A globally convergent method for finding all steady-state solutions of distillation columns

(Supplementary material)

Ali Baharev, Arnold Neumaier

Contents

This document gives additional explanation of the algorithm presented in [1]; it is therefore assumed that the reader has already read that paper.

1 An illustrative linear example

The following system of linear equations is being solved.

$$\begin{aligned}
x_1 + 10x_{21} &= 11 \\
10x_1 + x_2 + x_{21} &= 12 \\
x_1 + 10x_2 + x_3 &= 12 \\
x_2 + 10x_3 + x_4 &= 12 \\
&\vdots \\
x_{18} + 10x_{19} + x_{20} &= 12 \\
x_{19} + 10x_{20} &= 11
\end{aligned} \tag{1}$$

The sparsity pattern of the coefficient matrix is shown in Figure 1; the problem is in bordered lower triangular form. The exact solution is $x_i = 1$ $(i = 0 \ldots 20)$. The condition number of the coefficient matrix is approximately 1.49.

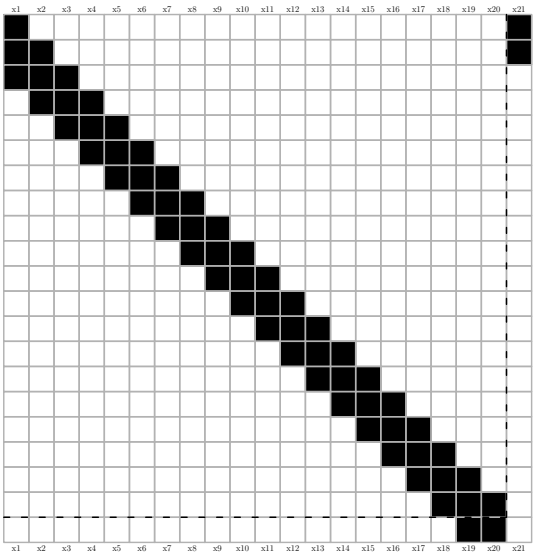

Figure 1: The sparsity pattern of the coefficient matrix of problem (1). The leading submatrix, surrounded by dashed lines, is singular to working precision.

1.1 The sequential approach

That family of methods that work in the same fashion as the as the stage-by-stage method [4] or the sequential modular approach to process flowsheeting and optimization [2, Chapter 8] will be referred to

as the sequential approach. The goal of this section is to show why the sequential approach is prone to fail.

For any linear problem in bordered lower triangular form (BLTF) the sequential approach is essentially equivalent to Gaussian elimination without pivoting. In many cases, although the total system is well-conditioned, the coefficient matrix of any leading $m \times m$ submatrix has a condition number that grows exponentially with m and ultimately becomes singular to working precision. This causes the Gaussian elimination, and hence the sequential approach, to fail.

A typical example is (1). We choose x_{21} as the "tear variable." Given an initial estimate for x_{21}, the remaining variables can be determined recursively as follows.

$$\begin{aligned} x_1 &= -10x_{21} + 11 \\ x_2 &= -x_{21} - 10x_1 + 12 \\ x_3 &= -x_1 - 10x_2 + 12 \\ x_4 &= -x_2 - 10x_3 + 12 \\ &\vdots \\ x_{20} &= -x_{18} - 10x_{19} + 12 \end{aligned} \tag{2}$$

The sequential approach would then check the residual

$$r := x_{19} + 10x_{20} - 11 \tag{3}$$

of the final equation and iterate, by changing the initial guess for x_{21} appropriately, until r becomes smaller than a predefined threshold.

The above explicit formulas (2) are evaluated in floating-point arithmetic; the ANSI C code is in Listing 4. **The initial estimate is $\mathbf{x_{21} = 1 + 10^{-10}}$, which is far closer to the exact solution than one could hope for in any practical application.** The following values are obtained.

