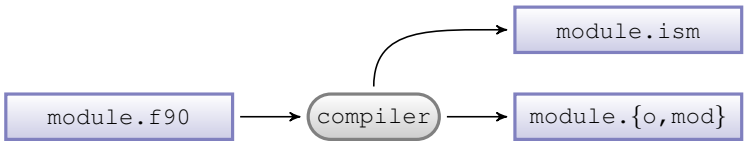

Fig. 3. Files processed and generated with the inliner's collector mode enabled. An ISM file is generated in addition to the default module object and interface definition files.

3.3 Propagation Mode

In propagation mode the given Fortran source files are scanned for occurrences of subroutine or function calls recorded in the corresponding ISM file. Every matching call is replaced by a dynamically generated intermediate parse tree recreated from the stored description of its original method body. Method matching is es-

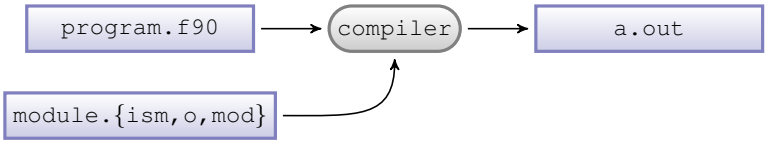

Fig. 4. Files processed and generated with the inliner propagation mode enabled; an additional ISM file is required

tablished both based on specific[5] method name comparison and on compatibility tests of the method signature. Each method parameter is consecutively checked against their stored counterparts for data type equality. All successful signature matches are stored in an intermediate map. This map is used during the creation of an intermediate parse tree from the stored description of its original method body to replace local variables of the original method with their matching callee counterpart. Temporary local variables are dynamically created in the current scope. The inliner stores its recursion state to determine whether a new statement is processed and re-uses local temporary variables for optimized runtime behavior. Unmatched method calls are skipped and left unaltered. While unsupported statements may result in a loss of optimization they conservatively preserve the correctness of the program.

3.4 Recursive Expansion

An additional recursion parameter controls the upper limit on how many times already inlined statements are at most re-scanned and probably are again expanded by the inliner. Recursive expansions resolve dependencies between operators and functions and flatten recursive functions up to the given recursion limit. In general, a recursion limit between 1 and 3 is mostly sufficient. Using recursive operators, the complexity of the generated code increases exponentially. Consequently, standard compiler optimizations may require substantial memory resources which may force the user to disable further optimizations.

[5] A specific method name is the name of the resolved method call used for an overloaded method or operator call.

3.5 Side Effects

In some cases, usage of the inliner produces side effects and needs special consideration regarding the design of a module.

Derived Type Requirements During propagation, complex statements are separated into smaller statements for each operation. This decomposition involves the generation of temporary variables as discussed in Section 3.3. These variables might be re-used within loops as shown in Listing 2.8 on lines 5 and 6.

```
1 ! Original.                    ! Augmented (intermediate).
2 type(user_type) :: a, b        type(user_type) :: a, b, t
3
4 do i = 1, n                    do i = 1, n
5                                   t = a / b
6    z = (a / b) ** n               z = t ** n
7 enddo                          enddo
```

Listing 2.8. Intermediate variable generation inside loops

A derived type *must* support assignments on possibly already initialized instances, even if its originally intended use does not require this. This is especially important for types using dynamically allocated memory as the `mccormick_type` defined by modMC. Aside from this new requirement, well designed data types might benefit from this behavior: Dynamically allocated memory can be re-used with each iteration and therefore result in an even less memory load and runtime improvement. The Fortran McCormick implementation supports re-using variables with allocated memory blocks for subgradient storage and gains an additional significant runtime speedup.

Private Members In Fortran modules may declare members private. Private members are not exported in module description files and cannot be referenced from outside the module. In order to enable the inliner to support as many methods as possible, every method that should be exported needs to be declared public, including every method or variable referenced by that method. To optimize recursive expansion, the chain of referenced methods and variables should be declared public up to a depth equal to the optimal recursion limit used.

3.6 Statistics

The inliner has been applied successfully to tangent-linear codes consisting of up to 122,000 lines of Fortran code. Up to 68,764 segments were inlined resulting in speedup of between 5 and 6. Slightly lower factors can be observed for tangent-linear McCormick code due to the larger complexity of each overloaded method call.

Inliner and Recursion Figure 3.6 shows a benchmark based on a low-resolution heat conduction problem to be discussed in Section 4.2. Various build scenarios are applied to repeated runs of the tangent-linear McCormick code. The test programs were compiled with and without the inliner using different levels of standard NAG Fortran compiler optimization and recursion depths. Preprocessing by the inliner without recursion results in a speedup of between 1.42 (no optimization) and 1.46 (`-O4`). Recursion level 2 raises this factor to 2.97.

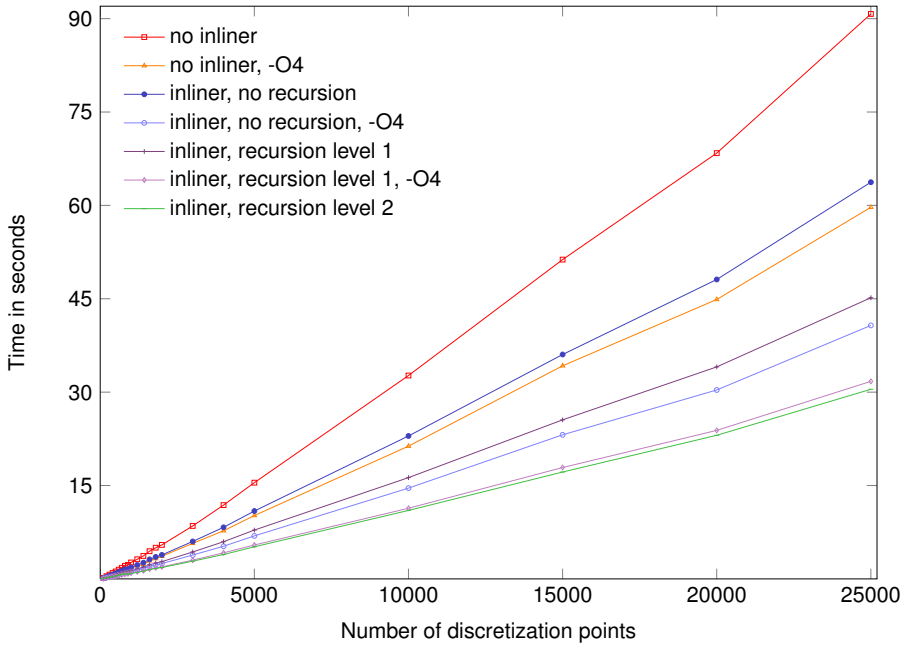

Fig. 5. Heat conduction example: Runtime for optimization runs with different problem sizes and build setups

Compile Time The following table lists compile time statistics, including the number of segments inlined and the size of the resulting binary for five different compilations of the heat conduction example. *Standard* compilation denotes a reference test case without inlining. The four inliner columns contain test results using the inliner recursion up to a depth of 3. The number of segments inlined equals the number of expansions performed by the compiler (including recursive expansions) by replacing a method call with its re-created body. We list the number of temporary variables generated and the total number of times new local temporaries are re-used.

	Standard	Inliner	Inliner 1	Inliner 2	Inliner 3
compilation time	0.40 s	1.89 s	7.11 s	32.67 s	769.88 s
segments inlined	-	48	147	670	3784
temporaries generated	-	9	47	255	1455
temporaries reused	-	26	146	618	3410
C source (generated)	38 Ki	245 Ki	1.1 Mi	5.9 Mi	35 Mi
binary size in Bytes	368 Ki	436 Ki	688 Ki	1.9 Mi	8.8 Mi
runtime 1001 points	1.97 s	1.50 s	1.25 s	0.96 s	0.96 s

Table 1. Inliner compilation statistics

Finally, we report the size of the generated C source[6] and binary file in Bytes and the runtime of the compiled binary evaluating the objective function using 1001 discretization points. Any optimization level in conjunction with inliner recursion depth ≥ 2 required more memory than our test machine (equipped

[6] The NAG Fortran compiler used is a multiple phase Fortran to C compiler.

with 3 GiB of RAM and 16 GiB of swap space) was able to provide; hence `-O0` was used. Runtime improvement stalled at a recursion depth of 3. A recursion depth of 2 turned out to be advantageous for the heat conduction example and will hence be used in further test setups to be presented in Section 4.2.

