



(Almost) matrix-free solver for piecewise linear functions in abs-normal form

Torsten Bosse

Department of Advanced Computing,
Friedrich-Schiller-Universität Jena, Jena,
Germany

Correspondence

Torsten Bosse, Department of Advanced Computing, Friedrich-Schiller-Universität Jena, Erns-Abbe-Platz 2, 07743 Jena, Germany.

Email: torsten.bosse@uni-jena.de

Summary

The abs-normal form (ANF) is a compact algebraic representation for piecewise linear functions. These functions can be used to approximate piecewise smooth functions and contain valuable information about the nonsmoothness of the investigated function. The information helps to define step directions within general Newton methods that obey the structure of the original function and typically yield better convergence. However, the computation of the generalized Newton directions requires the solution of a piecewise linear equation in ANF. It was observed that the ANF can become very large, even for simple functions. Hence, if a solver is based on the ANF and uses the (Schur-complement) matrices of the explicit ANF representation, it has to be considered computationally expensive. In this paper, we will address this question and present the first (almost) matrix-free versions of some solver for ANFs. The theoretical discussion is supported by some numerical run-time experiments.

KEYWORDS

abs-normal form, algorithmic differentiation, matrix-free, piecewise linear algebra

1 | INTRODUCTION

Streubel et al.¹ considered functions $\Delta y = \Delta F(x, \Delta x)$, $\Delta F(\cdot, \cdot) : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}^m$, that are piecewise linear with respect to the second argument $\Delta x \in \mathbb{R}^n$. It was also shown² that these functions can be used to approximate any abs-factorable function $y = F(x)$, $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$, at a fixed base point $x \in \mathbb{R}^n$ for variable directional increment $\Delta x \in \mathbb{R}^n$ and satisfy locally

$$\|F(x + \Delta x) - F(x) - \Delta F(x; \Delta x)\| \leq \gamma \|\Delta x\|^2, \quad 0 < \gamma \in \mathbb{R}. \quad (1)$$

Moreover, it was observed³ that the piecewise linear approximations ΔF contain valuable information about the nonsmoothness of the underlying function F and allow for a compact algebraic representation, which will be called abs-normal form (ANF) in the following. Namely, the piecewise linear approximations ΔF can be represented by the set of piecewise affine equations

$$\begin{bmatrix} \Delta z \\ \Delta y \end{bmatrix} = \begin{bmatrix} a \\ b \end{bmatrix} + \begin{bmatrix} Z & L \\ J & Y \end{bmatrix} \begin{bmatrix} \Delta x \\ |\Delta z| \end{bmatrix} \quad (2)$$

using an additional vector of switching variables $\Delta z \in \mathbb{R}^s$ of suitable dimension $s \in \mathbb{N}$ and its component-wise absolute value $|\Delta z| \in \mathbb{R}^s$. Here, the vectors and matrices $a = a(x) \in \mathbb{R}^s$, $b = b(x) \in \mathbb{R}^m$, $Z = Z(x) \in \mathbb{R}^{s \times n}$, $L = L(x) \in \mathbb{R}^{s \times s}$, $J = J(x) \in \mathbb{R}^{m \times n}$, $Y = Y(x) \in \mathbb{R}^{m \times s}$ represent certain derivative information that might depend on the base point x and can be computed by techniques from algorithmic differentiation (AD).^{4–7} In particular, it was shown² that the matrix $L \in \mathbb{R}^{s \times s}$

is strictly lower (left) triangular such that the result of the piecewise linear approximation Δy can be evaluated for any given increment Δx .

Based on the specific structure of the compact representation, several methods were developed to find a solution of an ANF if $n = m$ and given $\Delta y \in \mathbb{R}^n$, that is, compute $\Delta x \in \mathbb{R}^n$ such that $\Delta y = \Delta F(x, \Delta x)$ or, respectively, (2) holds. Two simple methods are the simple modulus-like iteration

$$\Delta z^{j+1} = [a - ZJ^{-1}(b - \Delta y)] + S|\Delta z^j|, \quad \text{for } j = 0, 1, \dots \quad (3)$$

and the signed fixed-point iteration

$$\Delta z^{j+1} = [I - S\Sigma_{\Delta z}]^{-1}[a - ZJ^{-1}(b - \Delta y)], \quad \text{for } j = 0, 1, \dots \quad (4)$$

to find a solution of the fixed-point equation $\Delta z = [a - ZJ^{-1}(b - \Delta y)] + S|\Delta z|$ —a rigorous derivation of the stated fixed-point equation/iterations, their connection to closely related linear complementarity problems (LCPs)⁸, and the absolute value equation⁹ besides other methods^{10–14} can be found in the cited references. Here,

$$S = L - ZJ^{-1}Y \in \mathbb{R}^{s \times s} \quad (5)$$

represents the Schur complement of the ANF's system matrix, and $\Sigma_{\Delta z} \in \mathbb{R}^{s \times s}$ is the diagonal matrix containing the signatures of the current switching variable Δz on its main diagonal such that $\Sigma_{\Delta z}\Delta z = |\Delta z|$. The fixed points Δz^* of these iterations* then yield the root $\Delta x^* = -J^{-1}[b - \Delta y + Y|\Delta z^*|]$, which satisfies $\Delta y = \Delta F(x, \Delta x^*)$. Obviously, these approximations and methods are of great interest because they allow the computation of generalized Newton directions Δx with $0 = F(x) + \Delta F(x, \Delta x)$ to find the root $x^* \in \mathbb{R}^n$ with $F(x^*) = 0$ for a large class of piecewise smooth functions $F : \mathbb{R}^n \rightarrow \mathbb{R}^n$.

Although preliminary results indicate that the generalized Newton directions using previously described approximations and methods yield qualitative better search direction,^{15–17} they are not yet common standard. The main reason is that the dimension $s \in \mathbb{N}$ of the switching variable vector Δz grows linear with the number of absolute value, minimum, or maximum function calls in the evaluation procedure of the original function F . Hence, any computation involving the Schur-complement matrix $S \in \mathbb{R}^{s \times s}$ has to be considered as numerically very expensive because almost any realistic applications might contain several thousands of absolute value calls.¹⁸

In this paper, we will address this problem and provide an (almost) matrix-free method for the numerically efficient solution of ANFs under the assumption that the tracing/evaluation of F is at reasonable costs and $m = n \ll s$. In detail, we will use standard techniques from AD² to generate four tapes from the function $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$, each of which coincides to one of four functions F_1, F_2, F_3 , and F_4 , respectively. The tapes are generated in such a way that the Jacobians of the resulting functions coincide with the ANF entries Z, L, Y , and J of the piecewise linearization of F . Thus, the forward-mode and the tapes then allow a matrix-free evaluation of the matrix–vector products

$$Zu, Lv, Ju, \text{ and } Yv$$

for general vectors $u \in \mathbb{R}^n$ and $v \in \mathbb{R}^s$ that can be used to reformulate (3) and (4) in a numerically more efficient way. Therefore, the functions F_1, F_2, F_3, F_4 and their relation to the ANF will be examined in more detail in Section 2. A practical implementation of how the tapes for the functions are generated using the tool ADOL-C¹⁹ will be shown in Section 3. In Section 4, we will discuss the (almost) matrix-free versions of the fixed-point iteration (3) and (4) and provide the necessary code samples for an implementation. The resulting codes will be applied on some examples for numerical run-time experiments in Section 5 before the conclusion in Section 6.

2 | COMPACT EVALUATION PROCEDURE REVISIT

In the sequel, we will only consider abs-factorable functions,² namely, functions $F : X \subseteq \mathbb{R}^n \rightarrow \mathbb{R}^m$ that are piecewise smooth and can be represented by the general evaluation procedure listed in Table 1. In detail, we assume that the function

*If existing and found; see the work of Griewank et al.⁸ for a more rigorous discussion on the convergence of both iterations

TABLE 1 (Left) general evaluation procedure for a piecewise smooth function $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$, $y = F(x)$, and (right) its piecewise linear extension $\Delta F : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}^m$, $\Delta y = \Delta F(x, \Delta x)$, using elemental functions φ_i from a library Φ for the computation of the intermediate variables $v_i \in \mathbb{R}$

| | | | |
|--------------------------------|-------------------------|---|-------------------------|
| $v_{i-n} = x_i$ | $i = 1 \dots n$ | $[v_{i-n}, \Delta v_{i-n}] = [x_i, \Delta x_i]$ | $i = 1 \dots n$ |
| $v_i = \varphi_i(v_j)_{j < i}$ | $i = 1 \dots l$ | $[v_i, \Delta v_i] = [\varphi_i(v_j)_{j < i}, \Delta \varphi_i(v_j, \Delta v_j)_{j < i}]$ | $i = 1 \dots l$ |
| $y_{i-l} = v_{i-m}$ | $i = l + 1 \dots l + m$ | $[y_{i-l}, \Delta y_{i-l}] = [v_{i-m}, \Delta v_{i-m}]$ | $i = l + 1 \dots l + m$ |

TABLE 2 (Left) reduced evaluation procedure for a piecewise smooth function $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$, $y = F(x)$, and (right) its reduced extension $\Delta F : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}^m$, $\Delta y = \Delta F(x, \Delta x)$, with preaccumulation of the smooth parts

| | | | |
|---------------------------------|-------------------------|---|-------------------------|
| $v_{i-n} = x_i$ | $i = 1 \dots n$ | $[v_{i-n}, \Delta v_{i-n}] = [x_i, \Delta x_i]$ | $i = 1 \dots n$ |
| $z_i = \psi_i(v_j)_{j < i}$ | $i = 1 \dots s$ | $[z_i, \Delta z_i] = [\psi_i(v_j)_{j < i}, \Delta \psi_i(v_j, \Delta v_j)_{j < i}]$ | $i = 1 \dots s$ |
| $v_i = \text{abs}(z_i)$ | | $[v_i, \Delta v_i] = [\text{abs}(z_i), \text{abs}(\Delta z_i) - \text{abs}(z_i)]$ | |
| $y_{i-s} = \psi_i(v_j)_{j < i}$ | $i = s + 1 \dots s + m$ | $[y_{i-s}, \Delta y_{i-s}] = [\psi_i(v_j)_{j < i}, \Delta \psi_i(v_j, \Delta v_j)_{j < i}]$ | $i = s + 1 \dots s + m$ |

$y = F(x)$ is given by a straight line code, which consists of finitely many intermediate values[†] $v_i = v_i(x) \in \mathbb{R}$ and only smooth elemental functions $\varphi_i \in \Phi$ (over the considered domain X)[‡] besides the absolute value function[§], that is, $\Phi = \{\pm, *, \text{rec}, \sin, \cos, \exp, \log, \dots, \text{abs}, \dots\}$.