Listing 1: Sequential approach with initial estimate $x_{21} = 1 + 10^{-10}$

```
x1  =  0.9999999990
x2  =  1.0000000099
x3  =  0.9999999020
x4  =  1.0000009701
x5  =  0.9999903970
x6  =  1.0000950599
x7  =  0.9990590039
x8  =  1.0093149009
x9  =  0.9077919874
x10 =  1.9127652254
x11 = -8.0354442416
x12 =  90.441677191
x13 = -884.38132766
x14 =  8765.3715994
x15 = -86757.334667
x16 =  858819.97507
x17 = -8501430.4160
x18 =  84155496.185
x19 = -833053519.44
x20 =  8246379710.2
```

As it is can be seen in Listing 1, the error in x_i grows exponentially with i; the x_{10}–x_{20} values have no correct significant digits. To explain this observation, we analyze how the eliminated variables x_i $(i = 1 \ldots 20)$ depend on the initial guess x_{21}. Since (6) is linear, all the eliminated variables can be expressed as some appropriate linear function of x_{21},

$$x_i = a_i x_{21} + b_i \qquad (i = 1 \ldots 20). \tag{4}$$

Equations (2) build up this dependence (4) between the eliminated variables and the initial guess by recursion.

The actual value of the coefficients a_i and b_i are sought. For reasons that will soon become apparent, we include the identity

$$x_{21} = a_{21} x_{21} + b_0 \text{ where } a_0 = 1, \quad b_0 = 0.$$

From the first equation $10x_{21} + x_1 = 11$ we have

$$x_1 = -10x_{21} + 11,$$

that is,

$$a_1 = -10, \quad b_1 = 11.$$

From the equations

$$x_{i-1} + 10x_i + x_{i+1} = 12 \qquad (i = 1 \ldots 19)$$

with substitutions $x_{i-1} = a_{i-1} x_{21} + b_{i-1}$ and $x_i = a_i x_{21} + b_i$ we get

$$(a_{i-1} x_{21} + b_{i-1}) + 10(a_i x_{21} + b_i) + x_{i+1} = 12,$$

and by simple rearrangements

$$x_{i+1} = (-10a_i - a_{i-1}) x_{21} + (-10b_i - b_{i-1} + 12).$$

That is, for $i = 1 \ldots 19$ we have

$$\begin{aligned} a_{i+1} &= -10a_i - a_{i-1} \\ b_{i+1} &= -10b_i - b_{i-1} + 12. \end{aligned} \tag{5}$$

with the initial values $a_0 = 1$, $a_1 = -10$, $b_0 = 0$, $b_1 = 11$. The values of a_i and b_i with 16 digit precision are given in Listing 2, the ANSI C code of the program used for the computations are given in Listing 4.

Listing 2: Coefficients a_i and b_i in $x_i = a_i x_{21} + b_i$ with 16 digit precision, obtained by recursion (5).

```
a0  =  1
b0  =  0

a1  = -10
b1  =  11

a2  =  99
b2  = -98

a3  = -980
b3  =  981

a4  =  9701
b4  = -9700

a5  = -96030
b5  =  96031

a6  =  950599
b6  = -950598

a7  = -9409960
b7  =  9409961

a8  =  93149001
b8  = -93149000

a9  = -922080050
b9  =  922080051

a10 =  9127651499
b10 = -9127651498

a11 = -90354434940
b11 =  90354434941

a12 =  894416697901
b12 = -894416697900

a13 = -8853812544070
b13 =  8853812544071

a14 =  87643708742799
b14 = -87643708742798

a15 = -867583274883920
b15 =  867583274883921

a16 =  8588189040096401
b16 = -8588189040096400

a17 = -8.501430712608010e+16
b17 =  8.501430712608010e+16

a18 =  8.415548822207046e+17
b18 = -8.415548822207046e+17

a19 = -8.330534515080966e+18
b19 =  8.330534515080966e+18

a20 =  8.246379026858897e+19
b20 = -8.246379026858897e+19
```