4 Results

In this Section, the architecture of the software developed to use modMC in combination with a branch-and-bound solver is presented, followed by three examples the software has been successfully applied to.

4.1 Software Architecture

This project is based on prior work on McCormick-based relaxations of algorithms by [21]. The libMC library discussed in Section 1.4 and the implementation of the branch-and-bound algorithm in C++ used in [21] are provided. As previously described in Section 1.2, the basic idea of branch-and-bound is to bound the optimal objective value between an upper and a lower bound. The optimization variables' set is then partitioned (branching) and the bounds are recalculated on this smaller interval. In the case of McCormick relaxations, the bounds converge with the optimization variables' bounds (by construction). Throughout this article minimization problems are considered. Convergence is checked by means of the user-defined absolute and relative tolerances ε_A and ε_R. Let ub and lb be the current upper and lower bound, respectively, then the bounds are considered to have converged if either $ub - lb < \varepsilon_A$ or $\frac{ub-lb}{|ub|} < \varepsilon_R$. Note that either tolerance leads to convergence and therefore one of the tolerances can be set to zero by the user, thus enforcing the other convergence check.

The user provides a function for the lower bound and one for the upper bound. The software calls these functions and provides the intervals within which the optimization variables are to take their values. In the case of the upper bounding function, the values of the optimization variables obtained by the lower bounding problem are also passed to the function. It is up to the user to evaluate upper and lower bounds within these functions. The method of interest in this paper is by means of McCormick relaxations. The function for the lower bound uses the McCormick data type for the optimization variables, the intermediate variables and the objective function. To obtain a lower bound from this, there are numerous possibilities. For example, a local solver can be used on the non-linear convex relaxation of the objective function to find its global minimum. Other possibilities are to use the subgradients of the relaxations to derive affine underestimators and calculate their minimum or the usage of the natural interval extensions which are propagated simultaneously throughout the calculation, see [21]. Throughout the article the lower bound is calculated as maximum of the values obtained via the latter two possibilities.

Now the importance of the choice of tolerances becomes clear: Assume that in a non-convex minimization problem the upper bound ub has reached the global minimum and is therefore the solution. As lb is a lower bound obtained via relaxations, it underestimates the exact value and converges towards it. Thus, although the solution has been found, the solver iterates until the lower bound

has converged to within the tolerances set by the user. Due to this, reducing the tolerances does not always lead to a better approximation although the solver requires more iterations.

In the following, the Fortran interface for the branch-and-bound solver described above and the structure of the user-provided problem specification are described.

The Fortran Interface The interface consists of four subroutines the user must supply: `getnpar(...)` to get the number of optimization variables, `init(...)` for initialization and `ubp(...)` and `lbp(...)` for the upper and lower bounding procedures, respectively.

When executed, the software first acquires the number of optimization variables via `getnpar` to allocate arrays of the correct dimension for the optimization variables' values and their bounds. Next, the `init` procedure is called to load data required by the bounding procedures, set the value and bounds of each optimization variable and configure the branch-and-bound algorithm's convergence tolerances discussed in Subsection 4.1. To obtain an initial upper bound, the upper bound procedure `ubp` is called. Finally, the branch-and-bound solver is started, which induces calls to the upper and lower bounding functions `ubp` and `lbp`. Upon convergence, the value of the objective function and the values of all optimization variables are printed to `clog`.

The user must provide these four subroutines named exactly as they appear here with the parameter lists matching the specification given below. The tasks each subroutine is expected to execute are summarized below the explanation of the parameters. Note that by default most Fortran compilers append an underscore (_) at the end of subroutines and functions. This behavior is expected and required by the software.

1. `getnpar(`**`int`**`& npar)`

   `int``& npar` is to return the number of optimization variables, so that arrays of the correct dimension can be allocated. Initially `npar` is set to zero by the software.

   Task: Set `npar` to the number of optimization parameters.

2. `init(`**`double`**`* pl,` **`double`**`* pu,` **`double`**`* guess,` **`double`**`& epsilon_bb_a,` **`double`** `& epsilon_bb_r)`

   `double``* pl` is an array for the lower bound of each optimization variable. The entry `pl[ip]` corresponds to the optimization variable with the unique identifier `ip` and has to be set accordingly. The required memory is allocated and initially all entries are arbitrarily set to zero by the software.

   `double``* pu` is an array for the upper bound of each optimization variable. The entry `pu[ip]` corresponds to the optimization variable with the unique identifier `ip` and has to be set accordingly. The required memory is allocated and initially all entries are arbitrarily set to zero by the software.

   `double``* guess` is an array for the initial guess of each optimization variable's value. The entry `guess[ip]` corresponds to the optimization variable with the unique identifier `ip` and has to be set accordingly. If no further information is available, often the arithmetic mean of the upper and lower

bound `pu` and `pl` is chosen. The required memory is allocated and initially all entries are arbitrarily set to zero by the software.

double `& epsilon_bb_a` has to be set to the absolute tolerance of the branch-and-bound solver (see Subsection 4.1). Initially, `epsilon_bb_a` is arbitrarily set to 0.1.

double`& epsilon_bb_r` has to be set to the relative tolerance of the branch-and-bound solver (see Subsection 4.1). Initially, `epsilon_bb_r` is arbitrarily set to 0.1.

Tasks: initialize `pl`, `pu`, `guess`, `epsilon_bb_a` and `epsilon_bb_r` to the problem specific values. If necessary, run tasks that only need to be executed once (e.g. read data from file, initialize shared variables etc.).

3. `ubp(`**double**`* pl,` **double**`* pu,` **double**`* solval,` **double**`* solution)`

   double`* pl` is an array containing the current lower bound of each optimization variable. Changing the values of `pl` has no effect and is discouraged.

   double`* pu` is an array containing the current upper bound of each optimization variable. Changing the values of `pu` has no effect and is discouraged.

   double`* solval` is a scalar into which an upper bound has to be written.

   double`* solution` is an array which contains the values of the optimization variables obtained by `lbp` which are to be overwritten by the new values of the optimization variables.

   Tasks: The subroutine must calculate an upper bound for the problem within the given bounds of the optimization variables and write the upper bound and the values of the optimization variables to `solval` and `solution`, respectively.

4. `lbp(`**double**`* pl,` **double**`* pu,` **double**`* solval,` **double**`* solution)`

   double`* pl` is an array containing the current lower bound of each optimization variable. Changing the values of `pl` has no effect and is discouraged.

   double`* pu` is an array containing the current upper bound of each optimization variable. Changing the values of `pu` has no effect and is discouraged.

   double`* solval` is a scalar into which a lower bound has to be written.

   double`* solution` is an array into which the new values of the optimization variables have to be written. The required memory is allocated and initially all entries are arbitrarily set to zero by the software.

   Tasks: The subroutine must calculate a lower bound for the problem within the given bounds of the optimization variables and write the lower bound and the values of the optimization variables to `solval` and `solution`, respectively.

4.2 Case Studies

In this Section, three examples using the Fortran interface of the branch-and-bound algorithm and modMC are considered. They deal with heat conduction, reaction kinetics and reverse osmosis. In the first two cases, a nonconvex parameter estimation is performed. As these examples are based on [21] and have already been compared to the commercial code BARON, they will only be compared to the existing implementations that use libMC. In the latter case, an unconstrained optimization problem is solved and compared to BARON version 8.1.5 available through GAMS version 23.0.2. The commercial code BARON [25] is a deterministic nonlinear program solver that uses a branch-and-reduce algorithm.