As was shown by Griewank,² the piecewise linear approximation ΔF of an abs-factorable function F is obtained by augmenting each line in the original evaluation procedure according to the rules for piecewise linear differentiation

$$v_i = \varphi_i(v_j)_{j < i} \Rightarrow \Delta v_i = \Delta \varphi_i(v_j, \Delta v_j)_{j < i} = \begin{cases} \text{abs}(v_j + \Delta v_j) - \text{abs}(v_j), & \text{if } \varphi_i(v_j)_{j < i} \equiv \text{abs}(v_j), \\ \sum_{j < i} \left(\frac{\partial \varphi_i}{\partial v_j}(v_j) \right) \Delta v_j, & \text{else.} \end{cases} \quad (6)$$

The resulting *extended* evaluation procedure is also presented in Table 1 and looks similar to the standard tangent procedure for smooth functions.⁵ In fact, the proposed extended evaluation procedure is a generalization of the standard tangent evaluation procedure and reduces to it if F is smooth. Thus, if the evaluation routine of F does not contain any abs, min, max, or any other nonsmooth function calls, then the extended evaluation procedure simply yields the usual Jacobian-vector product $F'(x)\Delta x$ in the variable vector Δy . On the other hand, if the evaluation routine of F contains some absolute value function calls, then it can be represented by the *reduced* evaluation procedure given in Table 2. This reduced representation contains only the arguments $v_j \in \mathbb{R}$ and results $v_i = |v_j|$ of all $s \in \mathbb{N}$ absolute value function calls within the evaluation procedure besides the (in)dependent variables x and y . It can be obtained by preaccumulating all occurring smooth elemental functions φ_i to some more complex functions ψ_i . Furthermore, one can derive a reduced representation for the corresponding extended evaluation procedure, which is also provided in Table 2. This reduced version of the extended evaluation procedure then induces the algebraic formulation (2), where $\Delta z_i = \Delta z_i(x, \Delta x)$ now denotes the argument $v_i + \Delta v_i$ and $z_i = z_i(x) \equiv v_i$ of all absolute values for all $i = 1, \dots, s$.

All evaluation procedures can also be understood in terms of computational graphs,⁵ which helps to interpret the entries of the ANF. Namely, each evaluation procedure induces a directed, acyclic graph $G = (V, E)$, where each intermediate variable v_i corresponds to exactly one vertex $v_i \in V$ and the edges in $E \subseteq V \times V$ represent the underlying direct data dependence according to the precedence relation, that is, there exists a directed edge $(v_j, v_i) \in E$ from the vertex v_j to v_i if and only if $j < i$. Analogously, all intermediate variables Δv_i and their precedence relation induce a computational graph for the extended evaluation procedure. The reduced (extended) evaluation procedure then corresponds to graphs that are obtained by vertex elimination. Namely, eliminate all vertices $v \in V$ (and accumulate the corresponding edges) that do not belong to any (in)dependent variable or intermediate variable, which is either the argument or the result of an absolute value function call. The visualization of such a reduced computational graph for a function $F : \mathbb{R}^m \rightarrow \mathbb{R}^n$ with four absolute value calls and its piecewise linearization ΔF is presented in Figure 1.

As can be deduced from this figure, the computational graphs can be subdivided in to four different parts, each of which corresponds to one of the four mappings F_1 , F_2 , F_3 , and F_4 . In detail, F_1 corresponds to the mapping $x \rightarrow z(x)$, which only involves the direct computations that do not contain any absolute value calls. Accordingly, F_2 contains the parts

[†]For brevity, we will state the dependence of the intermediate variables with respect to the base point x and increment Δx only if necessary and use the abbreviation v_i , z_i , Δz_i , ... instead of the mathematically correct notation $v_i(x)$, $z_i(x)$, $\Delta z_i(x, \Delta x)$,

[‡]In the following, we will tacitly assume $x \in X$ such that $F(x)$ and $\Delta F(x, \Delta x)$ are always well defined.

[§]This includes the minimum and maximum, which can be written in terms of the absolute value, for example, $\min(u, v) = \frac{1}{2}(u + v - |u - v|)$.

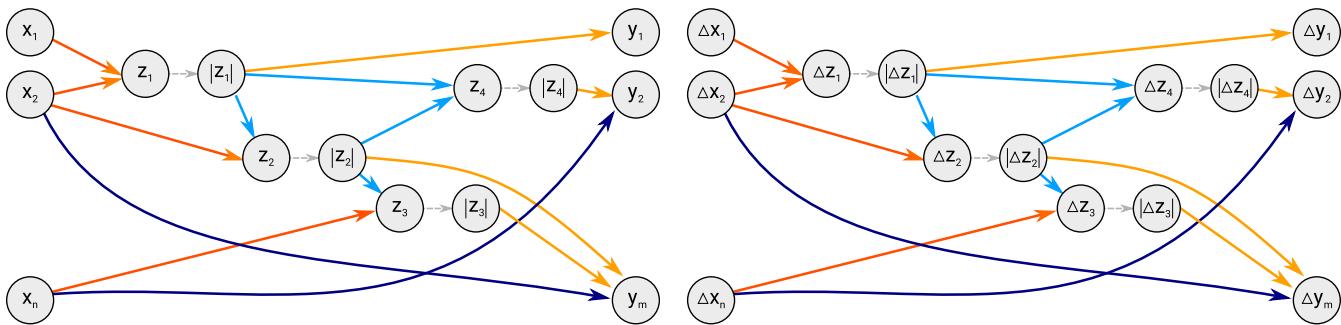


FIGURE 1 (Left) reduced computational graph of a general function $y = F(x)$ containing absolute value calls and (right) its piecewise linearization $\Delta y = \Delta F(x, \Delta x)$ for a fixed base point x with the induced subgraphs that correspond to the functions F_1 (dark orange), F_2 (light blue), F_3 (light orange), and F_4 (dark blue). The gray dashed arrows represent dependencies that are part of the computational graph but are not considered by the induced functions F_1, F_2, F_3 , and F_4

of F that directly map the result of all absolute values functions $|z_j(x)|$ to the argument $z_i(x)$ of the other absolute value calls. Hence, the function F_3 coincides with the direct mapping $|z(x)| \rightarrow y(x, |z(x)|)$ and F_4 belongs to the direct mapping $x \rightarrow y(x, |z(x)|)$. It turns out that these parts correspond exactly to the entries of the ANF, namely, the matrices of the ANF are given by the Jacobi matrices

$$Z \equiv \frac{\partial z(x)}{\partial x}(x), \quad L \equiv \frac{\partial z(x)}{\partial |z(x)|}(x), \quad Y \equiv \frac{\partial y(x, |z(x)|)}{\partial |z(x)|}(x), \quad \text{and} \quad J \equiv \frac{\partial y(x, |z(x)|)}{\partial x}(x),$$

where the partial derivatives now only consider direct dependencies, that is, only the parts of F_i that do not belong to paths in the computational graph, which are interrupted by some absolute value call. In addition, the two vectors $a = a(x)$ and $b = b(x, z(x))$ can be expressed using those functions:

$$a(x) = z(x) - L|z(x)| \quad \text{and} \quad b(x) = -Y(x)|z(x)|.$$

3 | TRACING THE ANF COMPONENTS SEPARATELY USING ADOL-C

In this section, we describe the necessary modifications for the software tool ADOL-C¹⁹ to compute the entries of the ANF separately, that is, the vectors a , b and matrices Z , L , Y , J . The modifications allow to generate four different tapes that represent the functions F_1 , F_2 , F_3 , and F_4 . The resulting tapes can be used in terms of the forward/backward mode to efficiently evaluate matrix–vector products with those matrices and provide the basis for the matrix-free solvers presented in the next section.

Therefore, we assume that the function $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$ is given by a straight line C/C++ code similar to Listing 1 that satisfies the previously stated assumptions, namely, the function can be formulated as an evaluation procedure that only uses smooth elemental functions besides minimum, maximum, and the absolute value function.

Furthermore, we assume that there exists a duplicate version of Listing 1 using the `adouble` variables from ADOL-C for all active intermediate variables and the (in)dependent variables $x \in \mathbb{R}^n$ and $y \in \mathbb{R}^m$, respectively. The only other

```
void eval(int m, int n, double* x, double* y){
    ... // Original code for y = F(x) using doubles with
    ... // std::fabs, std::fmin, and std::fmax
}
```

Listing 1 “Original implementation of the function $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$ using variables of type `double`”

```
#include "pladouble.hpp"
void eval(int m, int n, adouble* x, adouble* y){
    ... // Overloaded code for y = F(x) using adoubles with
    ... // pladolc::fabs, pladolc::fmin, and pladolc::fmax}
```

Listing 2 “Original implementation of the function $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$ using variables of type `adouble`”

```

#include "pladouble.hpp"
using namespace pladolc;
int main( int argc, char **argv ){
    int n=..., m=...;                                // Number of (in-)dependents variables
    double* x = new double[n];                        // Vector of independent variables
    double* y = new double[m];                        // Vector of dependent variables
    ...
    int my_tracemode = ...;                          // Set values for x and other computations
    int ANFtape = ...;                               // Specify the tracing mode to be {0,1,2,3,4}
    set_tracemode(my_tracemode);                     // Tape-number - has to be specified by the user
    trace_on(ANFtape);                             // Set tracemode: 1(=Z), 2(=L), 3(=Y) or 4(=J)
    trace_on(ANFtape);                             // Set tape for current mode and start tracing
    adouble *ax = new adouble[ n ];                  // adouble vector for independent variables
    adouble *ay = new adouble[ m ];                  // adouble vector for dependent variables
    set_independent(n, x, ax);                      // Set independent variables
    eval(m, n, ax, ay);                            // Call original function with adoubles
    set_dependent(m, ay, y);                        // Set dependent variables
    delete[] ax;                                    // Free vector of intermediate adoubles ax
    delete[] ay;                                    // Free vector of intermediate adoubles ay
    trace_off();                                   // Stop tracing
    ...
    delete[] x;                                     // Some other operations using
    delete[] y;                                     // Free double vector x
                                                // Free double vector y
}

```

Listing 3 “Tracing of the four tapes that can be used to compute the different parts of the ANF”

```

namespace pladolc{
    int tracemode;
    void set_tracemode( int my_tracemode ){
        if (tracemode==0){ enableMinMaxUsingAbs(); }
        tracemode = my_tracemode;
    }
}

```

Listing 4 “Method to set the global trace-mode variable”

modification, besides using `adouble` instead of `double` variables, is to use a specialized version of the maximum, minimum, and absolute value, as shown in Listing 2.

The taping of the four functions is almost equivalent to the standard procedure described in the manual of ADOL-C¹⁹ for the general tracing of abs-factorable functions. An actual implementation of the tracing process for the four functions is presented in Listing 3. It follows the standard procedures of ADOL-C: the initialization of the base point x , the definition of a tape number for the `trace_on` command, the declaration of the independent variables followed by the evaluation of the function F using the `adouble` version, and the declaration of the dependent variables before the tracing is stopped by the `trace_off` call.

The main difference to the usual tracing is the function `set_tracemode` presented in Listing 4, which sets a global integer variable `tracemode` to the user-specified value `my_tracemode`. In case that the `tracemode` is set to the value “0,” the tracing procedure in Listing 3 generates the standard tape of the function F using the piecewise linear mode as described in the work of Walther et al.²⁰ The result is stored in the tape with the user-specified tape-number `ANFtape`.

For the values “1,” “2,” “3,” or “4,” Listing 3 will generate tapes that correspond to one of the previously introduced functions F_1 , F_2 , F_3 , and F_4 , respectively. The key for the automatic generation of those four tapes is hidden in the methods `set_independent` and `set_dependent`, as well as the special versions of `fabs`, `fmin`, and `fmax` from the namespace `pladolc`. All these methods depend on the `tracemode` variable as can be deduced from their actual implementation given in Listings 5, 6, and 7.

