RWTH Aachen

Department of Computer Science
Technical Report

Adjoint Subgradient Calculation for McCormick Relaxations

Markus Beckers, Viktor Mosenkis, Michael Maier and Uwe Naumann

ISSN 0935–3232 · Aachener Informatik-Berichte · AIB-2011-10

RWTH Aachen · Department of Computer Science · October 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/

Adjoint Subgradient Calculation for McCormick Relaxations

Markus Beckers and Viktor Mosenkis and Michael Maier and Uwe Naumann

Lehr- und Forschungsgebiet Informatik 12
Software and Tools for Computational Engineering
RWTH Aachen, Germany
Email: beckers@stce.rwth-aachen.de

Abstract. In [CMN+11] the library modMC was presented which allows the propagation of McCormick relaxations and their corresponding subgradients based on the forward mode of Algorithmic Differentiation (AD). Subgradients are natural extensions of usual derivatives which allow the application of derivative based methods on possibly nondifferentiable convex and concave functions. These subgradients can be computed by AD, a method which allows the computation of derivatives with machine accuracy even for highly complex functions implemented by a computer program. In this article we present the advancement of modMC by reverse mode AD. Reverse mode AD is an adjoint method for the propagation of derivatives which is preferable when scalar functions are considered. We describe the theory behind the application of reverse mode in subgradient propagation as well as the improved library amodMC in detail. The calculated subgradients are used in an deterministic global optimization algorithm which is based on a branch-and-bound method. The improvements gained using Reverse instead of Forward mode AD are illustrated by examples.

1 Motivation & Context

Optimization problems in engineering often have nonconvex objective and constraints and require global optimization algorithms. Deterministic global optimization algorithms based on the branch-and-bound methods solve relaxations of the original program. These are constructed by convex/concave under-/overestimators of the functions involved. One of the alternative methods to construct these estimating functions are McCormick relaxations (see [McC76]). Without auxiliary variables this technique results in nonsmooth estimators, and thus to obtain derivative information, subgradients are needed. These can be calculated using techniques from AD [MCB09],[CMN+11]. This is especially very useful if the functions are given by a long and complex computer program. AD allows the calculation of derivatives, and here additionally the relaxations, with machine accuracy. Hence numerical error based on finite difference approximations are avoided. Two important methods of AD are the forward (or tangent-linear) and the reverse (or adjoint) mode. The choice of the method depends on the dimensionality of the function to be relaxed/differentiated.

The combination of the afore mentioned methods into global optimization was first discussed in [MCB09] using forward mode AD. Enhancements of the implementation were presented in [CMN+11] wherein runtimes were improved by using Fortran specifics and compiler support enabled AD by source transformation. However, [CMN+11] was still lacking a reverse mode implementation, which is a disadvantage since sometimes the number of inputs (optimization variables) is much greater than the number of outputs (objective, constraints). Such an enhancement is given in this paper.

2 Theoretical Development

Let $F : Z \to I\!R$ be a function given on a convex set $Z \subseteq I\!R^n$. Then, a convex (concave) function F^{cv} (F^{cc}) for which $F^{cv}(\mathbf{z}) \leq F(\mathbf{z})$ ($F^{cc}(\mathbf{z}) \geq F(\mathbf{z})$) holds for all $\mathbf{z} \in Z$ is called a *convex (concave) relaxation* of F. As a special case we now observe McCormick relaxations of factorizable functions. In [MCB09] a procedure for propagating *subgradients* of relaxations is presented, which is based on the forward mode of *Algorithmic Differentiation* (AD). Our goal is to extend this procedure to a reverse mode.

Examine first the structure of propagating subgradients with AD methods. The propagation of the convex and concave relaxation of the function

$$F : Z \subseteq I\!R^n \to I\!R, \;\; (z_1, \ldots, z_n) \mapsto y$$

can be considered as the *composition* $g \circ f = \begin{pmatrix} F^{cv} \\ F^{cc} \end{pmatrix}$ of the two functions

$$f : Z \subseteq I\!R^n \to I\!R^{2n}, (z_1, \ldots, z_n) \mapsto (z_1^{\mathrm{cv}}, z_1^{\mathrm{cc}}, \ldots, z_n^{\mathrm{cv}}, z_n^{\mathrm{cc}}) = (z_1, z_1, \ldots, z_n, z_n)$$

and

$$g = (g^{cv}, g^{cc}) : Z^+ \subseteq I\!R^{2n} \to I\!R^2, \;\; (z_1^{\mathrm{cv}}, z_1^{\mathrm{cc}}, \ldots, z_n^{\mathrm{cv}}, z_n^{\mathrm{cc}}) \mapsto (y^{\mathrm{cv}}, y^{\mathrm{cc}}) \quad ,$$

where Z^+ denotes the set $\left\{(z_1, z_1, \ldots, z_n, z_n) \in I\!R^{2n} \mid \mathbf{z} = (z_1, \ldots, z_n) \in Z\right\}$.

This means $y^{cv} = F^{cv}(\mathbf{z}) = g^{cv}(f(\mathbf{z})) = g^{cv}(\mathbf{z}^+)$ and $y^{cc} = F^{cc}(\mathbf{z}) = g^{cc}(f(\mathbf{z})) = g^{cc}(\mathbf{z}^+)$. Here g really represents the simultaneous propagation process of the convex and concave relaxation, for which the duplication f of the variables is needed. (Note that both the convex and concave relaxation of the identity $f(\mathbf{z}) = \mathbf{z}$ are equal to f.)
The following Theorem explains our further proceedings. For a vector $\mathbf{z} = (z_1, \ldots, z_n) \in Z$, $\mathbf{z}^+$ denotes the corresponding vector $\mathbf{z}^+ = (z_1, z_1, \ldots, z_n, z_n) \in Z^+$.

Theorem 1. *Let $g \circ f$ be defined as the above composition, $z \in Z$ and let*

$$s_{g^{cv}}\left(\mathbf{z}^+\right) = \left(\tfrac{\partial y^{cv}}{\partial z_1^{cv}}, \tfrac{\partial y^{cv}}{\partial z_1^{cc}}, \ldots, \tfrac{\partial y^{cv}}{\partial z_n^{cv}}, \tfrac{\partial y^{cv}}{\partial z_n^{cc}}\right)$$

denote a subgradient of the convex relaxation g^{cv} at z^+ and

$$s_{g^{cc}}\left(\mathbf{z}^+\right) = \left(\tfrac{\partial y^{cc}}{\partial z_1^{cv}}, \tfrac{\partial y^{cc}}{\partial z_1^{cc}}, \ldots, \tfrac{\partial y^{cc}}{\partial z_n^{cv}}, \tfrac{\partial y^{cc}}{\partial z_n^{cc}}\right)$$

denote a subgradient of the concave relaxation g^{cc} at z^+.
Then a subgradient of the convex relaxation F^{cv} of F at z is given by

$$s_{F^{cv}}(\mathbf{z}) := \left(\frac{\partial y^{cv}}{\partial z_1^{cv}} + \frac{\partial y^{cv}}{\partial z_1^{cc}}, \ldots, \frac{\partial y^{cv}}{\partial z_n^{cv}} + \frac{\partial y^{cv}}{\partial z_n^{cc}}\right) \quad . \tag{1}$$

Similarly the subgradient of the concave relaxation F^{cc} of F at z is given by

$$s_{F^{cc}}(\mathbf{z}) := \left(\frac{\partial y^{cc}}{\partial z_1^{cv}} + \frac{\partial y^{cc}}{\partial z_1^{cc}}, \ldots, \frac{\partial y^{cc}}{\partial z_n^{cv}} + \frac{\partial y^{cc}}{\partial z_n^{cc}}\right) \quad . \tag{2}$$

Proof. It is easy to see that Z^+ is a convex set and g^{cv} is a convex function for all convex sets $Z \subseteq I\!R^n$. This implies the existence of the above subgradients. According to Definition 1.1.4 on page 165 in [HUL93] we need to show that

$$\langle s_{F^{cv}}(z), d\rangle \leq (F^{cv})'(\mathbf{z}, \mathbf{d}) \quad \forall \mathbf{d} \in I\!R^n \quad , \tag{3}$$

where

$$(F^{cv})'(\mathbf{z}, \mathbf{d}) := \lim_{t \to +0} \frac{F^{cv}(\mathbf{z} + t \cdot \mathbf{d}) - F^{cv}(\mathbf{z})}{t} \quad .$$