1D Heat Equation An ordinary differential equation for one-dimensional heat conduction

$$\frac{d^2T}{dx^2} = \frac{q^0(x) + q^1(x)T}{p}, \qquad x \in [0, 1]$$

based on [21] is considered in this example. The boundary conditions are set to $T(x)|_{x=0} = 500$ and $T(x)|_{x=1} = 600$ and the only unknown parameter is the thermal conductivity $p \in [0.01, 10]$. The affine heat source term is characterized by

$$q^1(x) = 1 \text{ and } q^0(x) = \begin{cases} -35,000 & \text{if } x \in [0.5, 0.6] \\ 5,000, & \text{otherwise} \end{cases}$$

Discretizing this boundary value problem by means of finite differences results in a linear equation system. The measurement data that is to be approximated by this model is generated for discretizations ranging from 100 to 25000 elements. Using a different heat source term to generate the data than for the model results in an imperfect match between the two and simulates the case that the model is incorrect. To determine the optimal value for p, a least-squares error approach is chosen.

The program used in [21] is rewritten in Fortran to use the developed software. Both programs' runtimes are measured for each discretization. To average out OS-dependent runtime-variations, the problem is solved 50 times in a loop. Reading data is excluded from the timing, only the performance of the branch-and-bound solver is measured. For both the Fortran and the C++ implementation, compiler optimization is set to `-O1` (to measure the different implementations but not compiler optimization), using gcc version 4.3.1 and the NAG Fortran compiler version 5.1. The respective runtime with full compiler optimization `-O3` compared to the current setting yields an additional speedup of approximately 1.5 for the C++ version and approximately 1.9 for the Fortran version without inliner. The scaling behavior with respect to the number of discretization points can be seen in Table 2 and Figure 6. It is worth noting that all three programs yield identical results and perform the same number of iterations for each discretization.

In this example, using the Fortran implementation with modMC results in an average speedup of 1.5 over the libMC version. The inliner-enabled Fortran implementation gains an addition speedup of factor 2.1 and results in an overall speedup of factor 3.2 over the libMC version. All three implementations scale approximately linearly with respect to the number of discretization points apart from small fluctuations.

The discretization error in the generated data becomes negligible for discretizations of 500 elements and more, while coarser resolutions generate values below the correct temperature. This can be seen in Figure 7. Figure 7 also shows the corresponding solutions. All solutions overshoot on the peak between $x = 0.5$ and $x = 0.6$ due to the purposely incorrect model chosen for this example. As the discretization error becomes negligible, the solutions also converge. The mismatch due to the incorrect model remains between the data and the results.

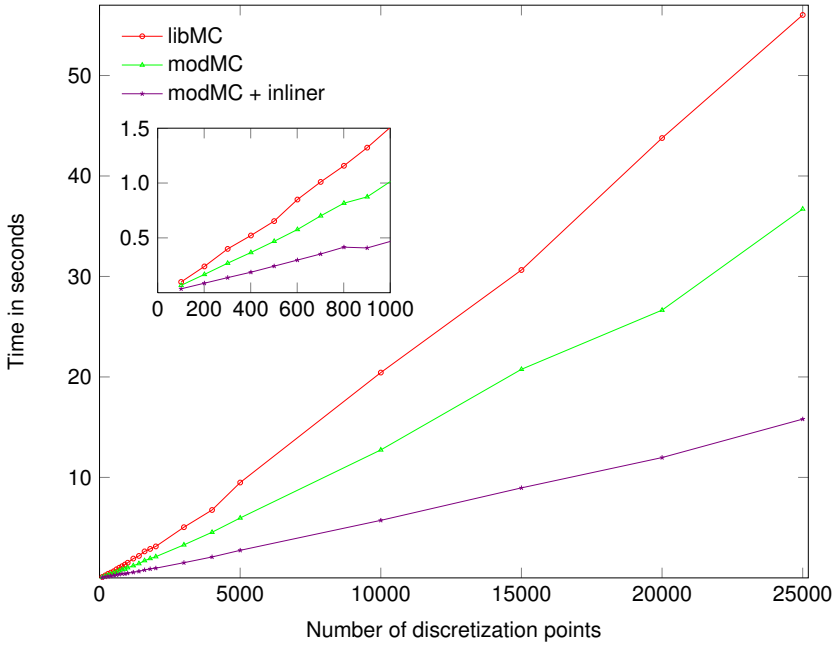

Fig. 6. Graphical version of Table 2

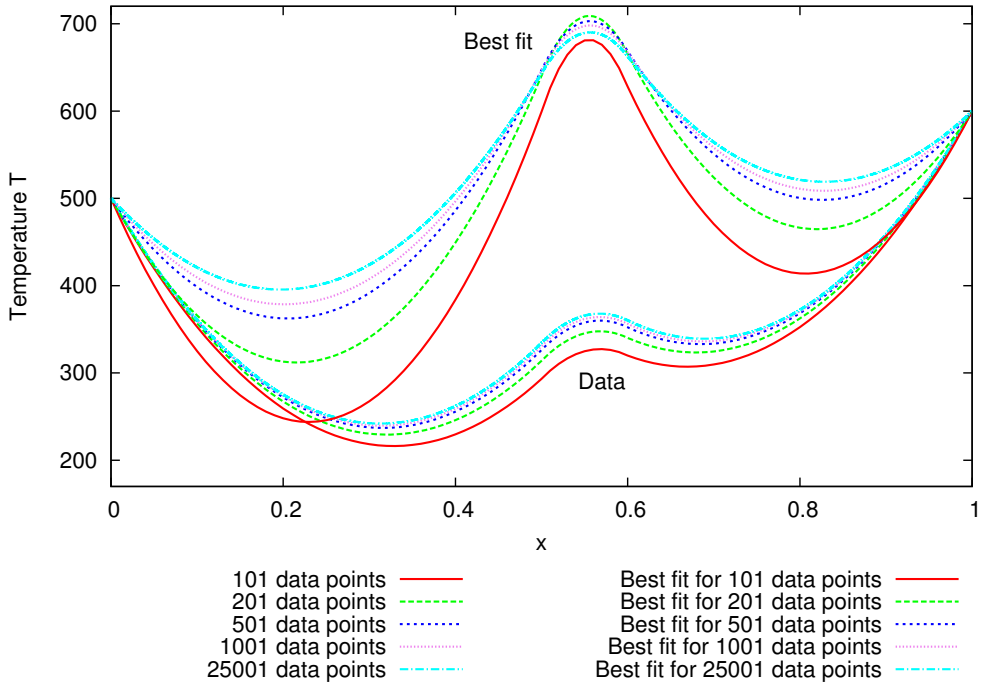

Fig. 7. Discretization errors and the solutions of the 1D heat example

Discretization	libMC	modMC	Inliner
101	0.0982 s	0.0675 s	0.0356 s
201	0.2396 s	0.1667 s	0.0868 s
301	0.3995 s	0.2692 s	0.1376 s
401	0.5218 s	0.3665 s	0.1869 s
501	0.6527 s	0.4688 s	0.2423 s
601	0.8495 s	0.5764 s	0.2976 s
701	1.0102 s	0.7006 s	0.3532 s
801	1.1573 s	0.8162 s	0.4152 s
901	1.3227 s	0.8741 s	0.4082 s
1001	1.5087 s	1.0140 s	0.4680 s
1201	1.9227 s	1.2149 s	0.5564 s
1401	2.1874 s	1.4212 s	0.6506 s
1601	2.6363 s	1.7183 s	0.7876 s
1801	2.8842 s	1.9316 s	0.8898 s
2001	3.1315 s	2.1107 s	0.9671 s
3001	5.0440 s	3.2869 s	1.5125 s
4001	6.7581 s	4.5422 s	2.0822 s
5001	9.4904 s	5.9644 s	2.7437 s
10001	20.4356 s	12.7373 s	5.7268 s
15001	30.6428 s	20.7603 s	8.9590 s
20001	43.7786 s	26.6504 s	11.9762 s
25001	56.0273 s	36.7148 s	15.8029 s

Table 2. Runtime comparison of libMC, modMC and modMC with inliner enabled for different numbers of discretization-points using the 1D heat example