In detail, `tracemode=1` provides a tape for the function $F_1 : \mathbb{R}^n \rightarrow \mathbb{R}^s$ with the user-specified tape number `ANFtape`. Therefore, all input variables $x \in \mathbb{R}^n$ will be marked as independent variables in `set_independent` using the “`<<=`”

```

namespace pladolc {
    void set_independent(int n, double* x, adouble* ax) {
        if ((tracemode==0)|| (tracemode==1)|| (tracemode==4)){
            for (int i=0; i < n; i++){ ax[i]<=x[i]; }
        } else {
            for (int i=0; i < n; i++){ ax[i] = x[i]; }
        }
    }

    void set_dependent(int m, adouble* ay, double* y) {
        if ((tracemode==0)|| (tracemode==3)|| (tracemode==4)){
            for (int i=0; i < m; i++){ ay[i]>=y[i]; }
        }
    }
}

```

Listing 5 “Definition of (in)dependent variables that supports tracing of separate parts”

```

namespace pladolc{
    adouble fabs( adouble az ){
        double v;
        adouble av;
        if (tracemode == 1){           // Only Z part will be traced
            az >>= v;
            av = std::fabs(v);
        } else if (tracemode==2){      // Only L part will be traced
            az >>= v;
            v = std::fabs(v);
            av <=> v;
        } else if (tracemode==3){     // Only Y part will be traced
            v = std::fabs(az.getValue());
            av <=> v;
        } else if (tracemode==4){     // Only J part will be traced
            v = std::fabs(az.getValue());
            av = v;
        } else{                      // Standard tracing with all parts
            av = fabs(az);           // Standard ADOLC fabs for adoubles
        }
        return av;
    }
}

```

Listing 6 “Modified absolute value function that supports tracing of separate parts”

operator from ADOL-C. Accordingly, all arguments of the absolute value function `fabs`[¶] are declared as the dependent variables $z \in \mathbb{R}^s$ using the “ $\gg=$ ” operator. The subsequent assignment `av = std::fabs(v)`; with the temporary double variable guarantees that all following computations are well defined but will not be traced as active part of the function evaluation anymore. In other words, the declaration of the arguments of the absolute value function as dependent variables and the use of the temporary variable `v` ensures that only direct parts of the mapping $x \rightarrow z(x)$ are traced, that is, only computations between independent and dependent variables that do not contain any intermediate absolute value, minimum, or maximum function call.

A similar trick is used for the choice `tracemode=2`, which yields a tape for the function $F_2 : \mathbb{R}^s \rightarrow \mathbb{R}^s$. As explained in Section 2, the function F_2 corresponds to the parts of F that consist of computations between the outputs of all absolute value function calls and the inputs of all other absolute value functions that depend directly on them. Hence, any

[¶]and, thus, the minimum and maximum functions, which are defined in terms of this `fabs`

```

namespace pladolc {
    adouble fmin( adouble au, adouble av ){
        adouble aw = 0.5*(au+av-fabs(au-av)); // Pladolc fabs
        return aw;
    }
    adouble fmin( double au, adouble av ){
        adouble aw = 0.5*(au+av-fabs(au-av)); // Pladolc fabs
        return aw;
    }
    adouble fmin( adouble au, double av ){
        adouble aw = 0.5*(au+av-fabs(au-av)); // Pladolc fabs
        return aw;
    }
    adouble fmax( adouble au, adouble av ){
        adouble aw = 0.5*(au+av-fabs(au+av)); // Pladolc fabs
        return aw;
    }
    adouble fmax( double au, adouble av ){
        adouble aw = 0.5*(au+av-fabs(au+av)); // Pladolc fabs
        return aw;
    }
    adouble fmax( adouble au, double av ){
        adouble aw = 0.5*(au+av-fabs(au+av)); // Pladolc fabs
        return aw;
    }
}
}

```

Listing 7 “Maximum and minimum that use the modified abs and support the tracing of separate parts”

output of `fabs` is marked as independent, whereas any input is defined as dependent. Again, the assignment `v = std::fabs(v);` using the temporary `double` variable `v` ensures that only direct computations are taped and omit any parts of the computational graph that involve some intermediate absolute value call. In contrast to before, the values of $x \in \mathbb{R}^n$ are now only assigned as constant values to the `adouble` array `ax` but not defined as independent variables. This ensures that all computations are well defined and that the tape for F_2 yields the correct matrix $L = L(x)$ for the base point x , where the function was traced.

Analogously, the correct specification of (in)dependent variables and the use of the intermediate `double` variable `v` yield traces for the functions F_3 and F_4 if `tracemode` equals “3” or “4,” respectively.

The generated tapes can be used to either compute matrix–vector products with the matrices Z, L, Y, J or evaluate the parts of the ANF itself using the standard Jacobian driver provided by ADOL-C. An incomplete example for the simple evaluation of direct matrix–vector products is given in Listing 8. It illustrates how the standard forward mode driver `fos_forward` from ADOL-C can be applied to compute $Z\Delta x, L\Delta z, Y\Delta z$, and $J\Delta x$ if the tapes of the functions F_i are identified by the corresponding tape variables `Tape_Z`, `Tape_L`, `Tape_Y`, and `Tape_J`, respectively. In a similar way, one can also use the driver for the reverse mode of ADOL-C to compute adjoint vector products with the specific matrices, if required.

The entries of the ANF are obtained in a similar way as can be deduced from Listing 9, where the standard Jacobian method from ADOL-C is used to derive the vectors a, b and the matrices Z, L, Y , and J separately. A drawback of the presented approach is that the tracing needs to be done four times to generate all tapes. In addition, the tapes for the functions F_i are only valid at the point $x \in \mathbb{R}^n$, where the functions were traced. Thus, some retaping is required whenever the base point x changes. In contrast, the standard ADOL-C driver `abs_normal` requires only one trace and the resulting tape can be reused at different base points x .

However, in some situations, one or more of the ANF matrices do not depend on x but are constant. In this case, it might pay off to initially invest the additional overhead for tracing and use the specific tapes to only re-evaluate the parts

```

#include "pladouble.hpp"
using namespace pladolc;
int main( int argc , char **argv ){
    ...
    // Tracing of separate parts of F at x
    ...

    int s      = get_switches();           // Get dimension of z (and Dz)
    double* z  = alloc_vec<double>( s ); // Allocate memory for z
    double* Dx = alloc_vec<double>( n ); // Allocate memory for Dx
    double* Dz = alloc_vec<double>( s ); // Allocate memory for Dz
    double* F1 = alloc_vec<double>( s ); // Allocate memory for F1(x)
    double* F2 = alloc_vec<double>( s ); // Allocate memory for F2(z)
    double* F3 = alloc_vec<double>( m ); // Allocate memory for F3(z)
    double* F4 = alloc_vec<double>( m ); // Allocate memory for F4(x)
    double* ZDx = alloc_vec<double>( s ); // Allocate memory for Z*Dx
    double* LDz = alloc_vec<double>( s ); // Allocate memory for L*Dz
    double* YDz = alloc_vec<double>( m ); // Allocate memory for Y*Dz
    double* JDx = alloc_vec<double>( m ); // Allocate memory for J*Dx

    ...
    // Compute z, Dx and Dz
    ...

    int keep = 1;
    fos_forward(Tape_Z, m, s, keep, x, Dx, F1, ZDx); // Evaluate F_1(x) and Z*Dx
    fos_forward(Tape_L, s, s, keep, z, Dz, F2, LDz); // Evaluate F_2(z) and L*Dz
    fos_forward(Tape_Y, m, s, keep, z, Dz, F3, YDz); // Evaluate F_3(z) and Y*Dz
    fos_forward(Tape_J, m, n, keep, x, Dx, F4, JDx); // Evaluate F_4(x) and J*Dx

    ...
    // Other operations using F1, ZDx, ...
    ...
}

```

Listing 8 “Computation of matrix vector products using the four different tapes and standard ADOL-C driver”

of the ANF that have changed rather than a complete re-evaluation of ANF. Moreover, the cost for tracing functions is usually given by a small multiple of the cost for evaluating the function itself, namely,

$$\text{cost(trace function)} \leq c_{\text{trace}} \text{cost(evaluate function)},$$

where c_{trace} is typically a small constant around 20 not depending on the dimensions m, n , or s . Hence, the initial overhead might pay off if only a comparatively small number of tape evaluations in terms forward or reverse mode calls are needed because the cost for one forward or reverse evaluation is roughly equal to the cost for evaluating the function, whereas the cost for evaluating the complete ANF is proportional to approximately evaluating the function $\min(n + s, m + s)$ times itself.

4 | MATRIX-FREE FIXED-POINT SOLVER

In this section, we will present different strategies to implement the simple modulus-like iteration (3) and signed fixed-point iteration (4) stated in the introduction. Although the presented reformulations are theoretically equivalent,

```

#include "pladouble.hpp"
using namespace pladolc;
int main( int argc , char **argv ){
    ...
    // Tracing separate parts of F at x
    ...
    int s      = get_switches();           // Get dimension of z (and Dz)
    double* a = alloc_vec<double>( s ); // Allocate memory for the components
    double* b = alloc_vec<double>( m ); // a,b,Z,L,Y,J of the ANF at x
    double** Z = alloc_mat<double>( s , n );
    double** L = alloc_mat<double>( s , s );
    double** Y = alloc_mat<double>( m , s );
    double** J = alloc_mat<double>( m , n );
    double* x = alloc_vec<double>( n ); // Base point x
    double* y = alloc_vec<double>( m ); // Function value y=F(x)
    double* z = alloc_vec<double>( s ); // Switching vector z=z(x)
    double* z_abs = alloc_vec<double>( s ); // Absolute value of switching vector z
    double* tmp1 = alloc_vec<double>( s ); // Temporary vectors required by the
    double* tmp2 = alloc_vec<double>( s ); // forward mode to compute the vectors

    int keep = 1;
    zos_forward(Tape_Z, s, n, keep, x, z);
    abs( s, z, 1, z_abs, 1 );
    fos_forward(Tape_L, s, s, keep, z, z_abs, tmp1, tmp2);
    for( int i=0;i<s;i++){
        a[ i ] = z[ i ]-tmp2[ i ];
    }
    fos_forward(Tape_Y, m, s, keep, z_abs, z_abs, y, b);
    for( int i=0;i<m;i++){
        b[ i ]=-b[ i ];
    }
    jacobian(Tape_Z, s, n, x, Z );           // Eval. Jacobian of F_1 stored on Tape_Z
    jacobian(Tape_L, s, s, z, L );           // Eval. Jacobian of F_2 stored on Tape_L
    jacobian(Tape_Y, m, s, z, Y );           // Eval. Jacobian of F_3 stored on Tape_Y
    jacobian(Tape_J, m, n, x, J );           // Eval. Jacobian of F_4 stored on Tape_J
    ...
    // Some other operations using a,b,Z, ...
    ...
}

```

Listing 9 “Evaluation of the ANF components using the four different tapes and standard ADOL-C driver”

we will see that the numerical effort for their practical implementation can be significantly different. An overview about the essential parts for each the presented approaches will be provided in Table 6.

4.1 | Matrix-free version for the simple modulus-iteration

Consider the simple modulus-like iteration $\Delta z^{j+1} = c + S|\Delta z^j|$, for $j = 0, 1, \dots$, which can be implemented in various ways. In any case, the vector $c = a - ZJ^{-1}[b - \Delta y]$ needs to be evaluated and stored only once at the beginning of the fixed-point iteration. However, the matrix–vector product $S|\Delta z|$ with the Schur-complement matrix $S = L - ZJ^{-1}Y$ can be obtained by at least six different approaches.