We observe for arbitrary $\mathbf{d} \in I\!R^n$:

$$\begin{aligned}
\langle s_{F^{cv}}(\mathbf{z}), \mathbf{d}\rangle &= \left(\frac{\partial y^{\mathrm{cv}}}{\partial z_1^{\mathrm{cv}}} + \frac{\partial y^{\mathrm{cv}}}{\partial z_1^{\mathrm{cc}}}, \ldots, \frac{\partial y^{\mathrm{cv}}}{\partial z_n^{\mathrm{cv}}} + \frac{\partial y^{\mathrm{cv}}}{\partial z_n^{\mathrm{cc}}}\right) \cdot \begin{pmatrix} d_1 \\ \vdots \\ d_n \end{pmatrix} \\
&= d_1 \cdot \left(\frac{\partial y^{\mathrm{cv}}}{\partial z_1^{\mathrm{cv}}} + \frac{\partial y^{\mathrm{cv}}}{\partial z_1^{\mathrm{cc}}}\right) + \cdots + d_n \cdot \left(\frac{\partial y^{\mathrm{cv}}}{\partial z_n^{\mathrm{cv}}} + \frac{\partial y^{\mathrm{cv}}}{\partial z_n^{\mathrm{cc}}}\right) \\
&= d_1 \cdot \frac{\partial y^{\mathrm{cv}}}{\partial z_1^{\mathrm{cv}}} + d_1 \cdot \frac{\partial y^{\mathrm{cv}}}{\partial z_1^{\mathrm{cc}}} + \cdots + d_n \cdot \frac{\partial y^{\mathrm{cv}}}{\partial z_n^{\mathrm{cv}}} + d_n \cdot \frac{\partial y^{\mathrm{cv}}}{\partial z_n^{\mathrm{cc}}} \\
&= \left(\frac{\partial y^{\mathrm{cv}}}{\partial z_1^{\mathrm{cv}}}, \frac{\partial y^{\mathrm{cv}}}{\partial z_1^{\mathrm{cc}}}, \ldots, \frac{\partial y^{\mathrm{cv}}}{\partial z_n^{\mathrm{cv}}}, \frac{\partial y^{\mathrm{cv}}}{\partial z_n^{\mathrm{cc}}}\right) \cdot \begin{pmatrix} d_1 \\ d_1 \\ \vdots \\ d_n \\ d_n \end{pmatrix} \\
&= \langle s_{g^{cv}}(\mathbf{z}^+), \mathbf{d}^+\rangle \\
&\underset{\text{Def. of subgradient}}{\leq} \lim_{t \to +0} \frac{g^{cv}(\mathbf{z}^+ + t \cdot d^+) - g^{cv}(\mathbf{z}^+)}{t}) \\
&= \lim_{t \to +0} \frac{g^{cv}(f(\mathbf{z} + t \cdot \mathbf{d})) - g^{cv}(f(\mathbf{z}))}{t} \\
&= (F^{cv})'(\mathbf{z}, \mathbf{d}) \quad .
\end{aligned}$$

This shows (3) and completes the proof for the convex relaxation. The proof for the concave relaxation is analogue by considering the convex function $-F^{cc}$.

Remark 1. Theorem 1 yields the following view on the propagation of subgradients of convex and concave McCormick relaxations:
If we interprete the matrices

$$Dg := \begin{pmatrix} \frac{\partial y^{\mathrm{cv}}}{\partial z_1^{\mathrm{cv}}} & \frac{\partial y^{\mathrm{cv}}}{\partial z_1^{\mathrm{cc}}} & \cdots & \frac{\partial y^{\mathrm{cv}}}{\partial z_n^{\mathrm{cv}}} & \frac{\partial y^{\mathrm{cv}}}{\partial z_n^{\mathrm{cc}}} \\ & & & & \\ \frac{\partial y^{\mathrm{cc}}}{\partial z_1^{\mathrm{cv}}} & \frac{\partial y^{\mathrm{cc}}}{\partial z_1^{\mathrm{cc}}} & \cdots & \frac{\partial y^{\mathrm{cc}}}{\partial z_n^{\mathrm{cv}}} & \frac{\partial y^{\mathrm{cc}}}{\partial z_n^{\mathrm{cc}}} \end{pmatrix} \in I\!R^{2 \times 2n} \quad . \tag{4}$$

and

$$Df := \begin{pmatrix} 1\,0\,0\ldots 0 \\ 1\,0\,0\ldots 0 \\ 0\,1\,0\ldots 0 \\ 0\,1\,0\ldots 0 \\ 0\,0\,1 \quad\; 0 \\ \vdots\;\vdots\;\vdots\;\ddots\;\vdots \\ 0\,0\,0\;\ddots 1 \\ 0\,0\,0\ldots 1 \end{pmatrix} \in I\!R^{2n\times n}$$

as "Jacobians" of g and f and multiply them we get

$$Dg * Df = \begin{pmatrix} \frac{\partial y^{\mathrm{cv}}}{\partial z_1^{\mathrm{cv}}} + \frac{\partial y^{\mathrm{cv}}}{\partial z_1^{\mathrm{cc}}} \cdots \frac{\partial y^{\mathrm{cv}}}{\partial z_n^{\mathrm{cv}}} + \frac{\partial y^{\mathrm{cv}}}{\partial z_n^{\mathrm{cc}}} \\ \\ \frac{\partial y^{\mathrm{cc}}}{\partial z_1^{\mathrm{cv}}} + \frac{\partial y^{\mathrm{cc}}}{\partial z_1^{\mathrm{cc}}} \cdots \frac{\partial y^{\mathrm{cc}}}{\partial z_n^{\mathrm{cv}}} + \frac{\partial y^{\mathrm{cc}}}{\partial z_n^{\mathrm{cc}}} \end{pmatrix} \in I\!R^{2\times n} \quad . \tag{5}$$

This is a coincidence with the chain rule

$$D[g \circ f](\mathbf{z}) = Dg\,(f(\mathbf{z})) * Df(\mathbf{z})$$

of differential calculus.

Lets call (5) the *sub-Jacobian* of F, where the first row contains the subgradient of the convex underestimator F^{cv} and the second row is given by the subgradient of the concave overestimator F^{cc} of F.

The above approach was used in [MCB09] and [CMN$^+$11] for propagating subgradients in tangent-linear mode. By seeding the tangent vectors $(\mathbf{s}_i^{cv})_{i=1,\ldots,n}$ and $(\mathbf{s}_i^{cc})_{i=1,\ldots,n}$ for the independent variables $(z_i)_{i=1,\ldots,n}$ to the Cartesian basis vectors

$$\mathbf{s}_i^{cv} = \mathbf{s}_i^{cc} = e_i$$

and applying the propagation rules given in [MCB09], the calculation of the sub-Jacobian given in (5) is performed.