Kinetic Mechanism An example for chemical kinetics taken from [21], originally based on [30], [26] and [27], is considered. The resulting nonlinear ordinary differential equation system is

$$\frac{dx_A}{dt} = k_1 x_Z x_Y - x_{O_2}\left(k_{2f} + k_{3_f}\right) x_A + \frac{k_{2f}}{K_2} x_D + \frac{k_{3_f}}{K_3} x_B - k_5 x_A^2$$

$$\frac{dx_Z}{dt} = -k_1 x_Z x_Y, \qquad \frac{dx_Y}{dt} = -k_{1s} x_Z x_Y$$

$$\frac{dx_D}{dt} = k_{2f} x_A x_{O_2} - \frac{k_{2f}}{K_2} x_D, \qquad \frac{dx_B}{dt} = k_{3f} x_A x_{O_2} - \left(\frac{k_{3f}}{K_3} + k_4\right) x_B$$

with the initial conditions

$$x_A(t)|_{t=0} = 0 \quad x_B(t)|_{t=0} = 0 \quad x_D(t)|_{t=0} = 0 \quad x_Y(t)|_{t=0} = 0.4 \quad x_Z(t)|_{t=0} = 140$$

and the constants $T = 273$, $K_2 = 46e^{\frac{6500}{T}}$, $K_3 = 2K_2$, $k_1 = 53$, $k_{1s} = k_1 \cdot 10^{-6}$, $k_5 = 0.0012$ and $x_{O_2} = 0.002$. Values for the unknown parameters $k_{2f} \in [10, 1200]$, $k_{3f} \in [10, 1200]$ and $k_4 \in [0.01, 40]$ are to be computed in such a way, that the model represents data obtained through measurements in the best possible manner. A least-squares error approach is chosen as optimization criterion and the integration in the time-domain is performed by means of the explicit Euler method with a constant step size dt. The output variable $x_I = x_A + \frac{2}{21} x_B + \frac{2}{21} x_D$ is measured.

The program used in [21] is rewritten in Fortran to use the developed software. As the data originates from experimental measurements, only 200 measurements with a resolution of $dt_{data} = 0.01$s are available. To scale the model, an n^{th} of the step size dt_{data} is used for the Euler method, $dt = \frac{dt_{data}}{n}$, and every n^{th} step is then compared to the measurements to calculate the squared error. Only integer values $n \in \mathbb{N}$ are considered here to avoid interpolation between measured data points. The effect of smaller step sizes on the solution can be seen in Figure 8, using the optimal solution values for k_{2f}, k_{3f} and k_4. It shows that although the Euler method is only of first order, the discretization error is negligible in comparison to the error due to the model. For step sizes $dt < \frac{dt_{data}}{5}$ the accuracy of the computed values does not improve significantly. For decreasing step sizes, the computed trajectory of $x_I(t, n)$ becomes flatter at the initial peak (see Figure 8). This is due to the explicit Euler method, that overshoots on concave parts of functions like the initial peak and undershoots on convex parts, with larger step sizes resulting in greater errors. As a result, the least-squares error increases for decreasing values of dt as the gap between the measured data and the computed values becomes larger.

Scaling the problem has a negative impact on performance with superlinear growth in both number of iterations and runtime. Obviously, halving dt_{data} results in double as many operations per solver iteration, as the Euler method requires twice the number of iterations. Moreover, increasing the number of operations results in weaker bounds. The reason for this is that for each operator the bounds, depending on the variable bounds and the evaluation point, are recalculated. With every additional operation, the relaxations can only become weaker, not tighter. Due to this, the convergence rate of the upper and lower bound decreases, resulting in an increasing number of necessary solver iterations.

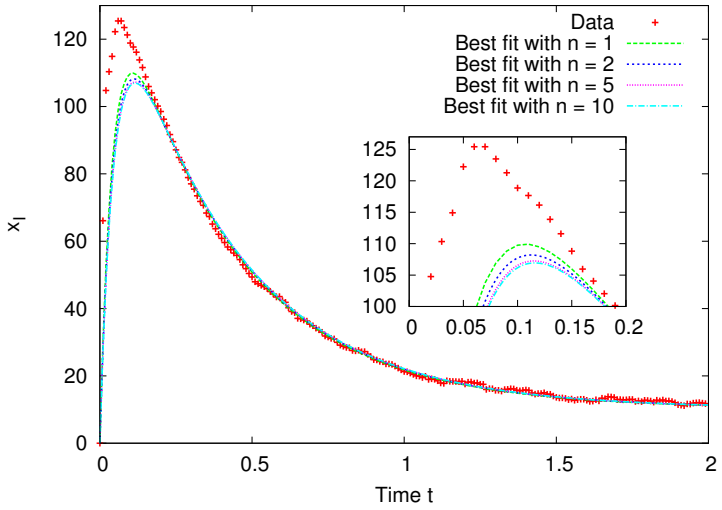

Fig. 8. Variation of n for the kinetic mechanism example with optimal parameters

This drawback is illustrated in Figure 9 regarding the runtime and in Figure 10 regarding the number of iterations, using absolute tolerances $\varepsilon_A = 2,000$ and $\varepsilon_A = 1,800$ for the branch-and-bound solver. Both the runtime and the number of iterations increase exponentially for decreasing step sizes dt.

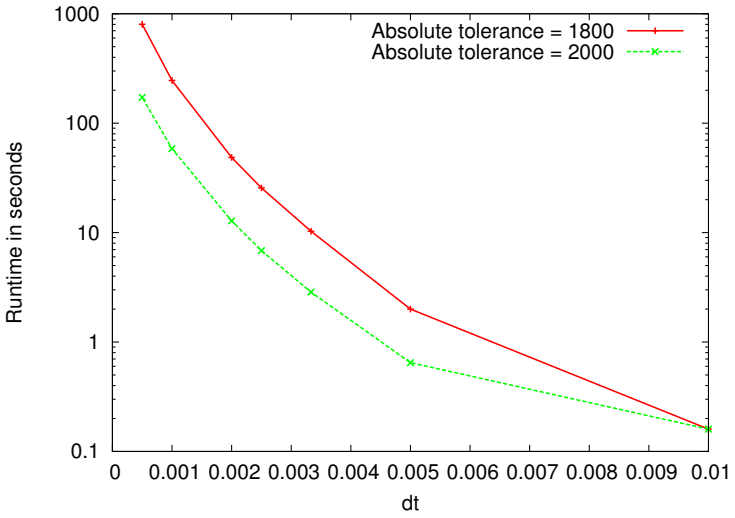

Fig. 9. Runtime of the kinetic mechanism example for different values of dt

In most cases, increasing the accuracy of the integration results in identical values for the parameters k_{2f}, k_{3f} and k_4, thus underlining that the discretization error is small compared to the error in the model. However the value of the objective function, the least-squares error, grows larger for smaller values of dt, due to the decreasing discretization error as explained above. For example, the

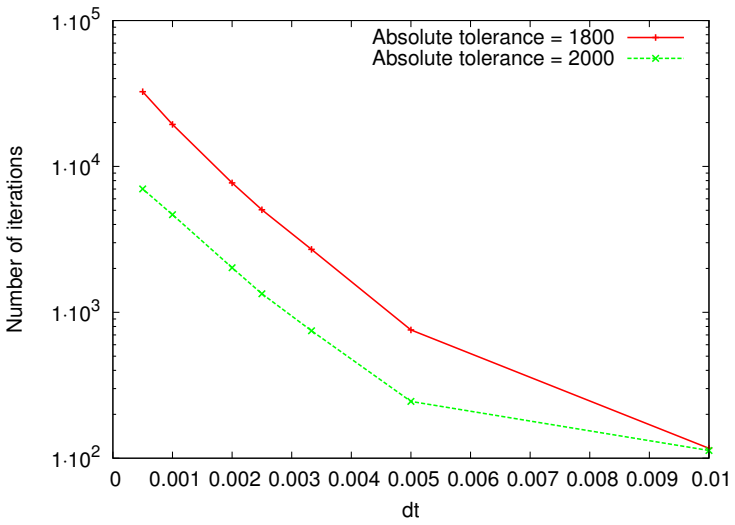

Fig. 10. Number of iterations for different values of dt with the kinetic mechanism example

same values for the unknown parameters are obtained with an absolute tolerance of 1,600 while the least-squares errors are 9,798, 11,639 and 12,256 for $n = 1$, $n = 2$ and $n = 3$, respectively.