The magnitude of $|a_i|$ is 10^i. In the recursion (5), a_i is multiplied by 10 in each step which explains the exponential growth of $|a_i|$. It is now clear that a small error in our initial estimate for x_{21} causes a 10^i time larger error in x_i. **This is the fundamental reason why the sequential approach fails:**

The method can become (and in fact typically becomes – in the distillation applications) **extremely sensitive to the initial estimate, up to a point where solving the problem is notoriously difficult or even impossible.**

1.2 Permutation to lower block Hessenberg form

The proposed method requires that the problem is first put in lower block Hessenberg form. It is trivial to permute the bordered lower triangular form shown in Figure 1 into lower block Hessenberg form: The last column (border), corresponding to x_{21}, is moved to the first position and x_{21} is relabeled to x_0 ($x_{21} \equiv x_0$). The system of equations (1) becomes

$$\begin{aligned} 10x_0 + x_1 &= 11 \\ x_0 + 10x_1 + x_2 &= 12 \\ x_1 + 10x_2 + x_3 &= 12 \\ &\vdots \\ x_{18} + 10x_{19} + x_{20} &= 12 \\ x_{19} + 10x_{20} &= 11. \end{aligned} \tag{6}$$

The sparsity pattern is shown in Figure 2. The coefficient matrix is tridiagonal, well-conditioned and diagonally dominant, thus the problem is easily solvable by Gaussian elimination or with successive substitution.

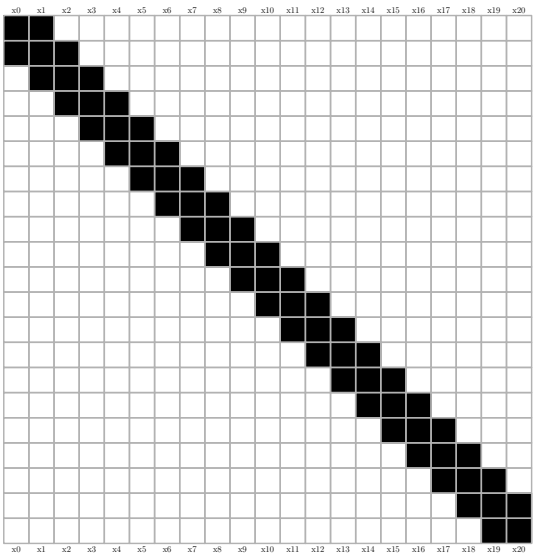

Figure 2: Sparsity pattern of the coefficient matrix.

1.3 Solving the linear example with the proposed method

The only purpose of this subsection is to demonstrate *how* the proposed method works as linear examples do not reveal the difficulties encountered only in the nonlinear case, see Section 2.

The presentation here strictly adheres to the steps of Section *3.1. Formal statement* of BAHAREV & NEUMAIER [1].

Input: The tridiagonal coefficient matrix of (6), which has the desirable sparsity pattern (lower block Hessenberg form, block size is 1×1), see Figure 2. Variable bounds are not considered to make the presentation simpler.

External dependency: The general proposed method in [1] depends on an auxiliary algorithm to parameterize the solution set of an underdetermined system of equations. We chose the trivial parameterization

$$x_i = s_i \qquad (i = 1 \dots 20). \tag{7}$$

This is possible only because of the simplicity of the example. To keep the presentation simple, bounds on the variables are not considered so that s_i are unbounded as well.

Initialization: The algorithm starts with the parameterization $x_0 = s_0$.

Forward sweep: For $i = 1 \dots 20$

Both the system (6) and the parameterizations (7) are linear, hence the transition maps relating s_{i-1} and s_i are linear as well: $s_{i-1} = a_i s_i + b_i$ $(i = 1 \dots 20)$. *In the forward sweep, the coefficients a_i and b_i are computed and stored;* the details are elaborated below.