We see that by this approach n projections of the matrix Dg have to be calculated. Similar to standard AD, high input dimensions n of the Function F lead to high computational costs for the evaluation of its sub-Jacobian. Since we are only considering scalar functions (leading to two-dimensional output) an adjoint computation of the subgradients is preferable. For applying an adjoint computation one has to be cautious about the seeding. In adjoint mode projections of the *transposed* Jacobian are calculated. In our context this means we derive the product

$$\begin{pmatrix} \frac{\partial y^{\mathrm{cv}}}{\partial z_1^{\mathrm{cv}}} & \frac{\partial y^{\mathrm{cc}}}{\partial z_1^{\mathrm{cv}}} \\ \\ \frac{\partial y^{\mathrm{cv}}}{\partial z_1^{\mathrm{cc}}} & \frac{\partial y^{\mathrm{cc}}}{\partial z_1^{\mathrm{cc}}} \\ \vdots & \vdots \\ \frac{\partial y^{\mathrm{cv}}}{\partial z_n^{\mathrm{cv}}} & \frac{\partial y^{\mathrm{cc}}}{\partial z_n^{\mathrm{cv}}} \\ \\ \frac{\partial y^{\mathrm{cv}}}{\partial z_n^{\mathrm{cc}}} & \frac{\partial y^{\mathrm{cc}}}{\partial z_n^{\mathrm{cc}}} \end{pmatrix} \cdot \bar{\mathbf{v}} \quad .$$

The structure of the transposed matrix implies that there is no possibility of gaining the sub-Jacobian (5) by projecting only one (adjoint) direction vector $\bar{v}$. We have to initialize the two adjoints $\bar{\mathbf{v}}_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and $\bar{\mathbf{v}}_2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$. leading to the adjoint AD propagation of the vectors

$$\mathbf{d}\bar{\mathbf{f}}^{cv} = \begin{pmatrix} \frac{\partial y^{\mathrm{cv}}}{\partial z_1^{\mathrm{cv}}} \\ \frac{\partial y^{\mathrm{cv}}}{\partial z_1^{\mathrm{cc}}} \\ \vdots \\ \frac{\partial y^{\mathrm{cv}}}{\partial z_n^{\mathrm{cv}}} \\ \frac{\partial y^{\mathrm{cv}}}{\partial z_n^{\mathrm{cc}}} \end{pmatrix} \text{ and } \quad \mathbf{d}\bar{\mathbf{f}}^{cc} = \begin{pmatrix} \frac{\partial y^{\mathrm{cc}}}{\partial z_1^{\mathrm{cv}}} \\ \frac{\partial y^{\mathrm{cc}}}{\partial z_1^{\mathrm{cc}}} \\ \vdots \\ \frac{\partial y^{\mathrm{cc}}}{\partial z_n^{\mathrm{cv}}} \\ \frac{\partial y^{\mathrm{cc}}}{\partial z_n^{\mathrm{cc}}} \end{pmatrix} .$$

Summing the right entries in each vector leads to the entries of the desired sub-Jacobian. The following example shows the operations performed in tangent-linear and adjoint mode to calculate the subgradients of a simple function.

Example 1. Consider the evaluation of the subgradients of the convex and concave relaxations of $F(z) = \exp(z_1) \cdot z_1 \cdot z_2$ on the box $[-1, 1] \times [-2, 2]$ at $(z_1, z_2) = (-0.5, 1.3)$. Table 1 demonstrates the evaluation of the relaxations and the corresponding subgradients in the forward mode according to the rules given in [MCB09].

Initialization	
$v_1^L = -1$ $\boldsymbol{v_1^{\text{cv}}} = z = \mathbf{-0.5}$ $\boldsymbol{s_1^{\text{cv}}} = (1, 0)$	$v_1^U = 1$ $\boldsymbol{v_1^{\text{cc}}} = z = \mathbf{-0.5}$ $\boldsymbol{s_1^{\text{cc}}} = (1, 0)$
$v_2^L = -2$ $\boldsymbol{v_2^{\text{cv}}} = z = \mathbf{1.3}$ $\boldsymbol{s_2^{\text{cv}}} = \mathbf{(0, 1)}$	$v_2^U = 2$ $\boldsymbol{v_2^{\text{cc}}} = z = \mathbf{1.3}$ $\boldsymbol{s_2^{\text{cc}}} = \mathbf{(0, 1)}$
$\boldsymbol{v_3 = \exp(v_1)}$	
$v_3^L = \exp(v_1^L) = \exp(-1) = 0.37$ $\boldsymbol{v_3^{\text{cv}}} = \exp(\text{mid}(v_1^{\text{cv}}, v_1^{\text{cc}}, x_{\min})$ $= \exp(\text{mid}(-0.5, -0.5, -1))$ $= \exp(-0.5) \approx \mathbf{0.61}$ $\boldsymbol{s_3^{\text{cv}}} = \begin{cases} 0 & \text{if } v_1^{\text{cv}} \le x_{\min} \le v_1^{\text{cc}}, \\ \sigma_3^{uo}\, s_1^{\text{cc}} & \text{if } v_1^{\text{cc}} < x_{\min}, \\ \sigma_3^{uu}\, s_1^{\text{cv}} & \text{if } x_{\min} < v_1^{\text{cv}}. \end{cases}$ $= \sigma_3^{uu}\, s_1^{\text{cv}}$ $= \exp(v_1^{\text{cv}}) * (1, 0)$ $= \exp(-0.5) * (1, 0) \approx \mathbf{(0.61, 0)}$	$v_3^U = \exp(v_1^U) = \exp(1) \approx 2.72$ $\boldsymbol{v_3^{\text{cc}}} = \frac{e^{v_1^U} - e^{v_1^L}}{v_1^U - v_1^L} \text{mid}(v_1^{\text{cv}}, v_1^{\text{cc}}, x_{\max})$ $+ \frac{v_1^U\, e^{v_1^L} - v_1^L\, e^{v_1^U}}{v_1^U - v_1^L}$ $= \frac{e^1 - e^{-1}}{2}(-0.5) + \frac{1\, e^{-1} - (-1)\, e^1}{2}$ $\approx \mathbf{0.96}$ $\boldsymbol{s_3^{\text{cc}}} = \begin{cases} 0 & \text{if } v_1^{\text{cv}} \le x_{\max} \le v_1^{\text{cc}}, \\ \sigma_3^{oo}\, s_1^{\text{cc}} & \text{if } v_1^{\text{cc}} < x_{\max}, \\ \sigma_3^{ou}\, s_1^{\text{cv}} & \text{if } x_{\min} < v_1^{\text{cv}}. \end{cases}$ $= \sigma_3^{oo}\, s_1^{\text{cc}}$ $= \frac{e^{v_1^U} - e^{v_1^L}}{v_1^U - v_1^L}(1, 0)$ $= \frac{e^1 - e^{-1}}{2} \approx \mathbf{(1.18, 0)}$
$\boldsymbol{v_4 = v_3 \cdot v_1}$	
$v_4^L = -2.72$ $\alpha_1 = \min(v_1^L \cdot v_3^{cv}, v_1^L \cdot v_3^{cc})$ $= \min(-0.61, -0.96) = -0.96$ $\beta_1 = \min(v_1^U \cdot v_3^{cv}, v_1^U \cdot v_3^{cc})$ $= \min(0.61, 0.96) = 0.61$ $\gamma_1 = \max(v_1^L \cdot v_3^{cv}, v_1^L \cdot v_3^{cc})$ $= \max(-0.61, -0.96) = -0.61$ $\delta_1 = \max(v_1^U \cdot v_3^{cv}, v_1^U \cdot v_3^{cc})$ $= \max(0.61, 0.96) = 0.96$ $\boldsymbol{v_4^{cv}} = \max(\alpha_1 + \alpha_2 - v_3^L \cdot v_1^L, \beta_1 + \beta_2 - v_3^U \cdot v_1^U)$ $= \max(-0.78, -3.47)$ $= \mathbf{-0.78}$ $s^{\alpha_1} = v_1^L \cdot s_3^{cc}$ $= (-1, 18, 0)$ $s^{\beta_1} = v_1^U \cdot s_3^{cv}$ $= (0.61, 0)$ $s^{\gamma_1} = v_1^L \cdot s_3^{cv}$ $= (-0.61, 0)$ $s^{\delta_1} = v_1^U \cdot s_3^{cc}$ $= (1.18, 0)$ $\boldsymbol{s_4^{cv}} = s^{\alpha_1} + s^{\alpha_2}$ $= \mathbf{(-0.81, 0)}$	$v_4^U = 2.72$ $\alpha_2 = \min(v_3^L \cdot v_1^{cv}, v_3^L \cdot v_1^{cc})$ $= \min(-0.19, -0.19) = -0.19$ $\beta_2 = \min(v_3^U \cdot v_1^{cv}, v_3^U \cdot v_1^{cc})$ $= \min(-1.36, -1.36) = -1.36$ $\gamma_2 = \max(v_3^U \cdot v_1^{cv}, v_3^U \cdot v_1^{cc})$ $= \max(-1.36, -1.36) = -1.36$ $\delta_2 = \max(v_3^L \cdot v_1^{cv}, v_3^L \cdot v_1^{cc})$ $= \max(-0.19, -0.19) = -0.19$ $\boldsymbol{v_4^{cc}} = \min(\gamma_1 + \gamma_2 - v_3^U \cdot v_1^L, \delta_1 + \delta_2 - v_3^L \cdot v_1^U)$ $= \min(0.75, 0.4)$ $= \mathbf{0.4}$ $s^{\alpha_2} = v_3^L \cdot s_1^{cv}$ $= (0.37, 0)$ $s^{\beta_2} = v_3^U \cdot s_1^{cv}$ $= (2.72, 0)$ $s^{\gamma_2} = v_3^U \cdot s_1^{cc}$ $= (2.72, 0)$ $s^{\delta_2} = v_3^L \cdot s_1^{cc}$ $= (0.37, 0)$ $\boldsymbol{s_4^{cc}} = s^{\delta_1} + s^{\delta_2}$ $= \mathbf{(1.55, 0)}$