In the case of an absolute tolerance of 1,400, a better approximation is found with $n = 2$ and $n = 3$ than with $n = 1$. The values of the optimization variables are $k_{2f} = 902.5$, $k_{3f} = 307.5$ and $k_4 = 40.0$ for $n = 1$ and $k_{2f} = 753.75$, $k_{3f} = 456.25$ and $k_4 = 40.0$ for $n \in \{2, 3\}$. Although they are very different values, they lead to similar trajectories. The least-squares errors each decrease by approximately 1% for $n \in \{2, 3\}$. However, the better approximation can also be obtained by choosing an absolute tolerance of 1,000 and $n = 1$, reducing the least-squares error by approximately 0.2%. As can be seen in Table 3, reducing the absolute tolerance and using $n = 1$ is faster in this example.

n	ε_A	Time	Iterations
1	1,000	11 s	8,073
2	1,400	25 s	9,495
3	1,400	89 s	22,883

Table 3. Time and number of iterations needed to obtain the same solution with respect to n and the absolute tolerance of the branch-and-bound solver ε_A

This leads to the conclusion, that refining the integration is not the best approach to gain more accurate results in this case, instead decreasing the absolute tolerance is preferable.

Reverse Osmosis In this section an example from [11] is considered. Reverse osmosis (RO) is a membrane based, pressure driven desalination technology (Figure 11). Semi-permeable membranes used in water desalination have high permeability for water and very low permeability for dissolved substances. Hence, by applying a pressure difference across the membrane the water contained in the feed water is forced through the membrane whereas most of the dissolved substances are rejected. The water passing through the membrane has a low concentration of dissolved substances, and forms what is called the permeate flow (Figure 11). The concentration of dissolved substances in the permeate flow is a function of the feed water quality and membrane characteristics, most importantly the membrane's salt rejection, R. The permeate flow rate is a function of several parameters such as feed pressure, feed water total dissolved concentration (TDS), membrane characteristics, and recovery ratio, defined as the ratio of the permeate flow rate over the feed water flow rate. Performance of the RO unit is also affected by several other phenomena such as fouling and concentration polarization. Thorough discussion of these phenomena is outside the scope of this article; they are explained in detail elsewhere (e.g., [31]). However, they are briefly introduced here as they appear in the following RO model. Fouling refers to the accumulation of foreign materials on the membrane's active surface. In presence of the fouling layer on the surface of the membrane, the resistance of the membrane to the flow of water through the membrane increases. Consequently, fouling affects the minimum pressure requirement for the RO, and hence the energy cost of the process. Concentration polarization refers to the concentration of solute at vicinity of the membrane surface, i.e. its boundary layer. The solute concentration in a thin boundary layer at the feed side of the membrane surface is higher than the solute concentration in the bulk of the feed water. Like membrane fouling, concentration polarization has effects on the permeate feed flow rate and the minimum required feed pressure.

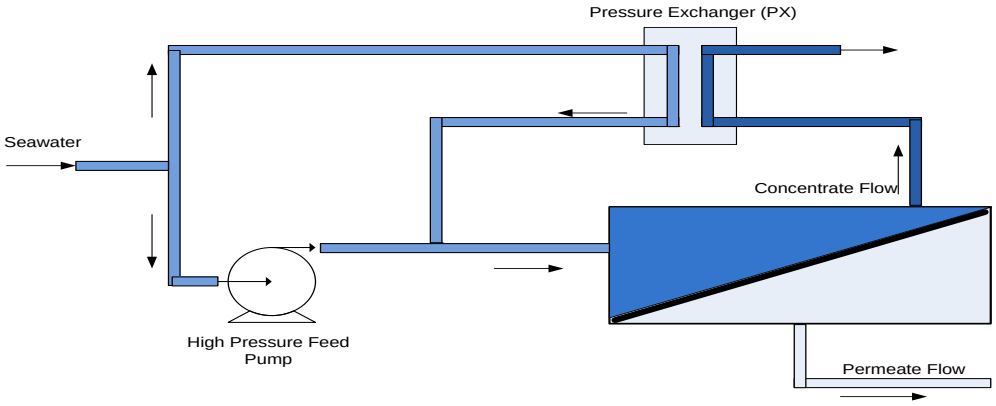

Fig. 11. Schematic of the RO process

The pressure difference across the membrane must be higher than the osmotic pressure difference between the feed water and the permeate water. Osmotic pressure is the pressure produced across the membrane due to the total

dissolved solids (TDS) concentration difference between the solutions on the two sides of the membrane (feed and permeate). In seawater RO (SWRO), the feed pressure to overcome the osmotic pressure of the feed side is high (45 - 85 bar, depending on the concentration and quality of the feed water). The pressure difference between the feed and concentrate streams is small (0 - 7 bar). Hence, the concentrate flow contains a considerable energy that is usually recovered using various energy recovery systems such as pressure exchangers [8, 19] or Pelton wheel turbines [3].

Theoretical Models for Specific Energy Requirement of RO

Flow Rates. The available RO models in the literature typically give the minimum energy requirements for separation. Here, a model developed by leading membrane provider DOW Chemical Company [10] has been adopted to estimate the actual energy consumption of SWRO using a FILMTEC SW-380® membrane. High efficiency pressure exchangers (PX) are also used in the model for estimating the recovered energy from the high pressure concentrate flow. The permeate water flow rate Q_{pw} and feed pressure P_f are related by:

$$Q_{pw} = A_{perm}(TCF)(FF)\left(P_f - \overline{\Pi} - \frac{\Delta P_{fc}}{2}\right) S_e \tag{3}$$

where A_{perm} is the membrane permeability, S_e the membrane's active surface, TCF the temperature correction factor, FF the fouling factor, ΔP_{fc} the average pressure difference between the feed and concentrate streams. $\overline{\Pi}$ is a function of the osmotic pressure of the feed water π, the concentration of feed C_f and concentrate flows C_c, and the concentration polarization factor PF:

$$\overline{\Pi} = (PF)\frac{C_c}{C_f}\pi \tag{4}$$

The polarizing factor depends on the recovery ratio Y and is approximated by

$$PF = \exp(0.7Y) \tag{5}$$

The concentration of dissolved solids in the permeate C_p and the concentrate stream is found from:

$$C_p = (1 - R)\, C_f \tag{6}$$

$$C_c = \frac{C_f}{1 - Y} - \frac{Y}{1 - Y}C_p \tag{7}$$

In the SWRO process, the higher the feed temperature, the higher the permeate flow rate. TCF may be approximated by the formulation provided by the membrane manufacturer or avaliable models in literature. Here, the following equation is used for TCF [5]:

$$TCF = 1.03^{(T_w - 298)} \tag{8}$$

where T_w is the feed water temperature in Kelvin. Other relations in the literature and the formulation provided by the membrane provider give similar values.