The first approach, Approach I, assumes that all matrices Z, L, J , and Y are precomputed and stored. Based on a factorization of J , it evaluates and stores the matrix S once at the beginning of the iteration and computes in every step only the

TABLE 3 Memory requirements and FLOPs for the simple modulus-iteration (3) using Approach I (with precomputation of S , factorization for J), Approach II (with precomputation of S , linear solver for J), Approach III (no precomputation of S , factorization for J), Approach IV (no precomputation of S , linear solver for J), Approach V (almost matrix-free, factorization for J), and Approach VI (matrix-free, linear solver for J)

| Modulus | Memory | FLOPs (once) | FLOPs (per iteration) |
|--------------|------------------------|------------------------------|-------------------------|
| Approach I | $O(s^2 + n^2 + 2ns)$ | $O(s^2 + n^3 + s^2n + n^2s)$ | $O(s^2)$ |
| Approach II | $O(s^2 + n^2 + 2ns)$ | $O(s^2 + ksn^2 + s^2n)$ | $O(s^2)$ |
| Approach III | $O(s^2/2 + n^2 + 2ns)$ | $O(n^3)$ | $O(s^2/2 + n^2 + 2ns)$ |
| Approach IV | $O(s^2/2 + n^2 + 2ns)$ | -- | $O(s^2/2 + kn^2 + 2ns)$ |
| Approach V | $O(n^2 + s)$ | $O(n^3)$ | $O(n^2)$ |
| Approach VI | $O(n + s)$ | -- | -- |

Note. FLOPs = floating point operations.

matrix product $S|\Delta z|$. Similar to the first approach, the second approach, Approach II, assumes that all matrices Z, L, J , and Y are stored but employs an iterative solver to evaluate $J^{-1}Y$ for computing the Schur-complement matrix S instead of using a factorization for J . The third and fourth alternatives omit the explicit computation of the matrix S but only use the stored matrices Z, L, J , and Y to evaluate in each iteration $L|\Delta z| - ZJ^{-1}Y|\Delta z|$ from right to left. As before, one can distinguish two cases: either a factorization of J is computed once at the beginning of the fixed-point iteration (Approach III) or some iterative solver is used in each iteration to compute the products $J^{-1}u$ (Approach IV). As further alternatives, we propose two (almost) matrix-free versions. The proposed alternatives are based on the third and fourth approach. In detail, instead of computing and storing the matrices Z, L, J , and Y , both (almost) matrix-free approaches use the generated tapes from the previous section and the forward-mode of AD to compute the necessary matrix–vector products to evaluate the expression $L|\Delta z| - ZJ^{-1}Y|\Delta z|$ from right to left. Again, one can distinguish between two approaches: the almost matrix-free approach, Approach V, which explicitly computes and stores a factorization of the matrix J , and the matrix-free approach, Approach VI, which employs an iterative solver for $J^{-1}u$ that only uses the corresponding tape F_4 to evaluate Jv for given $v \in \mathbb{R}^n$.

As mentioned at the beginning of this section, all approaches for this method are theoretically equivalent but differ by their numerical effort in terms of memory requirements and floating point operations (FLOPs). In detail, the required memory to store the matrices Z, L, J , and Y for Approach I–Approach IV is of order $O(s^2/2 + n^2 + 2ns)$. Additionally, the first two approaches, Approach I and Approach II, require $O(s^2)$ to store the matrix S . In contrast, the (almost) matrix-free approaches, Approach V and Approach VI, only need store a few intermediate vectors of order $O(s + n)$ besides $O(n^2)$ for the matrix J and its factorization.

The computational effort in terms of FLOPs can be subdivided into two parts: the initial overhead that only needs to be invested once at the very beginning of the fixed-point iteration and the effort required in every iteration of the fixed-point iteration. The initial overhead to precompute S in Approach I is approximately $O(s^2 + n^3 + n^2s + s^2n)$ FLOPs for the involved matrix subtraction, the matrix factorization of J , and the two matrix multiplications. Analogously, the effort for Approach II is approximately $O(s^2 + kn^2s + s^2n)$ FLOPs, where $k \in \mathbb{N}$ now denotes the (average) number of inner iterations for the iterative linear solver to compute $J^{-1}u$. In both cases, the effort per (outer) fixed-point iteration step is $O(s^2)$. Obviously, the initial overhead for computing S does not apply for Approach II–Approach VI. The only initial overhead, required by Approach III and Approach V, is of order $O(n^3)$ for the factorization of the matrix J , which allows a subsequent evaluation of $J^{-1}u$ within $O(n^2)$ per iteration.

An overview about the memory requirements and FLOPs for all presented simple modulus approaches is given in Table 3. Obviously, these costs only represent the FLOPs for matrix operations but not the additional effort for evaluating the matrices and Jacobi-vector evaluations using AD. These extra costs are listed separately in Table 4 and can be divided into three parts: the initial costs for taping the function(s), the initial costs to evaluate derivative matrices, and the costs per iteration. In detail, the initial effort needs to be invested only once at the beginning of the fixed-point iteration, whenever x changes. In particular, the costs for taping the function F is similar for the first four approaches and corresponds to the cost for evaluating the original function F itself times a certain small factor $p \in \mathbb{R}$, which is typically of magnitude $p \approx 10 - 20$.

Once the tape for F is available, one can use the standard AD driver provided by ADOL-C to compute the derivative matrices Z, L, J , and Y , at a cost that is approximately $O(s + n)\text{cost}(F)$, namely, $(s + n)$ times the effort to evaluate F itself.

TABLE 4 Algorithmic differentiation (AD) operations for the simple modulus-iteration (3) using Approach I (with precomputation of S , factorization for J), Approach II (with precomputation of S , linear solver for J), Approach III (no precomputation of S , factorization for J), Approach IV (no precomputation of S , linear solver for J), Approach V (almost matrix-free, factorization for J), and Approach VI (matrix-free, linear solver for J)

| Modulus | Taping | Forward-mode (once) | Forward-mode (per iteration) |
|---------------|----------------------|--------------------------|--|
| Approach I-IV | $O(1)\text{cost}(F)$ | $O(s + n)\text{cost}(F)$ | -- |
| Approach V | $O(4)\text{cost}(F)$ | $O(n)\text{cost}(F_4)$ | $O(1)\text{cost}(F_1, F_2, \text{ and } F_3)$ |
| Approach VI | $O(4)\text{cost}(F)$ | -- | $O(k)\text{cost}(F_4) + O(1)\text{cost}(F_1, F_2, \text{ and } F_3)$ |

This initial cost is partly compensated by the effort required in each iteration, where no further AD operations are needed by Approach I–Approach IV. In contrast, the initial cost to generate the tapes for Approach V and Approach VI is slightly higher because both methods require the tracing to be done four times to obtain the tapes for the functions F_1, F_2, F_3 , and F_4 . On the other hand, no or only a small initial effort of order $O(n)\text{cost}(F)$ is required to compute the derivative matrices by the matrix-free approach and the almost matrix-free approach to evaluate J , respectively.

In summary, there are at least six different approaches to implement the simple modulus iteration (3), which can differ significantly in their numerical effort and memory requirements, as can be seen from the Tables 3 and 4. The difference between the approaches mainly stems from the fact that initial costs are replaced by certain tasks that then need to be executed in every iteration. Obviously, there is no general rule to determine the most efficient approach because their efficiency strongly depends on several problem characteristics, namely, the underlying dimensions (m, s, n), the sparsity patterns of the involved matrices (Z, L, J, Y), the required number of fixed point and inner iterations (k), and the effort to evaluate the functions (F, F_1, F_2, F_3, F_4). An overview about the different strategies to compute the respective parts within the six different approaches is given in Table 5. A summary of the table can be found in 6, which also contains the summary for the simple signed iteration (4) that will be presented in the following.

4.2 | Matrix-free version for the signed iteration

In contrast to the simple modulus fixed-point iteration, the signed fixed-point iteration

$$\Delta z^{j+1} = [I - S\Sigma_{\Delta z^j}]^{-1}c \quad \text{for } j = 0, 1, \dots$$

requires in each iteration the solution of the linear equation with the matrix $\tilde{S} = I - S\Sigma_{\Delta z^j}$ that involves the matrix S and depends on the signatures of the current approximation Δz^j . As before, the signed iteration can be implemented in various ways: the linear problem $\Delta z^{j+1} = \tilde{S}^{-1}c$ can be solved either using a factorization of \tilde{S} or some iterative method that requires the evaluation of the matrix–vector products $\tilde{S}v$. Moreover, one can also either precompute the matrix S , only store the matrices (Z, L, J, Y) to evaluate Sv from right to left, or use the AD tapes for the functions (F_1, F_2, F_3, F_4). Furthermore, the linear subproblem $J^{-1}v$ can be solved using a (pre)factorization of J or some iterative method. This yields at least six different approaches for the signed fixed-point iteration that are presented in the following.

Similar to Approach I for the modulus iteration, Approach I for the signed fixed-point iteration assumes that the matrices Z, L, J , and Y are available and precomputes the matrix S using a factorization for J . The matrix S is then used to compute a factorization of \tilde{S} in each iteration to find the next approximation Δz^{j+1} . The second approach, Approach II, is equivalent to the first approach except that it uses an iterative solver for \tilde{S} with precomputed S . Based on the second approach, Approach III and Approach IV employ an iterative solver for \tilde{S} and omit the precomputation of the Schur-complement matrix S . Instead, both approaches use the stored matrices Z, L, J , and Y to evaluate the required matrix–vector products $\tilde{S}v = (I - (L - ZZ^{-1}Y)\Sigma_{\Delta z^j})v$ from right to left for given $v \in \mathbb{R}^s$. The only difference between Approach III and Approach IV is that the subproblems involving J^{-1} are either solved using a factorization of J or an iterative solver, respectively. Consequently, Approach V and Approach VI coincide with Approach III and Approach IV but now use the forward mode of AD and the tapes for F_1, F_2, F_3 , and F_4 , to evaluate the required matrix–vector expressions involving Z, L, Y , and J .

The effort in terms of memory requirements, FLOPs, and AD calls for all six approaches of the signed fixed-point iteration is listed in Tables 7 and 8, where $k \in \mathbb{N}$ and $l \in \mathbb{N}$ denote the number of iterations required by the linear solver to find a solution of $J^{-1}u$ and $\tilde{S}^{-1}v$, respectively. A summary for the expected effort for the computation of c and Δx is given in Table 9.