$\boldsymbol{v_5 = v_4 \cdot v_2}$	
$v_5^L = -5.44$	$v_5^U = 5.44$
$\alpha_1 = \min(v_2^L \cdot v_4^{cv}, v_2^L \cdot v_4^{cc})$ $= \min(1.56, -0.8) = -0.8$ $\beta_1 = \min(v_2^U \cdot v_4^{cv}, v_2^U \cdot v_4^{cc})$ $= \min(-1.56, 0.8) = -1.56$ $\gamma_1 = \max(v_2^L \cdot v_4^{cv}, v_2^L \cdot v_4^{cc})$ $= \max(1.56, -0.8) = 1.56$ $\delta_1 = \max(v_2^U \cdot v_4^{cv}, v_2^U \cdot v_4^{cc})$ $= \max(-1.56, 0.8) = 0.8$	$\alpha_2 = \min(v_4^L \cdot v_2^{cv}, v_4^L \cdot v_2^{cc})$ $= \min(-3.54, -3.54) = -3.54$ $\beta_2 = \min(v_4^U \cdot v_2^{cv}, v_4^U \cdot v_2^{cc})$ $= \min(3.54, 3.54) = 3.54$ $\gamma_2 = \max(v_4^U \cdot v_2^{cv}, v_4^U \cdot v_2^{cc})$ $= \max(3.54, 3.54) = 3.54$ $\delta_2 = \max(v_4^L \cdot v_2^{cv}, v_4^L \cdot v_2^{cc})$ $= \max(-3.54, -3.54) = -3.54$
$\boldsymbol{v_5^{cv}} = \max(\alpha_1 + \alpha_2 - v_4^L \cdot v_2^L, \beta_1 + \beta_2 - v_4^U \cdot v_2^U)$ $= \max(-9.78, -3.46)$ $= \mathbf{-3.46}$	$\boldsymbol{v_5^{cc}} = \min(\gamma_1 + \gamma_2 - v_4^U \cdot v_2^L, \delta_1 + \delta_2 - v_4^L \cdot v_2^U)$ $= \min(10.54, 2.7)$ $= \mathbf{2.7}$
$s^{\alpha_1} = v_2^L \cdot s_4^{cc}$ $= (-3.1, 0)$ $s^{\beta_1} = v_2^U \cdot s_4^{cv}$ $= (-1.62, 0)$ $s^{\gamma_1} = v_2^L \cdot s_4^4$ $= (1.62, 0)$ $s^{\delta_1} = v_2^U \cdot s_4^{cc}$ $= (3.1, 0)$	$s^{\alpha_2} = v_4^L \cdot s_2^{cv}$ $= (0, -2.72)$ $s^{\beta_2} = v_4^U \cdot s_2^{cv}$ $= (0, 2.72)$ $s^{\gamma_2} = v_4^U \cdot s_2^{cc}$ $= (0, 2.72)$ $s^{\delta_2} = v_4^L \cdot s_2^{cc}$ $= (0, -2.72)$
$\boldsymbol{s_5^{cv}} = s^{\beta_1} + s^{\beta_2}$ $= \mathbf{(-1.62, 2.72)}$	$\boldsymbol{s_5^{cc}} = s^{\delta_1} + s^{\delta_2}$ $= \mathbf{(3.1, -2.72)}$

Table 1: Forward Propagation of Relaxations and Subgradients

Hence the values of the relaxations of F are given by $F^{vc} = -3.46$ and $F^{cc} = 2.7$. The corresponding subgradients are shown to be $s_{F^{cv}} = (-1.62, 2.72)$ and $s_{F^{cc}} = (3.1, -2.72)$.

In the following we show how these subgradients are computed in reverse mode. Here the "Jacobian" (4) of the (extended) function g of F from page 5 is given by

$$Dg := \begin{pmatrix} \frac{\partial F^{cv}}{\partial z_1^{cv}} & \frac{\partial F^{cv}}{\partial z_1^{cc}} & \frac{\partial F^{cv}}{\partial z_2^{cv}} & \frac{\partial F^{cv}}{\partial z_2^{cc}} \\ \\ \frac{\partial F^{cc}}{\partial z_1^{cv}} & \frac{\partial F^{cc}}{\partial z_1^{cc}} & \frac{\partial F^{cc}}{\partial z_2^{cv}} & \frac{\partial F^{cc}}{\partial z_2^{cc}} \end{pmatrix} \in I\!R^{2\times 4}.$$

In reverse mode we calculate the product

$$Df_2^t * \bar{v} = \begin{pmatrix} \frac{\partial F^{cv}}{\partial z_1^{cv}} & \frac{\partial F^{cc}}{\partial z_1^{cv}} \\ \\ \frac{\partial F^{cv}}{\partial z_1^{cc}} & \frac{\partial F^{cc}}{\partial z_1^{cc}} \\ \\ \frac{\partial F^{cv}}{\partial z_2^{cv}} & \frac{\partial F^{cc}}{\partial z_2^{cv}} \\ \\ \frac{\partial F^{cv}}{\partial z_2^{cc}} & \frac{\partial F^{cc}}{\partial z_2^{cc}} \end{pmatrix} * \bar{v}$$

of the *transposed* sub-Jacobian with an adjoint vector $\bar{v}$. So in order to obtain the subgradients given by (1) and (2) in this case we have to, simultaneously, propagate two adjoint vectors $\bar{v}_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and $\bar{v}_2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$. This leads to the two vectors $d\bar{f}^{cv} = \left(\frac{\partial F^{cv}}{\partial z_1^{cv}} \ \frac{\partial F^{cv}}{\partial z_1^{cc}} \ \frac{\partial F^{cv}}{\partial z_2^{cv}} \ \frac{\partial F^{cv}}{\partial z_2^{cc}} \right)^t$ and $d\bar{f}^{cv} = \left(\frac{\partial F^{cv}}{\partial z_1^{cv}} \ \frac{\partial F^{cv}}{\partial z_1^{cc}} \ \frac{\partial F^{cv}}{\partial z_2^{cv}} \ \frac{\partial F^{cv}}{\partial z_2^{cc}} \right)^t$. Summing the right entries of each vector leads to the desired subgradients s_u and s_o. The reverse sweep of our example function is given in Table 2. First all variables for the adjoint vectors, that are $\overline{s}_i^{cv_cv}$, $\overline{s}_i^{cv_cc}$, $\overline{s}_i^{cc_cv}$ and $\overline{s}_i^{cc_cc}$ for $i = 1, \ldots, 5$, are set to zero. The adjoint computation of the subgradients can be seen in Table 2.