System. It should be noted that an RO desalination system may have multiple stages in which the concentrate from one stage is fed into a subsequent stage for further recovery, and/or the permeate water from one stage is fed into a subsequent stage for increasing purity in the product water. Booster pumps may be required between stages. Each stage of the system has a number of pressure vessels N_v which contain a number of elements N_{epv} (usually 6, 7, or 8). A single stage system is considered here. This assumption, according to the membrane provider guidebook [10], is a valid assumption given the size of the membrane and the application selected here. The minimum number of elements required in an RO system depends on the desalination plant capacity Q_{pw}^T and the design flow rate chosen for the membrane. Choosing Q_{pw} as design permeate flow rate, found from the membrane performance data [31], the total number of elements in the system, N_e, is found from:

$$N_e = \frac{Q_{pw}^T}{Q_{pw}} \tag{9}$$

The total number of vessel is then simply calculated as

$$N_v = \frac{N_e}{N_{epv}} \tag{10}$$

Energy. In SWRO, the energy consumed by high pressure feed pump constitutes the main energy requirement of RO. The power required to pressurize the feed stream W_{hp}, and the recovered portion of this power W_{rec} are found from:

$$W_{hp} = (P_f - P_{in}) \frac{N_e \cdot Q_{pw}}{Y \eta_p \eta_m} \tag{11}$$

and

$$W_{rec} = (P_f - \Delta P_{fc}) \frac{N_e \cdot Q_{pw} (1 - Y)}{Y} \eta_{px} \tag{12}$$

where η denotes efficiency factors and the subscripts p, m and px stand for pump, electric motor, and pressure exchanger, respectively. The net total energy requirement of the system is then estimated by

$$E_{net} = \frac{W_{hp} - W_{rec}}{3600} \tag{13}$$

The net specific energy requirement, i.e. the energy required for producing 1 m^3 of fresh water is given by

$$E_s = (W_{hp} - W_{rec}) \frac{1}{3600 N_e Q_{pw}} \tag{14}$$

As shown above, various variables in the model, such as PF and the recovered energy of the system depend on the recovery ratio of the RO system. As such, it is desired to find out the optimum recovery ratio for the objective of minimizing the specific energy consumption.

The governing equations of the RO model and the values of the parameters are summarized below.

$$
\begin{aligned}
Q_{pw} &= A_{perm}(TCF)(FF)\left(P_f - \overline{\Pi} - \frac{\Delta P_{fc}}{2}\right) S_e \\
\overline{\Pi} &= (PF)\frac{C_c}{C_f}\pi \\
PF &= \exp\left(0.7Y\right) \\
C_p &= (1-R)\,C_f \\
C_c &= \frac{C_f}{1-Y} - \frac{Y}{1-Y}C_p \\
TCF &= 1.03^{(T_w - 298)} \\
N_e &= \frac{Q_{pw}^T}{Q_{pw}} \\
N_v &= \frac{N_e}{N_{epv}} \\
W_{hp} &= (P_f - P_{in})\,\frac{N_e \cdot Q_{pw}}{Y\eta_p\eta_m} \\
W_{rec} &= (P_f - \Delta P_{fc})\,\frac{N_e * Q_{pw}\,(1-Y)}{Y}\eta_{px} \\
E_{net} &= \frac{W_{hp} - W_{rec}}{3600}
\end{aligned}
$$

Parameter	Unit	Value
Q_{pw}	m^3/h	1.1
C_f	mg/l	35000
S_e	m^2	35.3
A_{perm}	$l/m^2/h/bar$	1.2
FF	-	0.85
T_w	K	303
R	-	0.9975
η_P	-	0.90
η_M	-	0.95
η_{PX}	-	0.95
P_{in}	Pa	$2.76 \cdot 10^5$
ΔP_{fc}	Pa	$3.45 \cdot 10^5$

Solution of the RO optimization problem The RO optimization problem is implemented in three versions, two using BARON and one in Fortran using the developed software. The difference between the two BARON models is the branching behavior. The first version uses BARON's default branching behavior, branching on all variables. In the second version, branching is restricted to only branch on the optimization variable Y by setting the branching priority to zero for all other variables.

All three implementations calculate a solution in less than 0.5 seconds for relative tolerances ε_R set to zero and absolute tolerances ε_A between 10^{-1} and the minimal tolerance supported by BARON, 10^{-9}. Note that BARON will use a relative tolerance of 10^{-9} even if it is set to zero. The solution calculated by the Fortran implementation is better for tolerances $\varepsilon_A \leq 10^{-2}$ by approximately $1.6 \cdot 10^{-3}$. Figure 12 compares the results obtained by the three implementations.

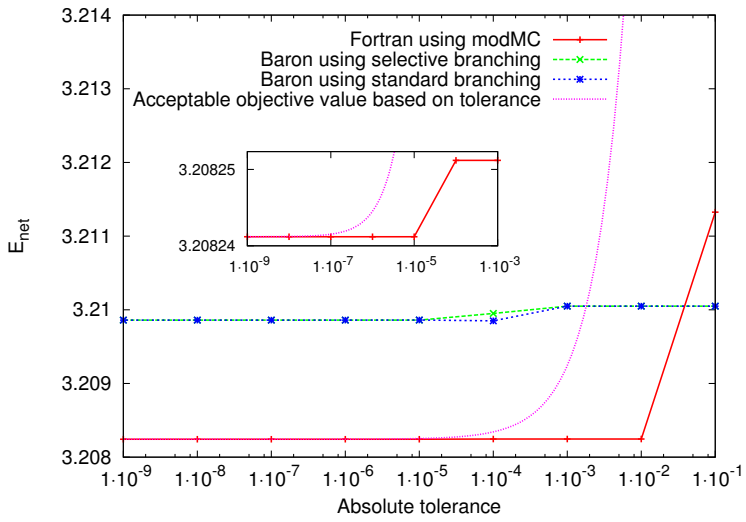

Fig. 12. Comparison of the solutions found for the reverse osmosis example by modMC and BARON

All solutions should lie beneath the acceptable objective value based on tolerance as it is defined as the sum of the optimal solution and the absolute tolerance, $E_{net,opt} + \varepsilon_A$. Both solutions calculated by BARON do not fulfill this condition.

Figure 13 pictures the convex and concave relaxations of $E_{net} = f(Y)$ and additionally the affine under- and overestimators are pictured in Figure 14. Note that the objective function is convex, although some intermediate functions are nonconvex. The concave envelope (i.e. the tightest concave overestimator), the secant between the two endpoints, is not the one calculated by modMC. Also, due to the nonconvex intermediate values, the convex relaxation is not the convex envelope, which is the function itself. For this example, a simple local solver would have sufficed to find the global optimum. Deterministic global optimization is applied nevertheless, as it is generally not possible to decide easily whether a function with non-convex intermediate expressions is convex or not.

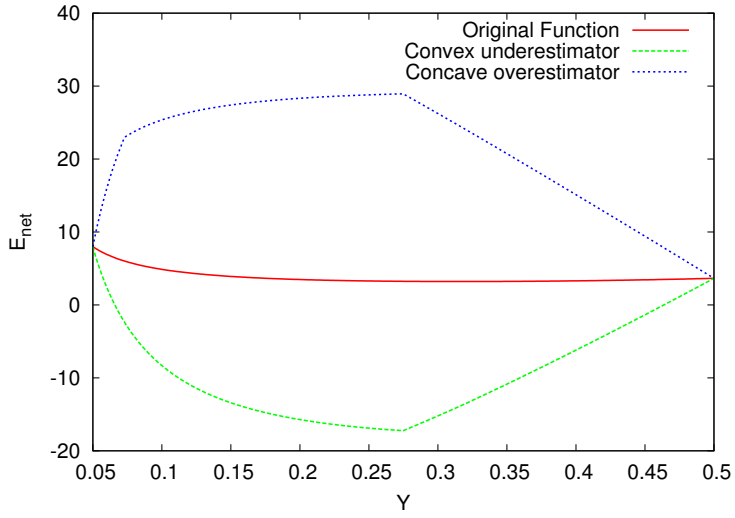

Fig. 13. Convex and concave relaxations of the reverse osmosis example

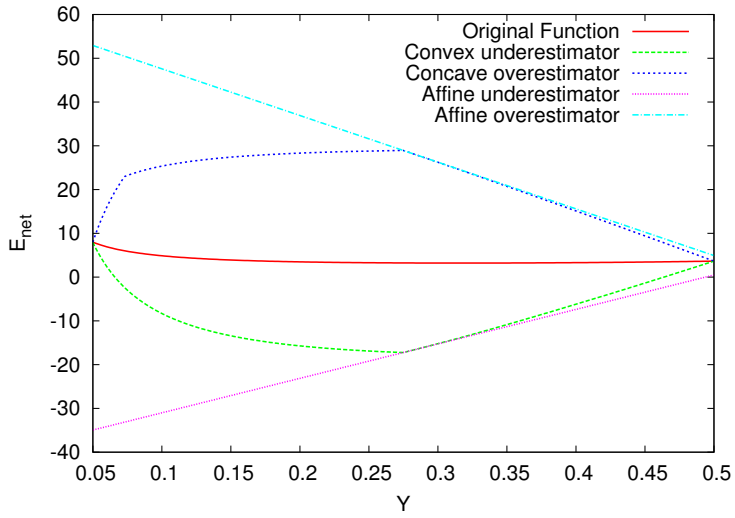

Fig. 14. Convex and concave relaxations and affine under- and overestimators of the reverse osmosis example