If $i = 1$, from the first equation of (6) with the $x_0 = s_0$ substitution we get $10s_0 + x_1 = 11$. Considering s_0 and x_1 as variables, the solution set of this linear equation can be parameterized with a single parameter s_1. The obvious parameterization gives:

$$\begin{aligned} x_1 &= s_1 \\ s_0 &= (11 - s_1)/10 = -0.1 s_1 + 1.1. \end{aligned}$$

That is, $s_0 = a_1 s_1 + b_1$ where $a_1 = -0.1$ and $b_1 = 1.1$.

If $i > 1$, from the ith equation $x_{i-2} + 10x_{i-1} + x_i = 12$ with the $x_{i-2} = s_{i-2}$, $s_{i-2} = a_{i-1}s_{i-1} + b_{i-1}$ and $x_{i-1} = s_{i-1}$ substitutions we get

$$(a_{i-1}s_{i-1} + b_{i-1}) + 10s_{i-1} + x_i = 12.$$

Given the parameterization $x_i = s_i$, simple rearrangements yield

$$s_{i-1} = \frac{-s_i - b_{i-1} + 12}{a_{i-1} + 10}.$$

That is, we have the following recursions

$$a_i = \frac{-1}{a_{i-1}+10} \qquad \text{and} \qquad b_i = \frac{-b_{i-1}+12}{a_{i-1}+10}$$

with $a_1 = -0.1$ and $b_1 = 1.1$.

Solving the final system: The $x_{19} = a_{20}s_{20} + b_{20}$, $x_{20} = s_{20}$ relations, obtained in the last iteration of the forward sweep, are substituted into the last equation $x_{19} + 10x_{20} = 11$ of (6), yielding

$$(a_{20}s_{20} + b_{20}) + 10s_{20} = 11.$$

This equation has the analytic solution

$$s_{20} = \frac{-b_{20}+11}{a_{20}+10}. \tag{8}$$

Backward sweep: For $i = 20 \ldots 1$

$x_{20} = s_{20}$, the remaining components are recovered by passing through the transition maps recursively

$$s_{i-1} = a_i s_i + b_i, \tag{9}$$

and recovering the solution at each block from the parameterization

$$x_{i-1} = s_{i-1}. \tag{10}$$

The algorithm ends here.

This algorithm has been implemented in ANSI C, see Listing 4. When run, the program prints the correct solution $x_i = 1$ $(i = 0 \ldots 20)$.

1.4 Sensitivity of the solution to the initial errors

The recursions (2) and (9), (10) are seemingly similar in the sense that given an initial value for one variable they both determine the remaining variables. In case of (2), an initial *guess* for x_0 is needed; in case of (9), $s_{20}(= x_{20})$ is *given* by (8).

The fundamental issue with sequential approach was that any error in x_0 grew exponentially with i. To compare the proposed method to the sequential approach in this respect, the sensitivity of $(x_i =)s_i$ $(i = 0 \ldots 20)$ to s_{20} is analyzed here, similarly to (4). Due to linearity, s_i is a linear function of s_{20},

$$s_i = \lambda_i s_{20} + \mu_i. \tag{11}$$

The goal is to determine the coefficients λ_i and μ_i. Substituting (11) to the transition maps $s_{i-1} = a_i s_i + b_i$

gives

$$\lambda_{i-1}s_{20} + \mu_{i-1} = a_i(\lambda_i s_{20} + \mu_i) + b_i.$$

From this, with simple rearrangements, we conclude that

$$\begin{aligned} \lambda_{i-1} &= a_i\lambda_i \\ \mu_{i-1} &= a_i\mu_i + b_i \end{aligned} \tag{12}$$

for $i = 20 \ldots 1$; the initial values $\lambda_{20} = 1$ and $\mu_{20} = 0$ are from the identity $s_{20} = 1 \cdot s_{20} + 0$. The coefficients λ_i and μ_i are computed in floating-point arithmetic, see Listing 3. The ANSI C source code of the program is in Listing 4.