Initialization of Adjoints	
$\overline{s}_5^{cv_cv} = 1$	$\overline{s}_5^{cc_cv} = 0$
$\overline{s}_5^{cv_cc} = 0$	$\overline{s}_5^{cc_cc} = 1$
$\boldsymbol{v_5 = v_4 \cdot v_2}$	
$\overline{s}_4^{cv_cv} = \overline{s}_4^{cv_cv} + \overline{s}_5^{cv_cv} \cdot \text{tmpcv}(2)$	$\overline{s}_4^{cc_cv} = \overline{s}_4^{cc_cv} + \overline{s}_5^{cc_cv} \cdot \text{tmpcv}(2)$
$\quad + \overline{s}_5^{cv_cc} \cdot \text{tmpcc}(2)$	$\quad + \overline{s}_5^{cc_cc} \cdot \text{tmpcc}(2)$
$\quad = 0 + 1 \cdot 2 + 0 \cdot 0 = 2$	$\quad = 0 + 0 \cdot 2 + 1 \cdot 0 = 0$
$\overline{s}_4^{cv_cc} = \overline{s}_4^{cv_cc} + \overline{s}_5^{cv_cc} \cdot \text{tmpcc}(1)$	$\overline{s}_4^{cc_cc} = \overline{s}_4^{cc_cc} + \overline{s}_5^{cc_cc} \cdot \text{tmpcc}(1)$
$\quad + \overline{s}_5^{cv_cv} \cdot \text{tmpcv}(1)$	$\quad + \overline{s}_5^{cc_cv} \cdot \text{tmpcv}(1)$
$\quad = 0 + 0 \cdot 2 + 1 \cdot 0 = 0$	$\quad = 0 + 1 \cdot 2 + 0 \cdot 0 = 2$
$\overline{s}_2^{cv_cv} = \overline{s}_2^{cv_cv} + \overline{s}_5^{cv_cv} \cdot \text{tmpcv}(4)$	$\overline{s}_2^{cc_cv} = \overline{s}_2^{cc_cv} + \overline{s}_5^{cc_cv} \cdot \text{tmpcv}(4)$
$\quad + \overline{s}_5^{cv_cc} \cdot \text{tmpcc}(4)$	$\quad + \overline{s}_5^{cc_cc} \cdot \text{tmpcc}(4)$
$\quad = 0 + 1 \cdot 2.72 + 0 \cdot (-2.72) = 2.72$	$\quad = 0 + 0 \cdot 2.72 + 1 \cdot (-2.72) = -2.72$
$\overline{s}_2^{cv_cc} = \overline{s}_2^{cv_cc} + \overline{s}_5^{cv_cc} \cdot \text{tmpcc}(3)$	$\overline{s}_2^{cc_cc} = \overline{s}_2^{cc_cc} + \overline{s}_5^{cc_cc} \cdot \text{tmpcc}(3)$
$\quad + \overline{s}_5^{cv_cv} \cdot \text{tmpcv}(3)$	$\quad + \overline{s}_5^{cc_cv} \cdot \text{tmpcv}(3)$
$\quad = 0 + 0 \cdot 0 + 1 \cdot 0 = 0$	$\quad = 0 + 1 \cdot 0 + 0 \cdot 0 = 0$
$\boldsymbol{v_4 = v_3 \cdot v_1}$	
$\overline{s}_3^{cv_cv} = \overline{s}_3^{cv_cv} + \overline{s}_4^{cv_cv} \cdot \text{tmpcv}(2)$	$\overline{s}_3^{cc_cv} = \overline{s}_3^{cc_cv} + \overline{s}_4^{cc_cv} \cdot \text{tmpcv}(2)$
$\quad + \overline{s}_4^{cv_cc} \cdot \text{tmpcc}(2)$	$\quad + \overline{s}_4^{cc_cc} \cdot \text{tmpcc}(2)$
$\quad = 0 + 2 \cdot 0 + 0 \cdot 0 = 0$	$\quad = 0 + 0 \cdot 0 + 2 \cdot 0 = 0$
$\overline{s}_3^{cv_cc} = \overline{s}_3^{cv_cc} + \overline{s}_4^{cv_cc} \cdot \text{tmpcc}(1)$	$\overline{s}_3^{cc_cc} = \overline{s}_3^{cc_cc} + \overline{s}_4^{cc_cc} \cdot \text{tmpcc}(1)$
$\quad + \overline{s}_4^{cv_cv} \cdot \text{tmpcv}(1)$	$\quad + \overline{s}_4^{cc_cv} \cdot \text{tmpcv}(1)$
$\quad = 0 + 0 \cdot 1 + 2 \cdot (-1) = -2$	$\quad = 0 + 2 \cdot 1 + 0 \cdot (-1) = 2$
$\overline{s}_1^{cv_cv} = \overline{s}_1^{cv_cv} + \overline{s}_4^{cv_cv} \cdot \text{tmpcv}(4)$	$\overline{s}_1^{cc_cv} = \overline{s}_1^{cc_cv} + \overline{s}_4^{cc_cv} \cdot \text{tmpcv}(4)$
$\quad + \overline{s}_4^{cv_cc} \cdot \text{tmpcc}(4)$	$\quad + \overline{s}_4^{cc_cc} \cdot \text{tmpcc}(4)$
$\quad = 0 + 2 \cdot 0.37 + 0 \cdot 0 = 0.74$	$\quad = 0 + 0 \cdot 0.37 + 2 \cdot 0 = 0$
$\overline{s}_1^{cv_cc} = \overline{s}_1^{cv_cc} + \overline{s}_4^{cv_cc} \cdot \text{tmpcc}(3)$	$\overline{s}_1^{cc_cc} = \overline{s}_1^{cc_cc} + \overline{s}_4^{cc_cc} \cdot \text{tmpcc}(3)$
$\quad + \overline{s}_4^{cv_cv} \cdot \text{tmpcv}(3)$	$\quad + \overline{s}_4^{cc_cv} \cdot \text{tmpcv}(3)$
$\quad = 0 + 0 \cdot 0.37 + 2 \cdot 0 = 0$	$\quad = 0 + 2 \cdot 0.37 + 0 \cdot 0 = 0.74$
$\boldsymbol{v_3 = \exp(v_1)}$	
$\overline{s}_1^{cv_cv} = \overline{s}_1^{cv_cv} + \overline{s}_3^{cv_cv} \cdot \text{tmpcv}(2)$	$\overline{s}_1^{cc_cv} = \overline{s}_1^{cc_cv} + \overline{s}_3^{cc_cv} \cdot \text{tmpcv}(2)$
$\quad = 0.74 + 0 \cdot 0.61 = 0.74$	$\quad = 0 + 0 \cdot 0.61 = 0$
$\overline{s}_1^{cv_cc} = \overline{s}_1^{cv_cc} + \overline{s}_3^{cv_cc} \cdot \text{tmpcc}(2)$	$\overline{s}_1^{cc_cc} = \overline{s}_1^{cc_cc} + \overline{s}_3^{cc_cc} \cdot \text{tmpcc}(2)$
$\quad = 0 + (-2) \cdot 1.18 = -2.36$	$\quad = 0.74 + 2 \cdot 1.18 = 3.1$
Subgradients	
$\overline{\boldsymbol{s}}^{\boldsymbol{cv}} = \begin{pmatrix} \overline{s}_1^{cv_cv} + \overline{s}_1^{cv_cc} \\ \overline{s}_2^{cv_cv} + \overline{s}_2^{cv_cc} \end{pmatrix}$	$\overline{\boldsymbol{s}}^{\boldsymbol{cc}} = \begin{pmatrix} \overline{s}_1^{cc_cv} + \overline{s}_1^{cc_cc} \\ \overline{s}_2^{cc_cv} + \overline{s}_2^{cc_cc} \end{pmatrix}$
$= \begin{pmatrix} 0.74 - 2.36 \\ 2.72 + 0 \end{pmatrix}$	$= \begin{pmatrix} 0 + 3.1 \\ -2.72 + 0 \end{pmatrix}$
$= \begin{pmatrix} \mathbf{-1.62} \\ \mathbf{2.72} \end{pmatrix}$	$= \begin{pmatrix} \mathbf{3.1} \\ \mathbf{-2.72} \end{pmatrix}$

Table 2. Reverse Propagation of Subgradients

3 amodMC: Adjoint McCormick Relaxations and Subgradients by Overloading in Fortran

This section covers amodMC, an adjoint variant of the tangent-linear modMC Fortran library described in [CMN+11]. amodMC calculates McCormick relaxations by overloading arithmetic operators and intrinsic functions for the *derived type* `McCormick`. Subgradients are calculated during the so-called *reverse sweep*, an evaluation phase of the amodMC library, manually invoked by the user at the end of the desired user function. Arguments required for subgradient calculation

are automatically stored on an internal tape during overloaded operator and overloaded intrinsic function calls. The reverse sweep evaluates tape entries from the end of the stored tape to the first entry, hence reverses the original program flow. Support subroutines are provided for type initialization, error handling, and to produce human-readable type instance and tape printouts for debugging and documentation.