TABLE 5 Description for the three main parts (initialization, fixed-point iteration, finalization) of a solver that implements the simple modulus-like iteration (3). Here, each of the right columns (*I–VI*) represents one of the six discussed approaches. The symbol ✓ indicates which strategy is used in the approach to solve the corresponding operation described in the left column. The right column provides the costs for each task in terms of floating point operations (FLOPs) and algorithmic differentiation (AD) operations

| Simple modulus iteration - Approach | | <i>I</i> | <i>II</i> | <i>III</i> | <i>IV</i> | <i>V</i> | <i>VI</i> | FLOPs and AD operation |
|--|--|-----------------|------------------|-------------------|------------------|-----------------|------------------|--|
| Initialization | Compute Z, L , and Y | ✓ | ✓ | ✓ | ✓ | | | $O((s + n) * \text{costs}(F_1, F_2, F_3))$ |
| | Compute J | ✓ | ✓ | ✓ | ✓ | ✓ | | $O(n * \text{costs}(F_4))$ |
| | Compute factorization of J | ✓ | | ✓ | | ✓ | | $O(n^3)$ |
| | Compute $c = a - ZJ^{-1}[b - \Delta y]$ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | |
| | -using factorization of J for $J^{-1}v$ | ✓ | | ✓ | | ✓ | | $O(n^2) + \dots$ |
| | -using iterative solver for $J^{-1}v$ | | ✓ | | ✓ | | ✓ | |
| | -using the matrix J for Jv | | ✓ | | ✓ | | | $O(kn^2) + \dots$ |
| | -using the tape F_4 for Jv | | | | | ✓ | | $O(k * \text{costs}(F_4)) + \dots$ |
| | -using the matrix Z for Zv | ✓ | ✓ | ✓ | ✓ | | | $\dots + O(ns)$ |
| | -using the tape F_1 for Zv | | | | | ✓ | ✓ | $\dots + O(\text{costs}(F_1))$ |
| Fixed-point iteration | Compute $S = L - ZJ^{-1}Y$ | ✓ | ✓ | | | | | $O(s^2/2 + s^2n) + \dots$ |
| | -using factorization of J for $J^{-1}Y$ | ✓ | | | | | | $\dots + O(n^2)$ |
| | -using the matrix J iterative solver for $J^{-1}Y$ | | ✓ | | | | | $\dots + O(k * \text{costs}(F_4))$ |
| | repeat | | | | | | | |
| | Compute Δz^{j+1} | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | |
| | -evaluating $c + S \Delta z^j $ | ✓ | ✓ | | | | | $O(s^2)$ |
| | -evaluating $c + L \Delta z^j - Z[J^{-1}[Y \Delta z^j]]$ | | | ✓ | ✓ | ✓ | ✓ | |
| | -using Z, L , and Y for Zv, Lv , and Yv | | | ✓ | ✓ | | | $O(ns + s^2/2 + sn) + \dots$ |
| | -using F_1, F_2 , and F_3 for Zv, Lv , and Yv | | | | | ✓ | ✓ | $O(\text{costs}(F_1, F_2, F_3)) + \dots$ |
| | -using factorization of J for $J^{-1}v$ | | | ✓ | | ✓ | | $\dots + O(n^2)$ |
| Finalization | -using iterative solver for $J^{-1}v$ | | | | ✓ | | ✓ | |
| | -using the matrix J for Jv | | | | ✓ | | | $\dots + O(k * n^2)$ |
| | -using the tape F_4 for Jv | | | | | ✓ | | $\dots + O(k * \text{costs}(F_4))$ |
| | while $(\ \Delta z^{j+1} - \Delta z^j\ > tol)$ | | | | | | | |
| | Compute $\Delta x^* = -J^{-1}[b - \Delta y + Y \Delta z^*]$ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | |
| | -using the matrix Y for Yv | ✓ | ✓ | ✓ | ✓ | | | $O(sn) + \dots$ |
| | -using the tape F_3 for Yv | | | | | ✓ | | $O(\text{costs}(F_3)) + \dots$ |
| | -using factorization of J for $J^{-1}v$ | ✓ | | ✓ | | ✓ | | $\dots + O(n^2)$ |
| | -using iterative solver for $J^{-1}v$ | ✓ | | | ✓ | | ✓ | |
| | -using the matrix J for Jv | | ✓ | | ✓ | | | $\dots + O(kn^2)$ |
| | -using the tape F_4 for Jv | | | | | ✓ | | $\dots + O(k * \text{costs}(F_4))$ |

Note. FLOPs = floating point operations.

TABLE 6 Overview about the different strategies that were used to compute the parts within the six different approaches for (left) the simple modulus iteration (3) and (right) the simple signed iteration (4)

| Task - Approach | <i>I</i> | <i>II</i> | <i>III</i> | <i>IV</i> | <i>V</i> | <i>VI</i> | Task - Approach | <i>I</i> | <i>II</i> | <i>III</i> | <i>IV</i> | <i>V</i> | <i>VI</i> |
|-----------------------------------|-----------------|------------------|-------------------|------------------|-----------------|------------------|---|-----------------|------------------|-------------------|------------------|-----------------|------------------|
| Compute Z, L, Y | ✓ | ✓ | ✓ | ✓ | | | Compute Z, L, Y | ✓ | ✓ | ✓ | ✓ | | |
| Compute J | ✓ | ✓ | ✓ | ✓ | ✓ | | Compute J | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Compute $S = L - ZJ^{-1}Y$ | ✓ | ✓ | | | | | Compute $S = L - ZJ^{-1}Y$ | ✓ | ✓ | | | | |
| Evaluate Sv | ✓ | ✓ | | | | | Use factorization for \tilde{S}^{-1} | ✓ | | | | | |
| Evaluate $L - Z(J^{-1}(Yv))$ | | ✓ | ✓ | ✓ | ✓ | ✓ | Use iterative solver for \tilde{S}^{-1} | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Use factorization for J^{-1} | ✓ | | ✓ | | ✓ | | Use factorization for J^{-1} | ✓ | ✓ | ✓ | | ✓ | |
| Use iterative solver for J^{-1} | | ✓ | | ✓ | | ✓ | Use iterative solver for J^{-1} | | | ✓ | | ✓ | |

5 | NUMERICAL RESULTS

In the following, some results of numerical experiments that were conducted to validate the correctness and compare the efficiency of the proposed approaches are reported. The proposed tracing methods from Section 3 and the different approaches for the two fixed-point iterations considered in Section 4 were tested on several piecewise (non)linear

TABLE 7 Floating point operations for the signed fixed-point iteration Approach I (with precomputation of S , factorization for \tilde{S} , factorization for J), Approach II (with precomputation of S , linear solve for \tilde{S} , factorization for J), Approach III (no precomputation of S , linear solve for \tilde{S} , factorization for J), Approach IV (no precomputation of S , linear solve for \tilde{S} , linear solver for J), Approach V (almost matrix-free, linear solver for \tilde{S} , factorization for J), and Approach VI (matrix-free, linear solver for \tilde{S} , linear solver for J)

| Signed | Memory | FLOPs (once) | FLOPs (per iteration) |
|--------------|------------------------|------------------------------|----------------------------|
| Approach I | $O(s^2 + n^2 + ns)$ | $O(s^2 + s^2n + sn^2 + n^3)$ | $O(s^3)$ |
| Approach II | $O(s^2 + n^2 + ns)$ | $O(s^2 + s^2n + sn^2 + n^3)$ | $O(ls^2)$ |
| Approach III | $O(s^2/2 + n^2 + 2ns)$ | $O(n^3)$ | $O(l(s^2/2 + 2sn + n^2))$ |
| Approach IV | $O(s^2/2 + n^2 + 2ns)$ | -- | $O(l(s^2/2 + 2sn + kn^2))$ |
| Approach V | $O(s + n^2)$ | $O(n^3)$ | $O(ln^2)$ |
| Approach VI | $O(s + n)$ | -- | $O(lkn^2)$ |

Note. FLOPs = floating point operations.

TABLE 8 Algorithmic differentiation operations for the signed fixed-point iteration (4) using Approach I (with precomputation of S , factorization for \tilde{S} , factorization for J), Approach II (with precomputation of S , linear solve for \tilde{S} , factorization for J), Approach III (no precomputation of S , linear solve for \tilde{S} , factorization for J), Approach IV (no precomputation of S , linear solve for \tilde{S} , linear solver for J), Approach V (almost matrix-free, linear solver for \tilde{S} , factorization for J), and Approach VI (matrix-free, linear solver for \tilde{S} , linear solver for J)

| Signed | Taping | Forward-mode (once) | Forward-mode (per iteration) |
|---------------|----------------------|--------------------------|---|
| Approach I-IV | $O(1)\text{cost}(F)$ | $O(s + n)\text{cost}(F)$ | -- |
| Approach V | $O(4)\text{cost}(F)$ | $O(n)\text{cost}(F_4)$ | $O(l)\text{cost}(F_1, F_2, \text{ and } F_3)$ |
| Approach VI | $O(4)\text{cost}(F)$ | - | $O(lk)\text{cost}(F_4) + O(l)\text{cost}(F_1, F_2, \text{ and } F_3)$ |

TABLE 9 Algorithmic differentiation and floating point operations for the evaluation of c and Δx using Approach I (with precomputation of S , factorization for J), Approach II (with precomputation of S , linear solver for J), Approach III (no precomputation of S , factorization for J), Approach IV (no precomputation of S , linear solver for J), Approach V (almost matrix-free, factorization for J), and Approach VI (matrix-free, linear solver for J)

| Modulus+Signed | $c = \mathbf{a} - \mathbf{Z}\mathbf{J}^{-1}[\mathbf{b} - \Delta\mathbf{y}]$ | $\Delta\mathbf{x} = -\mathbf{J}^{-1}(\mathbf{b} - \Delta\mathbf{y} + \mathbf{Y} \Delta\mathbf{z})$ |
|---------------------------|---|---|
| Approach I+II(Signed)+III | $O(n^2 + sn)$ | $O(n^2 + sn)$ |
| Approach II(Modulus)+IV | $O(sn) + O(k)\text{cost}(F_4)$ | $O(sn) + O(k)\text{cost}(F_4)$ |
| Approach V | $O(n) + O(k)\text{cost}(F_4) + O(1)\text{cost}(F_1)$ | $O(n) + O(k)\text{cost}(F_4) + O(1)\text{cost}(F_3)$ |
| Approach VI | $O(n^2) + O(1)\text{cost}(F_1)$ | $O(n^2) + O(1)\text{cost}(F_3)$ |

examples. From all tested problems, only the results of two simple piecewise linear examples are reported[#] that represent two different classes of important applications: functions with dense and sparse ANFs.

Example 1. The first example is given by a test function $y = F(x)$, $F : \mathbb{R}^n \rightarrow \mathbb{R}^n$ that is piecewise linear itself and is represented by a dense ANF with a vector $z \in \mathbb{R}^s$ of s switching variables:

$$\begin{bmatrix} z \\ y \end{bmatrix} = \begin{bmatrix} a \\ b \end{bmatrix} + \begin{bmatrix} Z & L \\ J & Y \end{bmatrix} \begin{bmatrix} x \\ |z| \end{bmatrix}.$$

The entries of the vectors and matrices a , b , Z , L , J , and Y were random (i.i.d.) from $[-1, 1]$ and rescaled by the number of matrix columns. The matrix J was perturbed by the identity to satisfy the general assumptions required for the convergence of the two fixed-point iterations. Note that the fixed-point equation $\Delta z = c + S|\Delta z|$ can be reformulated⁸ as the linear complementarity problem $0 \leq u \equiv q + Mw \perp w \geq 0$ by decomposing $\Delta z = u - w$, where $M = (I - S)^{-1}(I + S)$ and $q = (I - S)^{-1}c$.

Example 2. The second example arises from a discretized version of the 2D obstacle problem for the standard Laplace equation $-\frac{\partial^2 x(\omega)}{\partial^2 \omega} = f(\omega)$ with Dirichlet boundary condition over a domain $\Omega \subset \mathbb{R}^2$, that is, find the function $x : \Omega \rightarrow \mathbb{R}$

[#] The results for the nonlinear examples are intentionally not presented because they would require a generalized Newton method (with an appropriate line search), which might have some undesired side effects on the run-time measurements. However, they were still used in some unreported experiments to validate that the proposed tracing approach yields the correct derivative information.