This example is representative for typical engineering problems. In comparison to the number of intermediate expressions the degrees of freedom are very few. An example similar to this one is discussed in [4]

5 Conclusion

Both libMC and the current version of modMC are based on the forward mode of AD. The superior performance of modMC is due to the targeted exploitation of Fortran specifics and inlining techniques. Benchmarks show that the overhead created by method calls for overloaded operators is eliminated. Recursive inlining yields additional runtime improvements. Dynamically allocated subgradient storage can be re-used. Standard compiler optimization reduces runtime and memory consumption even further by removing redundant local variables and by performing intraprocedural instead of originally interprocedural optimization for the inlined operators.

References

1. Claire S. Adjiman, Ioannis P. Androulakis, Costas D. Maranas, and Christodoulos A. Floudas. A global optimization method, αBB, for process design. *Computers & Chemical Engineering*, 20(Suppl. A):S419–S424, 1996.
2. I. G. Akrotirianakis and C. A. Floudas. A new class of improved convex underestimators for twice continuously differentiable constrained NLPs. *Journal of Global Optimization*, 30(4):367–390, 2004.
3. S.A. Avlonitis, K. Kouroumbas, and N. Vlachakis. Energy consumption and membrane replacement cost for seawater ro desalination plants. *Desalination*, 157(1-3):151 – 158, 2003.
4. P. I. Barton, B. Chachuat, A. Mitsos, A. Selot, and M. Yunt. Global optimization of algorithms applied to short-term natural gas production planning. Washington D.C., 12th–15th October 2008. INFORMS 2008.
5. Edward E. Baruth, editor. *Water treatment plant design*. McGraw-Hill Professional, New York, USA, third edition, 1997.
6. Dimitri P. Bertsekas. *Nonlinear Programming*. Athena Scientific, Belmont, Massachusetts, Second edition, 1999.
7. Burkhard D. Burow. Mixed language programming. In R. Shellard and T.Nguyen, editors, *Computing in High Energy Physics (CHEP'95)*, pages 610–614, Rio de Janeiro, Brazil, September 1995.
8. Ian B. Cameron and Rodney B. Clemente. Swro with eri's px pressure exchanger device – a global survey. *Desalination*, 221(1-3):136 – 142, 2008.
9. B. Chachuat. `libMC`: A numeric library for McCormick relaxation of factorable functions. Documentation and Source Code available at: http://yoric.mit.edu/libMC/, 2007.
10. Dow Water Solutions, Midland, MI, USA. *FILMTEC Reverse Osmosis Membranes: Technical Manual*.
11. A. Ghobeity and A. Mitsos. Optimal time-dependent operation of seawater reverse osmosis. *In Press: Desaltination*, June 21, 2010.
12. A. Griewank and A. Walther. *Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation (Second Edition)*. SIAM, 2008.
13. J. Hiriart-Urruty and C. Lemaréchal. *Convex Analysis and Minimization Algorithms I*. Springer-Verlag, 1993.
14. Reiner Horst. A general class of branch-and-bound methods in global optimization with some new approaches for concave minimization. *Journal of Optimization Theory and Applications*, 51(2):271–291, 1986.
15. Reiner Horst and Hoang Tuy. *Global Optimization: Deterministic Approaches*. Springer, Berlin, Third edition, 1996.
16. P. Kesavan, R. J. Allgor, E. P. Gatzke, and P. I. Barton. Outer approximation algorithms for separable nonconvex mixed-integer nonlinear programs. *Mathematical Programming*, 100(3):517–535, 2004.
17. G. P. McCormick. Computability of global solutions to factorable nonconvex programs: Part I. Convex underestimating problems. *Mathematical Programming*, 10(2):147–175, 1976.
18. G. P. McCormick. *Nonlinear Programming: Theory, Algorithms and Applications*. John Wiley and Sons, New York, 1983.

19. Giorgio Migliorini and Elena Luzzo. Seawater reverse osmosis plant using the pressure exchanger for energy recovery: a calculation model. *Desalination*, 165:289 – 298, 2004.
20. A. Mitsos, G. M. Bollas, and P. I. Barton. Bilevel optimization formulation for parameter estimation in liquid-liquid phase equilibrium problems. *Chemical Engineering Science*, 64(3):548–559, 2009.
21. A. Mitsos, B. Chachuat, and P. I. Barton. McCormick-based relaxations of algorithms. *SIAM Journal on Optimization*, 20(2):573–601, 2009.
22. U. Naumann and J. Riehme. A differentiation-enabled Fortran 95 compiler. *ACM Transactions on Mathematical Software*, 31(4):458–474, 2005.
23. H. S. Ryoo and N. V. Sahinidis. A branch-and-reduce approach to global optimization. *Journal of Global Optimization*, 8(2):107–138, 1996.
24. N. V. Sahinidis. BARON: A general purpose global optimization software package. *Journal of Global Optimization*, 8:201–205, 1996.
25. Nick Sahinidis and Mohit Tawarmalani. BARON. http://www.gams.com/solvers/baron.pdf, 2005.
26. A. B. Singer, J. W. Taylor, P. I. Barton, and W. H. Green. Global dynamic optimization for parameter estimation in chemical kinetics. *Journal of Physical Chemistry A*, 110(3):971–976, 2006.
27. Adam B. Singer. *Global Dynamic Optimization*. PhD thesis, Massachusetts Institute of Technology http://yoric.mit.edu/download/Reports/SingerThesis.pdf, 2004.
28. M. Tawarmalani and N. V. Sahinidis. *Convexification and Global Optimization in Continuous and Mixed-Integer Nonlinear Programming*. Nonconvex Optimization and its Applications. Kluwer Academic Publishers, Boston, 2002.
29. M. Tawarmalani and N. V. Sahinidis. A polyhedral branch-and-cut approach to global optimization. *Mathematical Programming*, 103(2):225–249, 2005.
30. J. W. Taylor, G. Ehlker, H. H. Carstensen, L. Ruslen, R. W. Field, and W. H. Green. Direct measurement of the fast, reversible addition of oxygen to cyclohexadienyl radicals in nonpolar solvents. *Journal of Physical Chemistry A*, 108(35):7193–7203, 2004.
31. Mark Wilf. *The Guidebook to Membrane Desalination Technology*. Balaban Desalination Publications, L'Aquila, Italy, 2007.