3.1 Compilation

amodMC 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/amodMC/`. The build process complies with the GNU Code Standards[2] and is based on the GNU Autotools[3]. To compile amodMC upon successful download, the following steps are required:

1. Extract package contents using the `tar` command-line utility:

   ```
   tar -xvj amodmc-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 amodmc-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 amodMC. Binaries are placed inside the `src` directory:

   ```
   make
   ```

   The following files are created: `libamodmc.a` contains the binary object code and `mccormick.mod` contains the Fortran-specific interface description of amodMC. It is possible to compile a Fortran project that uses amodMC without actually installing amodMC. The file `mccormick.mod` needs to be placed inside the project directory. The static library `libamodmc.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 amodMC; `exit` closes the previously spawned subshell. If the `sudo` command is available, then the previous three lines can be combined into

   ```
   sudo make install
   ```

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

After installation amodMC 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 -lamodmc -I/usr/include/amodmc
```

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

3.2 Usage

Given a function implemented in Fortran, McCormick relaxations are easily obtained by replacing floating-point types for optimization, output and intermediate variables with the new derived type `McCormick`. Initialization of such variables requires calls to the `mccormick_init()` subroutine, setting the variable bounds, as well as a unique index for each variable. This procedure has to be repeated for each function or subroutine in the call tree of the main function.

```
1  program ex1
2    implicit none
3
4    double precision :: x1 = 4.5d0, x2 = 3.0d0
5    double precision :: y
6
7    ! Function.
8    y = sqrt(x1 ** 2 + exp(x2) / x1)
9
10   print *, y
11 end program
```

Listing 1.1. Sample user function with parameters `x1` and `x2` and result variable `y`

```
1    4.97126268161292
```

Listing 1.2. Output of example program Listing 1.1

No further modifications to the original source code are necessary. At the end of the main function, the convex underestimator and concave overestimator of each objective variable are accessible via the component values `cv` and `cc`. This is equally achieved using modMC. Confer to Listing 1.3 for a McCormick-enabled variant of the sample function in Listing 1.1.

```
1  program ex1_mc
2    use mccormick
3    implicit none
4
5    type(mccormick_type) :: x1, x2
6    type(mccormick_type) :: y
7
8    ! Type initialisation including boundaries.
9    call mccormick_init (x1, 2.0d0, 7.0d0, 4.5d0, 0)
10   call mccormick_init (x2, 1.0d0, 5.0d0, 3.0d0, 0)
```

```
11
12    ! Function.
13    y = sqrt(x1 ** 2 + exp(x2) / x1)
14
15    ! Get the convex underestimator.
16    write (*,*) y%cv
17    ! Get the concave overestimator.
18    write (*,*) y%cc
19  end program
```

Listing 1.3. Calculation of McCormick relaxations of previous sample function; this is equally achived using both modMC and amodMC

```
1     3.53077358673985
2     7.98733076586861
```

Listing 1.4. Output of example program Listing 1.3

To retrieve subgradients, amodMC needs information on whether a variable is *dependent* (i.e. evaluates the objective function or a constraint) or *independent* (i.e. is an optimization parameter). This is achieved by calling the subroutines `mccormick_dep()` and `mccormick_indep_set()` for each independent and dependent variable, respectively. These subroutines create specific tape entries for the variable and connect it with its tape entry. Registration of independent variables is required prior to the function code, registration of dependent variables afterwards. The latter additionally allows to provide a scaling factor for subgradients and defaults to `1.0d0`.
The reverse sweep must be manually initiated by calling the provided library function `mccormick_tape_interpret()` after function code and dependent variable registration. Once this process is finished, subgradients with respect to an independent (optimization) variable can be retrieved by calling `mccormick_indep_get()` on the selected independent variable. The function either extracts subgradients for the convex underestimator, concave overestimator or both: The `subcv` and `subcc` parameters denote destinations, are both optional and can be provided as necessary. Listing 1.5 extends the previous shown example function to retrieve subgradients calculated by amodMC.

```
1   program ex1_ad
2     use mccormick
3     implicit none
4
5     ! Type changes.
6     type(mccormick_type) :: x1, x2
7     type(mccormick_type) :: y
8     double precision scc, scv
9
10    ! Initialisation of optimisation parameters.
11    call mccormick_init (x1, 2.0d0, 7.0d0, 4.5d0, 0)
12    call mccormick_init (x2, 1.0d0, 5.0d0, 3.0d0, 0)
13
14    ! Setup tape with static size (no. of maximum entries).
15    call mccormick_tape_init (64)
16
17    ! Registration of independent variable(s).
18    call mccormick_indep_set (x1)
19    call mccormick_indep_set (x2)
```

```
20
21    ! Unmodified original program code.
22    y = sqrt(x1 ** 2 + exp(x2) / x1)
23
24    ! Registration of dependent variable(s).
25    call mccormick_dep (y, 1.0d0)
26
27    ! Initiate tape interpretation.
28    call mccormick_tape_interpret
29
30    ! Get the convex underestimator.
31    write (*,*) y%cv
32    ! Get the concave overestimator.
33    write (*,*) y%cc
34    ! Get the subgradients with respect to the 1st variable (x1).
35    call mccormick_indep_get (x1, subcv = scv, subcc = scc)
36    write (*,*) "subgradients␣for␣x1␣=␣", scv, scc
37    ! Get the subgradients with respect to the 2nd variable (x2).
38    call mccormick_indep_get (x2, subcv = scv, subcc = scc)
39    write (*,*) "subgradients␣for␣x2␣=␣", scv, scc
40  end program
```

Listing 1.5. `McCormick`-enabled variant based on amodMC subgradient calculation of sample function

```
1     3.53077358673985
2     7.98733076586861
3   subgradients for x1 = 0.671919627773534   -1.254705216708113E-002
4   subgradients for x2 = 0.217463600198994    4.078053471330346E-002
```

Listing 1.6. Output of example program Listing 1.5

More complex functions include calls to other functions and procedures, which themselves may contain further function calls. Type changes and initializations must be applied to all functions contained in the call tree created by tracing function calls. Registration of independent and dependent variables is only required once in the main function. The same holds true for tape interpretation and access to subgradients.

3.3 McCormick Implementation

The amodMC library defines a new derived type `McCormick` shown in Listing 1.7. This structure is similar to the `McCormick` type used by modMC but differs in components used for subgradient calculation: Instead of storing subgradient arrays that are propagated through the program flow (cf. [CMN+11]), each `McCormick` instance stores virtual tape addresses. 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 the virtual tape addresses `j` and `k`.

```
type mccormick_type
    sequence
    ! lower bound
    double precision l
    ! upper bound
    double precision u
    ! convex underestimator
    double precision cv
```

```
    ! concave overestimator
    double precision cc
    ! virtual tape address
    integer :: j = 0
    ! virtual tape address used during independent registration
    integer :: k = 0
end type mccormick_type
```

Listing 1.7. amodMC `McCormick` data type structure

The primary tape address `j` points to the tape entry stored with the instantiation of the instance itself. `k` points to the tape entry of an independent registration, hence is only set for independent variables (parameters, optimization variables) and zero otherwise. The latter is required to retain the connection between an independent variable and its registration even if a new value is assigned to the variable.