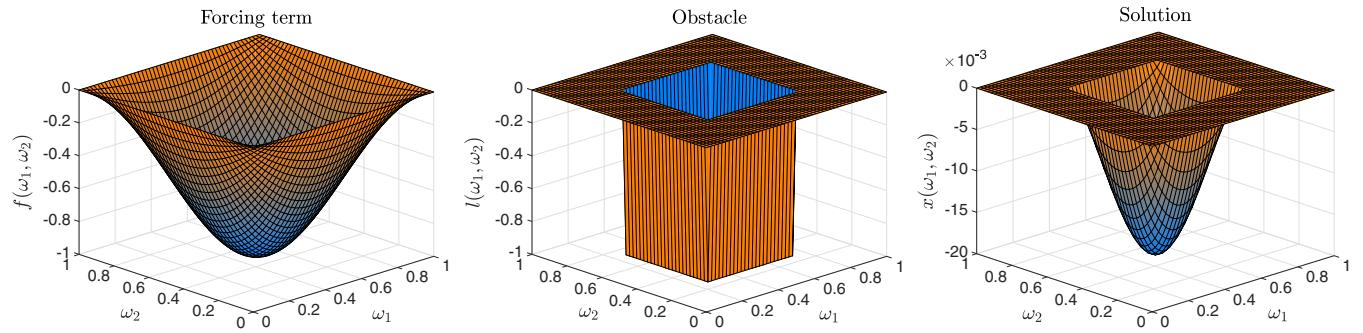


FIGURE 2 (Left) discrete approximation of the forcing term $f(\omega) = -\sin(\pi\omega_1)\sin(\pi\omega_2)$, (center) the obstacle function $l(\omega) = -\mathbb{1}_{\{\frac{1}{4} \leq \omega_1 \leq \frac{3}{4}, \frac{1}{4} \leq \omega_2 \leq \frac{3}{4}\}}$, and (right) resulting solution $x(\omega)$ for $\omega = (\omega_1, \omega_2) \in [0, 1]^2$ and $N = 100$

such that $x(\omega) = 0$ for all $\omega \in \partial\Omega$ and

$$0 \leq x(\omega) - l(\omega) \perp -\frac{\partial^2 x(\omega)}{\partial^2 \omega} - f(\omega) \geq 0, \quad \text{for almost all } (\omega_1, \omega_2) = \omega \in \Omega$$

for a prescribed forcing term $f : \Omega \rightarrow \mathbb{R}$ and obstacle function $l : \Omega \rightarrow \mathbb{R}$. Using the standard discrete Laplace operator $A \in \mathbb{R}^{N^2 \times N^2}$ for an equidistant discretization with mesh size $h = 1/(N - 1)$ of the unit square $\Omega = [0, 1]^2$ yields the discrete piecewise linear equation for $x \in \mathbb{R}^{N^2}$

$$0 = F(x) \equiv \min(x - l, -Ax - f) \in \mathbb{R}^{N^2},$$

where $l \in \mathbb{R}^{N^2}$ and $f \in \mathbb{R}^{N^2}$ denote the corresponding discrete vectors that approximate the test functions x , l , and f visualized in Figure 2. The identity $\min(u, v) = \frac{1}{2}(u + v - |u - v|)$ and the general propagation rules for piecewise linear AD² yield the piecewise linear model $\Delta y = F(x) + \Delta F(x; \Delta x)$ to approximate $F(x + \Delta x)$ with the following ANF:

$$\begin{bmatrix} \Delta z \\ \Delta y \end{bmatrix} = \begin{bmatrix} x + Ax - l + f \\ \frac{1}{2}(x - Ax - l - f) \end{bmatrix} + \begin{bmatrix} I + A & 0 \\ \frac{1}{2}(I - A) & -\frac{1}{2}I \end{bmatrix} \begin{bmatrix} \Delta x \\ |\Delta z| \end{bmatrix}.$$

Because the function F is piecewise linear itself, finding a solution x^* of $F(x) = 0$ is equivalent to finding a solution Δx^* of $F(x) + \Delta F(x; \Delta x) = 0$ with some fixed x and setting $x^* = x + \Delta x^*$.

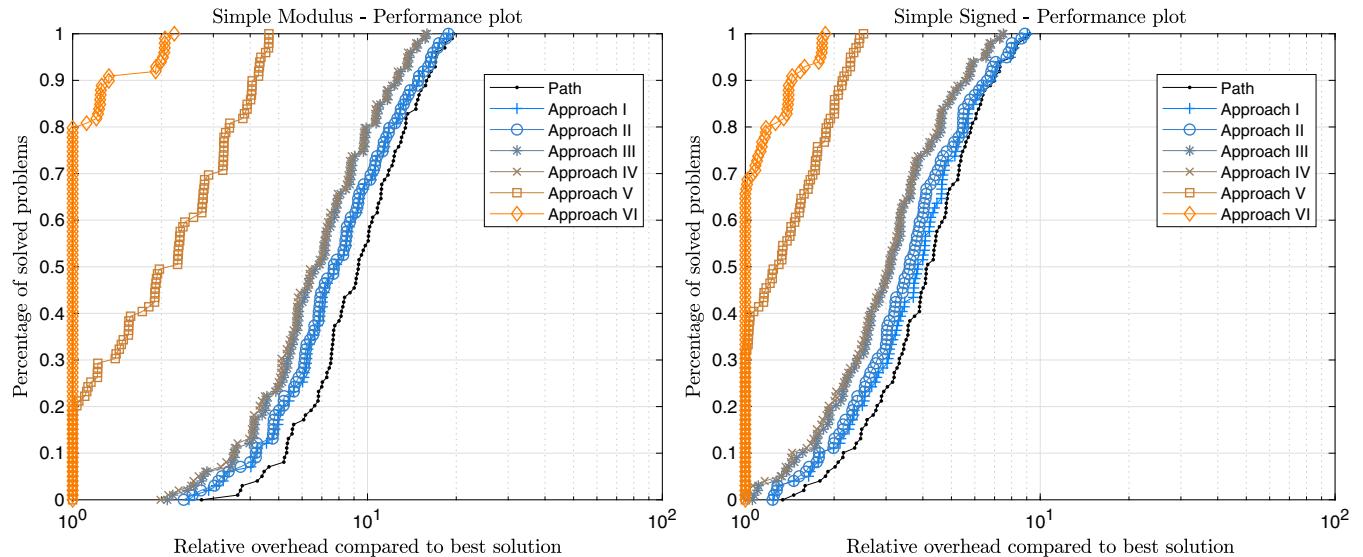


FIGURE 3 Performance plot for several test runs of Example 1 using different approaches for the (left) simple modulus and (right) signed fixed-point iteration with varying dimensions n, m , and $s \in \{200, 400, \dots, 2,000\}$

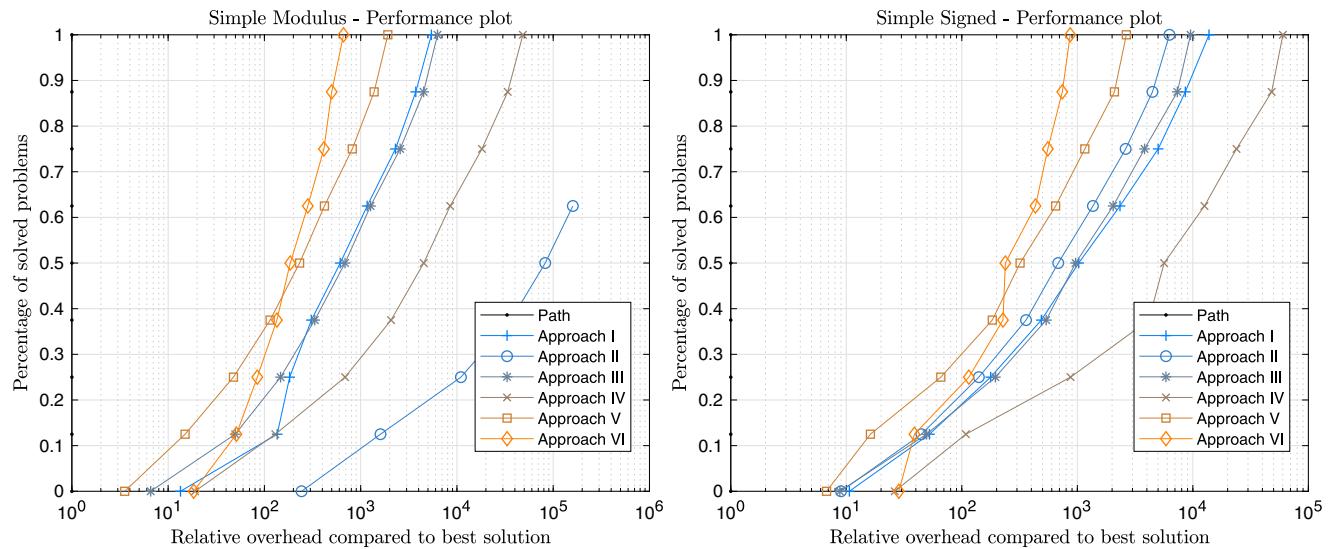


FIGURE 4 Performance plot for several test runs of Example 2 using different approaches for the (left) simple modulus and (right) signed fixed-point iteration with varying dimensions $n = m = s = N^2 \in \{20^2, 30^2, \dots, 100^2\}$

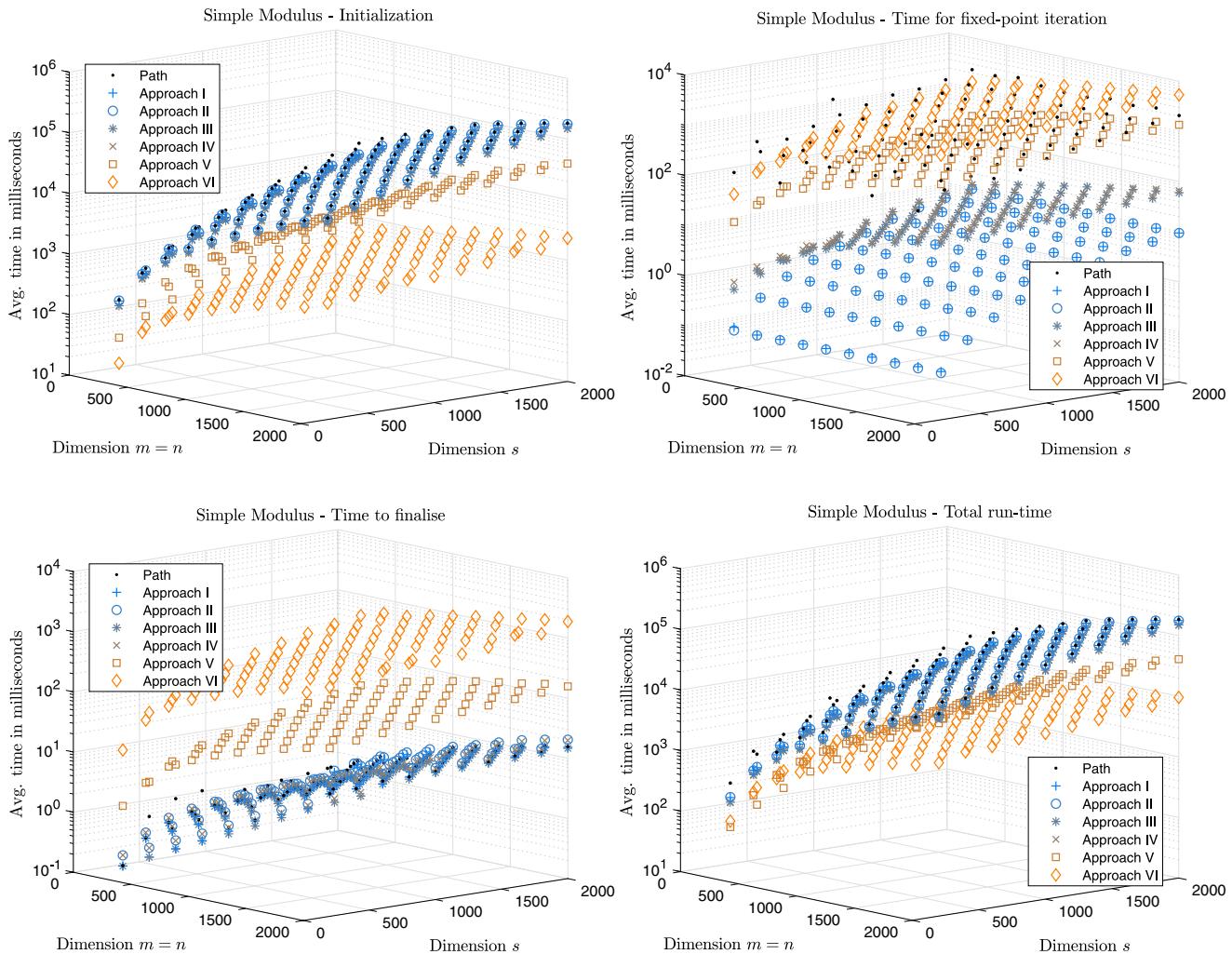


FIGURE 5 Average run-time results for the specific parts of the simple modulus method implementing the different approaches for Example 1: (top left) average time for initialization, (top right) average time for the iteration, (bottom left) average time to finalize, (bottom right) average overall run time