Aachener Informatik-Berichte

This list contains all technical reports published during the past three years. A complete list of reports dating back to 1987 is available from:

`http://aib.informatik.rwth-aachen.de/`

To obtain copies please consult the above URL or send your request to:

Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen,
Email: `biblio@informatik.rwth-aachen.de`

2008-01 *	Fachgruppe Informatik: Jahresbericht 2007
2008-02	Henrik Bohnenkamp, Marielle Stoelinga: Quantitative Testing
2008-03	Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, Harald Zankl: Maximal Termination
2008-04	Uwe Naumann, Jan Riehme: Sensitivity Analysis in Sisyphe with the AD-Enabled NAGWare Fortran Compiler
2008-05	Frank G. Radmacher: An Automata Theoretic Approach to the Theory of Rational Tree Relations
2008-06	Uwe Naumann, Laurent Hascoet, Chris Hill, Paul Hovland, Jan Riehme, Jean Utke: A Framework for Proving Correctness of Adjoint Message Passing Programs
2008-07	Alexander Nyßen, Horst Lichter: The MeDUSA Reference Manual, Second Edition
2008-08	George B. Mertzios, Stavros D. Nikolopoulos: The λ-cluster Problem on Parameterized Interval Graphs
2008-09	George B. Mertzios, Walter Unger: An optimal algorithm for the k-fixed-endpoint path cover on proper interval graphs
2008-10	George B. Mertzios, Walter Unger: Preemptive Scheduling of Equal-Length Jobs in Polynomial Time
2008-11	George B. Mertzios: Fast Convergence of Routing Games with Splittable Flows
2008-12	Joost-Pieter Katoen, Daniel Klink, Martin Leucker, Verena Wolf: Abstraction for stochastic systems by Erlang's method of stages
2008-13	Beatriz Alarcón, Fabian Emmes, Carsten Fuhs, Jürgen Giesl, Raúl Gutiérrez, Salvador Lucas, Peter Schneider-Kamp, René Thiemann: Improving Context-Sensitive Dependency Pairs
2008-14	Bastian Schlich: Model Checking of Software for Microcontrollers
2008-15	Joachim Kneis, Alexander Langer, Peter Rossmanith: A New Algorithm for Finding Trees with Many Leaves
2008-16	Hendrik vom Lehn, Elias Weingärtner and Klaus Wehrle: Comparing recent network simulators: A performance evaluation study
2008-17	Peter Schneider-Kamp: Static Termination Analysis for Prolog using Term Rewriting and SAT Solving
2008-18	Falk Salewski: Empirical Evaluations of Safety-Critical Embedded Systems

2008-19 Dirk Wilking: Empirical Studies for the Application of Agile Methods to Embedded Systems

2009-02 Taolue Chen, Tingting Han, Joost-Pieter Katoen, Alexandru Mereacre: Quantitative Model Checking of Continuous-Time Markov Chains Against Timed Automata Specifications

2009-03 Alexander Nyßen: Model-Based Construction of Embedded Real-Time Software - A Methodology for Small Devices

2009-04 Daniel Klünder: Entwurf eingebetteter Software mit abstrakten Zustandsmaschinen und Business Object Notation

2009-05 George B. Mertzios, Ignasi Sau, Shmuel Zaks: A New Intersection Model and Improved Algorithms for Tolerance Graphs

2009-06 George B. Mertzios, Ignasi Sau, Shmuel Zaks: The Recognition of Tolerance and Bounded Tolerance Graphs is NP-complete

2009-07 Joachim Kneis, Alexander Langer, Peter Rossmanith: Derandomizing Non-uniform Color-Coding I

2009-08 Joachim Kneis, Alexander Langer: Satellites and Mirrors for Solving Independent Set on Sparse Graphs

2009-09 Michael Nett: Implementation of an Automated Proof for an Algorithm Solving the Maximum Independent Set Problem

2009-10 Felix Reidl, Fernando Sánchez Villaamil: Automatic Verification of the Correctness of the Upper Bound of a Maximum Independent Set Algorithm

2009-11 Kyriaki Ioannidou, George B. Mertzios, Stavros D. Nikolopoulos: The Longest Path Problem is Polynomial on Interval Graphs

2009-12 Martin Neuhäußer, Lijun Zhang: Time-Bounded Reachability in Continuous-Time Markov Decision Processes

2009-13 Martin Zimmermann: Time-optimal Winning Strategies for Poset Games

2009-14 Ralf Huuck, Gerwin Klein, Bastian Schlich (eds.): Doctoral Symposium on Systems Software Verification (DS SSV'09)

2009-15 Joost-Pieter Katoen, Daniel Klink, Martin Neuhäußer: Compositional Abstraction for Stochastic Systems

2009-16 George B. Mertzios, Derek G. Corneil: Vertex Splitting and the Recognition of Trapezoid Graphs

2009-17 Carsten Kern: Learning Communicating and Nondeterministic Automata

2009-18 Paul Hänsch, Michaela Slaats, Wolfgang Thomas: Parametrized Regular Infinite Games and Higher-Order Pushdown Strategies

2010-02 Daniel Neider, Christof Löding: Learning Visibly One-Counter Automata in Polynomial Time

2010-03 Holger Krahn: MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering

2010-04 René Wörzberger: Management dynamischer Geschäftsprozesse auf Basis statischer Prozessmanagementsysteme

2010-05 Daniel Retkowitz: Softwareunterstützung für adaptive eHome-Systeme

2010-06 Taolue Chen, Tingting Han, Joost-Pieter Katoen, Alexandru Mereacre: Computing maximum reachability probabilities in Markovian timed automata

2010-07 George B. Mertzios: A New Intersection Model for Multitolerance Graphs, Hierarchy, and Efficient Algorithms

2010-08 Carsten Otto, Marc Brockschmidt, Christian von Essen, Jürgen Giesl: Automated Termination Analysis of Java Bytecode by Term Rewriting

2010-09 George B. Mertzios, Shmuel Zaks: The Structure of the Intersection of Tolerance and Cocomparability Graphs

2010-10 Peter Schneider-Kamp, Jürgen Giesl, Thomas Ströder, Alexander Serebrenik, René Thiemann: Automated Termination Analysis for Logic Programs with Cut

2010-11 Martin Zimmermann: Parametric LTL Games

2010-12 Thomas Ströder, Peter Schneider-Kamp, Jürgen Giesl: Dependency Triples for Improving Termination Analysis of Logic Programs with Cut

2010-13 Ashraf Armoush: Design Patterns for Safety-Critical Embedded Systems

2010-14 Michael Codish, Carsten Fuhs, Jürgen Giesl, Peter Schneider-Kamp: Lazy Abstraction for Size-Change Termination

2010-15 Marc Brockschmidt, Carsten Otto, Christian von Essen, Jürgen Giesl: Termination Graphs for Java Bytecode

2010-16 Christian Berger: Automating Acceptance Tests for Sensor- and Actuator-based Systems on the Example of Autonomous Vehicles

2010-17 Hans Grönniger: Systemmodell-basierte Definition objektbasierter Modellierungssprachen mit semantischen Variationspunkten

2010-18 Ibrahim Armaç: Personalisierte eHomes: Mobilität, Privatsphäre und Sicherheit

2010-19 Felix Reidl: Experimental Evaluation of an Independent Set Algorithm

2010-20 Wladimir Fridman, Christof Löding, Martin Zimmermann: Degrees of Lookahead in Context-free Infinite Games

2011-02 Marc Brockschmidt, Carsten Otto, Jürgen Giesl: Modular Termination Proofs of Recursive Java Bytecode Programs by Term Rewriting

2011-03 Lars Noschinski, Fabian Emmes, Jürgen Giesl: A Dependency Pair Framework for Innermost Complexity Analysis of Term Rewrite Systems

2011-04 Christina Jansen, Jonathan Heinen, Joost-Pieter Katoen, Thomas Noll: A Local Greibach Normal Form for Hyperedge Replacement Grammars

2011-07 Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe: An Operational Semantics for Activity Diagrams using SMV

2011-08 Thomas Ströder, Fabian Emmes, Peter Schneider-Kamp, Jürgen Giesl, Carsten Fuhs: A Linear Operational Semantics for Termination and Complexity Analysis of ISO Prolog

2011-11 Nils Jansen, Erika Ábrahám, Jens Katelaan, Ralf Wimmer, Joost-Pieter Katoen, Bernd Becker: Hierarchical Counterexamples for Discrete-Time Markov Chains

2011-12 Ingo Felscher, Wolfgang Thomas: On Compositional Failure Detection in Structured Transition Systems

2011-13 Michael Förster, Uwe Naumann, Jean Utke: Toward Adjoint OpenMP

2011-14 Daniel Neider, Roman Rabinovich, Martin Zimmermann: Solving Muller Games via Safety Games

2011-16 Niloofar Safiran, Uwe Naumann: Toward Adjoint OpenFOAM

2011-18 Kamal Barakat: Introducing Timers to pi-Calculus

2011-19 Marc Brockschmidt, Thomas Ströder, Carsten Otto, Jürgen Giesl: Automated Detection of Non-Termination and NullPointerExceptions for Java Bytecode

* These reports are only available as a printed version.
Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.