3.4 The Tape

The internal tape is a global array of `tape_entry` derived type data entries as shown in Listing 1.8. A tape entry is created by amodMC on each overloaded operator and overloaded intrinsic function call, including arithmetics with at least one argument of `McCormick` type, assignments to and independent plus dependent registration of `McCormick` instances. Each operation is identified by an integer operation code, short opcode, stored in `op`. Currently operators and intrinsic functions require at most 2 arguments: The `p1` and `p2` components store tape addresses of up to two arguments, zero otherwise. Copies of the current convex underestimator and concave overestimator (at the time of the overloaded operator or function execution) are stored in `cv` and `cc`, respectively, and are required to calculate subgradients during tape interpretation. Two arrays of temporaries, `xcv` and `xcc`, store additional operator-specific values required for tape interpretation. Their size is dynamically set by the overloaded operator implementation and usage differs vastly. During the reversed tape interpretation sweep, so-called *adjoints* are propagated through the tape originating from dependent variables (including the scaling factor used) to their independent destinations: `adcvcv` stores $\partial y^{\mathrm{cv}}/\partial z^{\mathrm{cv}}$, `adcvcc` stores $\partial y^{\mathrm{cv}}/\partial z^{\mathrm{cc}}$, `adcccv` stores $\partial y^{\mathrm{cc}}/\partial z^{\mathrm{cv}}$ and `adcccc` stores $\partial y^{\mathrm{cc}}/\partial z^{\mathrm{cc}}$. This corresponds directly to the adjoints and their propagation presented in Section 2.

```
type tape_entry
    integer :: op = 0   ! opcode
    integer :: p1 = 0   ! tape address of 1st argument
    integer :: p2 = 0   ! tape address of 2nd argument

    ! actual cv and cc values
    double precision :: cv = 0.0d0
    double precision :: cc = 0.0d0

    ! temporaries used for adjoint computation;
    ! usage depends on opcode
    double precision, dimension(:), allocatable :: xcv
    double precision, dimension(:), allocatable :: xcc

    ! adjoints
    double precision :: adcvcv = 0.0d0  ! dy.cv/dx.cv
    double precision :: adcvcc = 0.0d0  ! dy.cv/dx.cc
```

```
    double precision :: adcccv = 0.0d0  ! dy.cc/dx.cv
    double precision :: adcccc = 0.0d0  ! dy.cc/dx.cc
end type tape_entry
```

Listing 1.8. amodMC tape entry data type structure

Table 3 summarizes all tape operation codes currently used. Opcodes 0, 22 and higher should never occur and indicate a tape error: If a 0 opcode is encountered, a tape entry initialization is missing; opcodes 22 and higher are not yet defined, indicating a probably incomplete tape extension or damaged data. The letter "a" denotes a `McCormick` typed argument, "r" either a real or double precision value and "i" an integer. Since real arguments are stored as double precision temporaries in tape entries, real and double precision arguments are treated equally from the tape's point of view. The Arrhenius formula represents

Display	#	Opcode Description
`? undef`	0	Error: Undefined operand (not properly initialized).
`=(a,a)`	1	`McCormick` assignment.
`+(a,a)`	2	`McCormick` addition.
`-(a,a)`	3	`McCormick` subtraction.
`*(a,a)`	4	`McCormick` multiplication.
`sin(a)`	5	`McCormick` sine.
`exp(a)`	6	`McCormick` exponential.
`indep`	7	Independent registration.
`dep`	8	Dependent registration.
`-(a)`	9	Unary `McCormick` minus.
`log(a)`	10	Natural `McCormick` logarithm.
`*(d,a)`	11	Double value times `McCormick` instance.
`sqrt(a)`	12	`McCormick` square root.
`abs(a)`	13	Absolute value of `McCormick` instance.
`.inv.a`	14	Inverse of `McCormick` instance.
`-(r,a)`	15	Real or double value minus `McCormick` instance.
`=(a,r)`	16	Real or double value assigned to `McCormick` instance.
`pow(a,i)`	17	`McCormick` exponentiation with integer exponent.
`+(r,a)`	18	Real or double value plus `McCormick` instance or vice-versa.
`-(a,r)`	19	`McCormick` instance minus real or double value.
`arh(a)`	20	`McCormick` Arrhenius function.
`xlog(a)`	21	$x \log(x)$ variant for McCormick instances.
`N O O P`	...	Error: Invalid tape extension or tape damaged.

Table 3. amodMC tape entry operation codes (opcodes): Display name used by printout routine, internal identifier and type description

the dependence of the rate constant of a chemical reaction on the activation energy and absolute temperature. The amodMC variant of this function calculates this rate constant without a pre-exponential factor taking both the activation energy and absolute temperature as its arguments.

Tape Printout amodMC exports a `mccormick_tape_print()` subroutine that allows users to print tape contents either to the console or a file. The display names shown in the first column of Table 3 are used to output readable opcode names instead of a numeric opcode. Tape printouts contain all data stored on the requested tape segment including temporaries; additionally, the output format is

designed to be parseable. Control parameters allow printouts of single tape entries, segments or the complete tape: This allows to obtain live printouts during debug sessions, e.g. in `dbg`, and enables automated external test systems to verify data integrity. Listing 1.9 shows the complete internal tape of the example function of Listing 1.5.

```
%TAPE
\buffer-status[12:64]
\content[
id,     op,p1,p2,      cv,     cc, adcvcv, adcccv, adcvcc, adcccc;
11,   dep ,10, 0, 3.531e0,7.987e0,1.000e0,0.000e0,0.000e0,1.000e0;
10, =(a,a), 9, 0, 3.531e0,7.987e0,0.000e0,0.000e0,0.000e0,0.000e0;
 9,sqrt(a), 8, 0, 3.531e0,7.987e0,0.000e0,0.000e0,0.000e0,0.000e0;
 8, +(a,a), 3, 7, 2.334e1,6.380e1,0.000e0,0.000e0,0.000e0,0.000e0;
 7, =(a,a), 6, 0, 3.085e0,3.730e1,0.000e0,0.000e0,0.000e0,0.000e0;
 6, *(a,a), 4, 5, 3.085e0,3.730e1,0.000e0,0.000e0,0.000e0,0.000e0;
 5,.inv.a , 1, 0, 0.222e0,0.321e0,0.000e0,0.000e0,0.000e0,0.000e0;
 4, exp(A), 2, 0, 2.009e1,7.557e1,0.000e0,0.000e0,0.000e0,0.000e0;
 3,pow(ai), 1, 0, 2.025e1,2.650e1,0.000e0,0.000e0,0.000e0,0.000e0;
 2, indep , 0, 0, 3.000e0,3.000e0,0.000e0,0.000e0,0.000e0,0.000e0;
 1, indep , 0, 0, 4.500e0,4.500e0,0.000e0,0.000e0,0.000e0,0.000e0;
]
```

Listing 1.9. Tape contents in descending order prior interpretation call of example function in Listing 1.5; to improve readability, temporaries and spaces have been removed, numbers vastly simplified and headers abbreviated. An unmodified tape printout would include four additional columns of temporaries and contain extended precision numbers

After tape interpretation completes, adjoints are propagated through the reversed tape. Listing 1.10 contains the same tape as Listing 1.9 but tape interpretation has been completed; starting from index 11 with `adcvcv` and `acccc` initially set to `1.0d0`, during revere sweep these values are used in adjoint computation and are finally stored in tape entries of independent variable registrations (indices 1 and 2 in the given example).

```
%TAPE
\buffer-status[12:64]
\content[
id,     op,p1,p2,      cv,     cc, adcvcv, adcccv, adcvcc, adcccc;
11,   dep ,10, 0, 3.531e0,7.987e0,1.000e0,0.000e0,0.000e0,1.000e0;
10, =(a,a), 9, 0, 3.531e0,7.987e0,1.000e0,0.000e0,0.000e0,1.000e0;
 9,sqrt(a), 8, 0, 3.531e0,7.987e0,1.000e0,0.000e0,0.000e0,1.000e0;
 8, +(a,a), 3, 7, 2.334e1,6.380e1,0.076e0,0.000e0,0.000e0,0.008e0;
 7, =(a,a), 6, 0, 3.085e0,3.730e1,0.076e0,0.000e0,0.000e0,0.008e0;
 6, *(a,a), 4, 5, 3.085e0,3.730e1,0.076e0,0.000e0,0.000e0,0.008e0;
 5,.inv.a , 1, 0, 0.222e0,0.321e0,0.206e0,0.000e0,0.000e0,1.163e0;
 4, exp(A), 2, 0, 2.009e1,7.557e1,0.011e0,0.000e0,0.000e0,0.001e0;
 3,pow(ai), 1, 0, 2.025e1,2.650e1,0.076e0,0.000e0,0.000e0,0.008e0;
 2, indep , 0, 0, 3.000e0,3.000e0,0.217e0,0.000e0,0.000e0,0.041e0;
 1, indep , 0, 0, 4.500e0,4.500e0,0.682e0,-0.0831,-0.0102,0.071e0;
]
```

Listing 1.10. Tape contents in descending order after tape interpretation

Summarizing `adcvcv` and `adcvcc` yields the subgradient for the convex underestimator w.r.t. the independent variable, and the sum of `adcccv` and `adcccc` the

concave overestimator w.r.t. the independent variable. This equals a call to `mccormick_indep_get`.