5.1 | Run-time results

Both examples were used to conduct several run-time experiments for the proposed approaches of the simple modulus and signed algorithm. For benchmarking, the examples were also reformulated as linear complementarity problem and solved by the latest version of PATH.^{14,21} Each of the experiments was performed on one of the standard Broadwell nodes of the Ara Cluster²² that uses Intel Xeon E5-2650v4 cores with 2.2 GHz and 128 GB of RAM on a dual socket mainboard (Intel C621) running CentOS Linux release 7.4. The different approaches were all implemented in standard C/C++ and compiled with GCC,²³ Version 4.8.5. The linear algebra used the dense methods of the basic linear algebra package BLAS²⁴ from Intel, Version 2018. The occurring linear (sub)problems were either solved by the methods `getrs` (with preceding `getrf`) or the standard BiCGStab without preconditioning.²⁵ The ANF entries were computed with the four tapes generated by the tracing procedure given in Section 3 and the latest version of ADOL-C 2.6.3²⁰ using either the methods `zos_forward`, `fos_forward`, or `Jacobian`. The reported measured run times (in milliseconds) represent an average of 100 repetitions to solve the problem $F(x) + \Delta F(x, \Delta x) = 0$ for random base points x and varying dimension. As usual, the stopping tolerance for the fixed-point iteration and linear solvers were set to machine precision 10^{-8} or if a maximal run-time threshold of order 10^6 was exceeded. The required tolerance was attained by all test runs within the maximal run time and number of iterations— 10^4 for the outer loop and 10^4 for each iterative solution with J —except of the simple modulus Approach II for Example 2 with $n = s = N^2 > 70^2 = 4,900$.

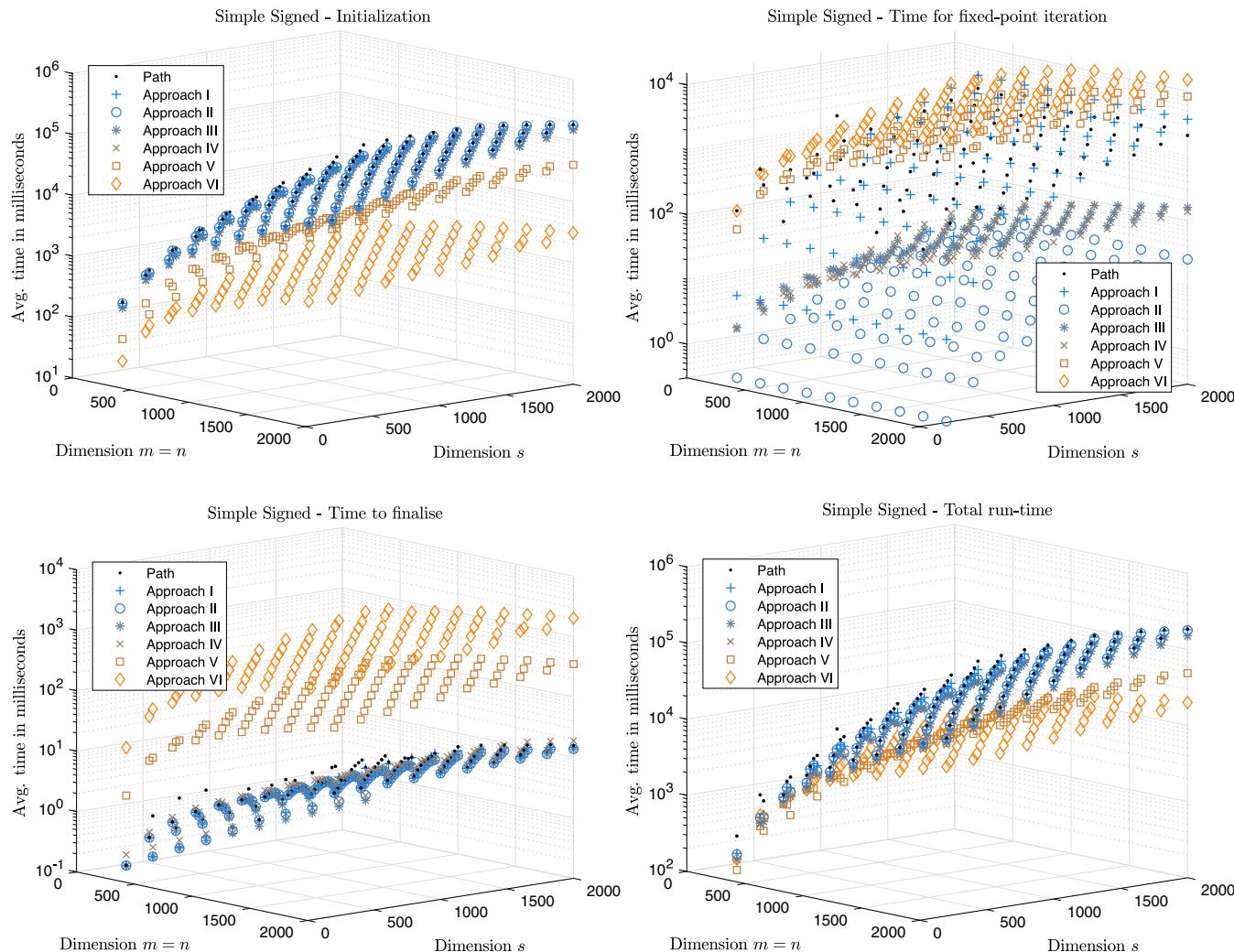


FIGURE 6 Average run-time results for the specific parts of the signed fixed-point method implementing the different approaches for Example 1: (top left) average time for initialization, (top right) average time for the iteration, (bottom left) average time to finalize, (bottom right) average overall run time

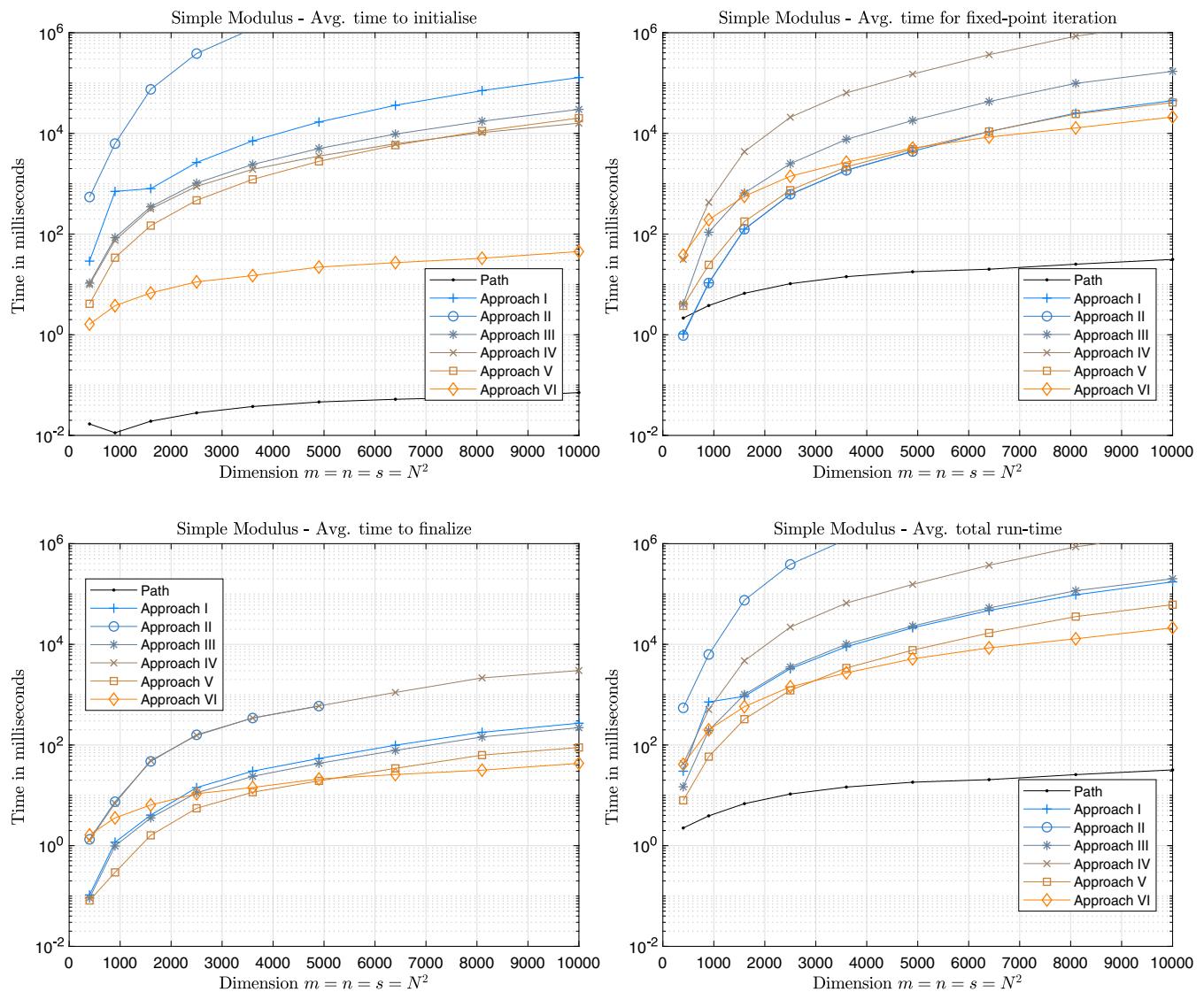


FIGURE 7 Average run-time results for the specific parts of the simple modulus method implementing the different approaches for Example 2: (top left) average time for initialization, (top right) average time for the iteration, (bottom left) average time to finalize, (bottom right) average overall run time

The summary of all tests for the two examples is presented in the performance plots in Figures 3 and 4. A more detailed overview for all approaches about the required run time for each of the subtasks (tracing, initialization, and effort for the fixed-point iteration) with respect to the dimension m , n , and s can be found in Figures 5–8.

All plots exhibit the expected behavior mentioned at the end of Section 4.1, namely, that the initial costs to compute the matrices Z , L , J , Y , and S are replaced by some additional effort required for the iteration of the (almost) matrix-free case. Thus, depending on the number of outer fixed-point iterations and the underlying problem structure (dimensions, sparsity pattern, ...), Approach V and Approach VI might yield a substantial benefit in terms of the total run time to solve the piecewise linear system, which is also depicted in the bottom right of Figures 5–8. The plots also indicate that the second approach of the simple modulus iteration failed for the second example due to exceeding the maximal run-time threshold in the initialization. This can be explained by a poor choice (BiCGStab without preconditioning) for the iterative solver to evaluate $J^{-1}Y = (I - A)^{-1}(-I)$ in the initialization, which is required to compute the matrix S . The latter argument and the special structure of the second sparse example are the main reasons why the proposed (almost) matrix-free methods were not able to compete in the experiments with a mature tool like the complementarity solver PATH. On the other hand, even this preliminary and very basic implementation of the different approaches were able to attain the same or even better performance than the more sophisticated tool PATH on the more general first example. As can be deduced from Figure 5

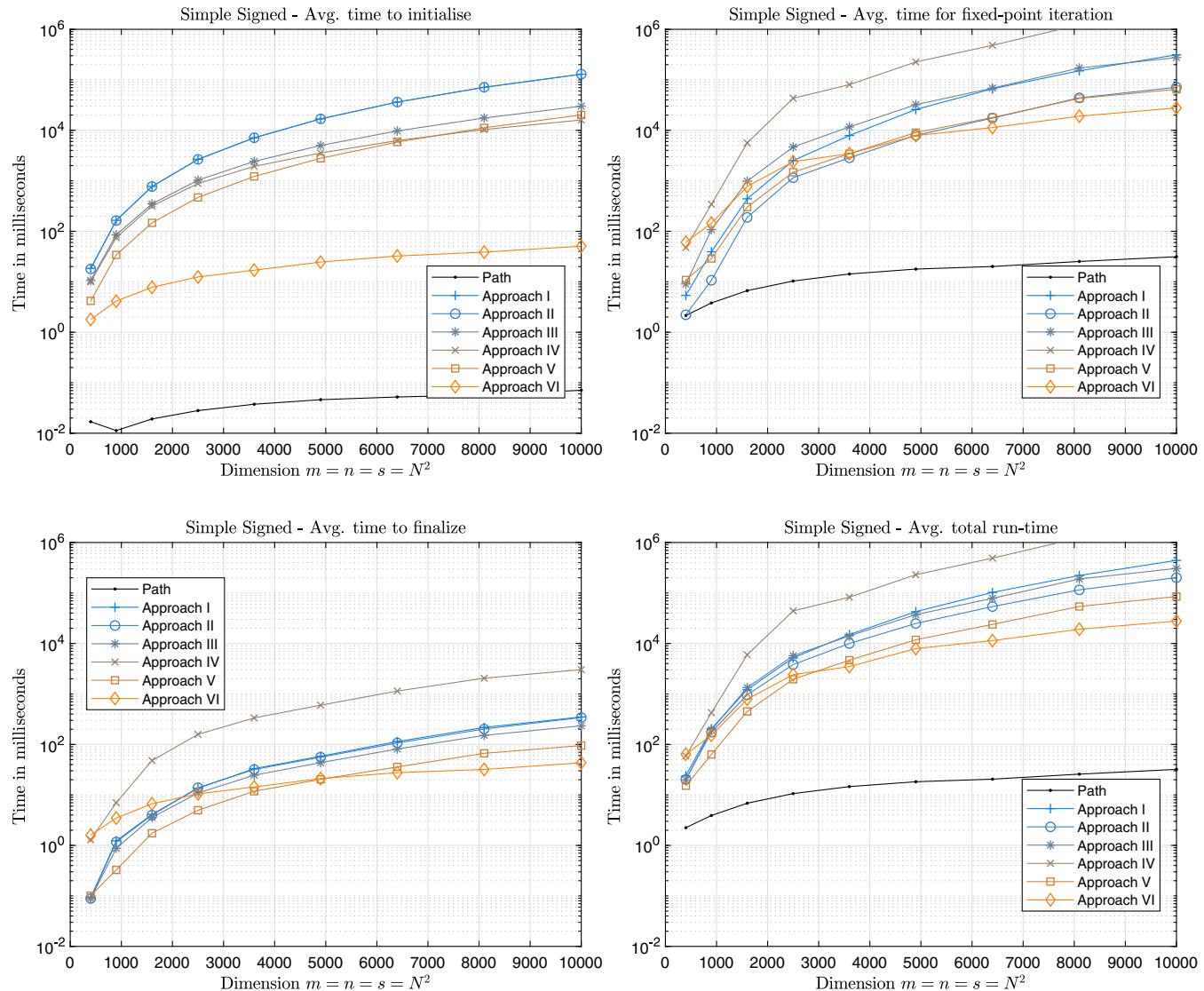


FIGURE 8 Average run-time results for the specific parts of the signed fixed-point method implementing the different approaches for Example 2: (top left) average time for initialization, (top right) average time for the iteration, (bottom left) average time to finalize, (bottom right) average overall run time

(upper plots), the speedup is not only due to the additional overhead in the initialization of PATH to compute the matrix M and q but also due to the time required to solve the LCP itself.

6 | CONCLUSIONS

In this paper, several implementations for the simple modulus and the signed fixed-point iteration were considered. The methods can be used to solve piecewise linear systems in ANF that, for example, arise as subproblems of a generalized Newton method. Therefore, a slight modification of the AD tool ADOL-C was presented that allows to evaluate certain parts of the ANF for general abs-factorable functions separately. It was shown how this modification can be used to define (almost) matrix-free variants of both methods, which only require matrix–vector products and do not require the explicit storage of the ANF or parts of it. For all resulting methods, explicit bounds on the numerical effort and memory requirements are given. The theoretical observations were validated for some numerical test examples. The numerical results indicate that the proposed (almost) matrix-free methods can yield a substantial performance improvement for certain examples.

ACKNOWLEDGEMENTS

This numerical results of this paper were performed on the ARA-Cluster at Friedrich-Schiller-Universität Jena, which was funded in part by support from the *EFRE Europäischer Fonds für regionale Entwicklung*, Project: *DAWI—Zusammenführung von Datenmanagement und-analyse in datengetriebenen Wissenschaften*.

CONFLICT OF INTEREST

The author declares no potential conflict of interests.

ORCID

Torsten Bosse  <https://orcid.org/0000-0002-4894-0742>

REFERENCES

1. Streubel T, Griewank A, Radons M, Bernt JU. Representation and analysis of piecewise linear functions in abs-normal form. System modeling and optimization: 26th IFIP TC 7 Conference, CSMO 2013, Klagenfurt, Austria, September 9-13, 2013, Revised Selected Papers. Berlin, Germany: Springer Berlin Heidelberg, 2014; p. 327–336. Available from: https://doi.org/10.1007/978-3-662-45504-3_32
2. Griewank A. On stable piecewise linearization and generalized algorithmic differentiation. Optim Methods Softw. 2013;28(6):1139–1178. Available from: <https://doi.org/10.1080/10556788.2013.796683>
3. Bokhoven WMV. Piecewise linear modeling and analysis. Dordrecht, The Netherlands: Kluwer Academic Publishers; 1981.
4. Corliss GF, Griewank A. Operator overloading as an enabling technology for automatic differentiation. Center for Research on Parallel Computation; 1993. Technical report 93431.
5. Griewank A, Walther A. Evaluating derivatives: Principles and techniques of algorithmic differentiation. 2nd ed. Philadelphia, PA: Society for Industrial and Applied Mathematics; 2008. Available from: <https://books.google.com/books?id=xoiiLaRxcbEC>
6. Naumann U. The art of differentiating computer programs: An introduction to algorithmic differentiation. Philadelphia, PA: Society for Industrial and Applied Mathematics; 2012.
7. Walther A, Griewank A, Vogel O. ADOL-C: automatic differentiation using operator overloading in C++. PAMM Proc Appl Math Mech. 2003;2:41–44.
8. Griewank A, Bernt J-U, Radons M, Streubel T. Solving piecewise linear systems in abs-normal form. Linear Algebra Appl. 2015;471:500–530.
9. Mangasarian OL. Absolute value programming. Comput Optim Appl. 2007;36(1):43–53. Available from: <https://doi.org/10.1007/s10589-006-0395-5>
10. Bai Z-Z, Zhang L-L. Modulus-based synchronous multisplitting iteration methods for linear complementarity problems. Numer Linear Algebra Appl. 2013;20(3):425–439. Available from: <https://doi.org/10.1002/nla.1835>
11. Bai Z-Z, Zhang L-L. Modulus-based synchronous two-stage multisplitting iteration methods for linear complementarity problems. Numerical Algorithms. 2013;62(1):59–77. Available from: <https://doi.org/10.1007/s11075-012-9566-x>
12. Brugnano L, Casulli V. Iterative solution of piecewise linear systems. SIAM J Sci Comput. 2008;30(1):463–472. Available from: <https://doi.org/10.1137/070681867>
13. Radons M. Direct solution of piecewise linear systems. Theor Comput Sci. 2016;626:97–109.
14. Dirkse SP, Ferris MC. The path solver: a nonmonotone stabilization scheme for mixed complementarity problems. Optim Methods Softw. 1995;5(2):123–156. Available from: <https://doi.org/10.1080/10556789508805606>
15. Clarke FH. Optimization and nonsmooth analysis. New York, NY: Wiley; 1983. Canadian Mathematical Society series of monographs and advanced texts. Available from: <https://books.google.de/books?id=UErvAAAAMAAJ>
16. Qi L, Sun J. A nonsmooth version of Newton's method. Math Program. 1993;58:353–367.
17. Ulbrich M. Semismooth Newton methods for variational inequalities and constrained optimization problems in function spaces. Philadelphia, PA: Society for Industrial and Applied Mathematics; 2011. Available from: <http://pubs.siam.org/doi/abs/10.1137/1.9781611970692>
18. Bosse TF, Narayanan SHK. Study of the numerical efficiency of structured abs-normal forms. Optim Methods Softw. 2019. Available from: <https://doi.org/10.1080/10556788.2019.1613654>
19. Walther A, Griewank A. Getting started with ADOL-C. In: Naumann U, Schenk O, editors. Combinatorial scientific computing. Boca Raton, FL: CRC Press, 2012; p. 181–202.
20. Walther A, Griewank A. ADOL-C (2.6.2.) documentation and code. Coin-OR Repository. 2019. Available from: <https://projects.coin-or.org/ADOL-C>
21. Munson T, Ferris M. Path solver (4.7.03) documentation and code. Github Repository. 2019. Available from: <https://github.com/ampl/pathlib>
22. Jena FSU. Hardware specifications of the ARA-cluster. 2017. https://www.uni-jena.de/Universitat/Einrichtungen/URZ/Dienste/Computeserver/Ara_Cluster-p-506533/Hardware.html

23. Stallman RM, GCC DeveloperCommunity. Using the GNU compiler collection: A GNU manual for GCC Version 4.3.3. Paramount, CA: CreateSpace; 2009.
24. Netlib. BLAS (basic linear algebra subprograms). 2017. <http://www.netlib.orgblas>
25. Barrett R, Berry MW, Chan TF, et al. Templates for the solution of linear systems: Building blocks for iterative methods. Philadelphia, PA: Society for Industrial and Applied Mathematics; 1994. Other titles in applied mathematics. Available from: <https://math.nist.gov/iml++/bicgstab.h.txt>

How to cite this article: Bosse T. (Almost) matrix-free solver for piecewise linear functions in abs-normal form. *Numer Linear Algebra Appl.* 2019;26:e2258. <https://doi.org/10.1002/nla.2258>