3.5 Tape Extensions

The amodMC library was designed to easily allow the integration of additional `McCormick`-enabled operators and intrinsics. For each operator family that shares the same reverse sweep implementation, a new opcode should be added to the internal `mccormick_tape_opcodes` array. Listing 1.11 contains the amodMC implementation of the exp function for `McCormick` arguments. `tape_alloc` is used to reserve two temporaries on the current tape entry. After computation of the convex underestimator and concave overestimator, the determined index and scaling factor are stored to these temporaries by directly accessing the tape. Temporaries are allocated per estimator, therefore four temporaries in total are available. At the end of the overloaded intrinsic, `tape_push` is called. This internal subroutine completes the current tape entry, using the opcode provided as its first argument (6), the variable returned by the overloaded operator or intrinsic (that is internally connected to the new tape entry) and up to two arguments of the overload operator or intrinsic used to trace the control flow of `McCormick` variables.

```
1  function exp_mc (a) result (r)
2      type(mccormick_type), intent(in) :: a
3      type(mccormick_type)             :: r
4      integer index
5      double precision scale
6
7      ! Compute boundaries.
8      r%l = exp(a%l)
9      r%u = exp(a%u)
10
11     call tape_alloc (tmp = 2)
12
13     ! Compute the convex underestimator.
14     index = -1
15
16     r%cv = exp(mid (a%cv, a%cc, a%l, index))
17
18     tape(tape_ptr)%xcv(1) = index
19     tape(tape_ptr)%xcv(2) = r%cv
20
21     ! Compute the concave overestimator.
22     index = -1
23     scale =  0.0d0
24
25     if (.not.equal_dp (a%l, a%u)) &
26     then
27         scale = (r%u - r%l) / (a%u - a%l)
28     endif
29
30     r%cc = exp(a%l) + scale * (mid (a%cv, a%cc, a%u, index) - a%l)
31
32     tape(tape_ptr)%xcc(1) = index
33     tape(tape_ptr)%xcc(2) = scale
34
35     call tape_push (6, r, a)
```

```
end function
```

Listing 1.11. amodMC implementation of exp function with `McCormick` type argument

Note, that composed functions implemented via calls to other composed or overloaded functions and operators require no special handling. The overloaded exponentiation intrinsic shown in Listing 1.12 will be represented by three tape entries: natural logarithm, multiplication and the exponential function.
Hence it is not possible during tape interpretation to determine whether a composed function has been called, operators and intrinsics that allow optimized (in terms of efficiency and numerical stability) reverse sweep implementations should be implemented directly, probably using redundant code, with an own opcode and reverse sweep code segment. Otherwise compositions are recommended due to improved readability, algorithmic stability and maintenance.

```
function pow_mc_mc (a, b) result (r)
    type(mccormick_type), intent(in) :: a, b
    type(mccormick_type)             :: r

    r = exp (b * log (a))
end function
```

Listing 1.12. Example implementation of a composed function

Tape interpretation iterates over the stored tape in descending order and executes specific code for each opcode. The current implementation uses a `select case` statement and combines opcodes for different operator groups and intrinsics with an equal reverse mode semantic. Listing 1.13 contains an excerpt from the tape interpretation subroutine, leaving out all but one adjoint computation for the overloaded `McCormick` assignment operator, addition of `McCormick` variables and scalars and subtraction of a scalar from a `McCormick` variable. New opcodes are easily appended to the current interpretation chain or combined with existing reverse sweeps. `tape_ptr` is a global variable storing the index of the next open tape entry.

```
subroutine mccormick_tape_interpret()
  integer i

  do i = tape_ptr - 1, 1, -1
    select case (tape(i)%op)
      case (1, 18, 19)
        ! Opcode  1 = =(a,a),
        !         18 = +(a,r) and +(r,a),
        !         19 = -(a,r)
        ! Computate subgradient convex.
        tape(tape(i)%p1)%adcvcv = &
        &   tape(tape(i)%p1)%adcvcv + tape(i)%adcvcv
        tape(tape(i)%p1)%adcvcc = &
        &   tape(tape(i)%p1)%adcvcc + tape(i)%adcvcc
        ! Computate subgradient concave.
        tape(tape(i)%p1)%adcccv = &
        &   tape(tape(i)%p1)%adcccv + tape(i)%adcccv
        tape(tape(i)%p1)%adcccc = &
        &   tape(tape(i)%p1)%adcccc + tape(i)%adcccc

      case (...)
```

```
        ...

    end select
  enddo
end subroutine
```

Listing 1.13. Excerpt from the main tape interpretation subroutine with one reverse code segment shared by three opcodes

4 Results

The runtimes for $f(\mathbf{x}) = \exp\left(\log\left(x_1 + \sum_{i=1}^{n-1} \frac{x_{i+1}}{x_i}\right)\right)$ can be found in Figure 1.

n	**Tangent-Linear**	**Adjoint**
50	0.004	0.004
100	0.006	0.004
200	0.011	0.005
250	0.018	0.005
500	0.043	0.008
1000	0.157	0.010
2000	0.607	0.017

Fig. 1. Runtimes in seconds for f

The runtimes for the branch-and-bound minimization of $g(\mathbf{x}) = -\exp\left(-\sum_{i=1}^{n}(x_j - 1)^2\right)$ for different n are given in Figure 2.

n	**Tangent-Linear**	**Adjoint**
10	0.4	0.45
20	2.4	1.75
30	6.2	3.9
40	13.0	7.1
50	23.0	10.9
60	42.2	20.36
90	129.34	51.32

Fig. 2. Runtimes for branch-and-bound for g

5 Conclusion and Outlook

This paper presented amodMC, a C++-library for propagating subgradients which enhances the former library modMC by an adjoint method. This method shares the same advantages as the reverse mode of usual AD for derivative calculations. Users now can choose between forward and reverse propagation of the needed subgradients based on the dimensionality of their problem. Usually the reverse mode is preferable if the input dimension is twice as high as the output dimension of the subdifferentiated function.

So up to now the two main procedures (forward an reverse) of AD have been successfully applied to subgradient propagation. The libraries modMC and

amodMC cover the same range of intrinsic functions as the original libMC presented in [MCB09]. This range is quite limited, i.e. it does not cover trigonometric or hyperbolic functions. The problem for these functions is the one of finding suitable convex and concave relaxations. Future work will especially be focused on this area making our library usable for a larger range of applications. Furthermore nearly all research results of AD, covering i.e. sparsity exploitation, elimination techniques or special heuristics, may be applied in this area.

References

[CMN+11] C. Corbett, M. Maier, U. Naumann, A. Ghobeity, M. Mitsos, and Markus Beckers. Compiler-generated subgradient code for McCormick relaxations. *ACM Transactions on Mathematical Software*, 2011. submitted.

[HUL93] J. Hiriart-Urruty and C. Lemaréchal. *Convex Analysis and Minimization Algorithms I.* Springer-Verlag, 1993.

[MCB09] A. Mitsos, B. Chachuat, and P. I. Barton. McCormick-based relaxations of algorithms. *SIAM Journal on Optimization*, 20(2):573–601, 2009.

[McC76] G. P. McCormick. Computability of global solutions to factorable nonconvex programs: Part I. Convex underestimating problems. *Mathematical Programming*, 10(2):147–175, 1976.

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.