Listing 3: Coefficients λ_i and μ_i in $s_i = \lambda_i s_{20} + \mu_i$ obtained by recursion (12).

```
lambda20 =  1.000000000000000e+00          mu20 =  0.0000000000000000
lambda19 = -1.010205144336438e-01          mu19 =  1.1010205144336438
lambda18 =  1.020514433643804e-02          mu18 =  0.9897948556635620
lambda17 = -1.030928930736557e-03          mu17 =  1.0010309289307366
lambda16 =  1.041449709275333e-04          mu16 =  0.9998958550290724
lambda15 = -1.052077853877629e-05          mu15 =  1.0000105207785388
lambda14 =  1.062814460229621e-06          mu14 =  0.9999989371855398
lambda13 = -1.073660635199117e-07          mu13 =  1.0000001073660636
lambda12 =  1.084617496949676e-08          mu12 =  0.9999999891538250
lambda11 = -1.095686175055874e-09          mu11 =  1.0000000010956862
lambda10 =  1.106867810619759e-10          mu10 =  0.9999999998893132
lambda9  = -1.118163556388491e-11          mu9  =  1.0000000000111817
lambda8  =  1.129574576873180e-12          mu8  =  0.9999999999988705
lambda7  = -1.141102048468942e-13          mu7  =  1.0000000000001141
lambda6  =  1.152747159576159e-14          mu6  =  0.9999999999999885
lambda5  = -1.164511110721751e-15          mu5  =  1.0000000000000011
lambda4  =  1.176395114559170e-16          mu4  =  0.9999999999999998
lambda3  = -1.188400383741868e-17          mu3  =  1.0000000000000000
lambda2  =  1.200526918269846e-18          mu2  =  1.0000000000000000
lambda1  = -1.212653452797825e-19          mu1  =  1.0000000000000000
lambda0  =  1.212653452797825e-20          mu0  =  1.0000000000000000
```

The λ_i are decreasing exponentially as i decreases. It means that any error in s_{20} also diminishes in an exponential rate. Thus, the proposed method is numerically stable. This is the exact opposite of the sequential approach: As it was shown in Listing 1, any error in the initial value of the recursion grew exponentially as recursion (2) progressed, making the sequential method numerically unstable.

1.5 Comparison of the sequential and the proposed method

Given an initial estimate for $x_{21}(\equiv x_0)$, the sequential approach determines all remaining x_i $(i = 1 \ldots 20)$ in one pass, see (2). Then, the initial estimate is updated based on the residual of the final equation (3) and all the variables are recomputed. This procedure is repeated until convergence. In contrast, the proposed method reaches convergence in just two passes: the forward and the backward sweep.

The sequential approach requires an initial estimate. Even a tiny error in the initial estimate grows exponentially with each step of (2) and makes the sequential method numerically unstable.

The proposed method does not require any initial estimate; unlike the sequential approach, the first

variable $x_0(\equiv x_{21})$ used to start the computations does not play any special role. The value to start the backward sweep (9), $s_{20}(=x_{20})$ in this example, does not require guesses; it is a result of computations. Any error (only rounding errors in case of this example) diminishes in an exponential rate with each step of the backward sweep, that is, the proposed method is numerically stable.

2 The nonlinear case: a reactive distillation column

In the linear case, the reparameterization simplifies to permuting, scaling and shifting so it does not reveal additional difficulties that arise only in the nonlinear case. These difficulties become visible when computing the reactive distillation column in [1], producing glycol from ethylene oxide and water. For this distillation column $d_0 = 1$.

2.1 Notes on the forward sweep

The ith block represents a $d_i \times (d_0 + d_i)$ underdetermined system of equations

$$F_i(x_0^{(i-1)}(s_{i-1}),\ldots,x_{i-1}^{(i-1)}(s_{i-1}),x_i) = 0 \qquad (i = 1,\ldots,N), \tag{13}$$

with d_0 superficial degrees of freedom, considering s_{i-1} and x_i as variables ($\dim F_i = d_i$; $\dim s_{i-1} = d_0$, $\dim x_i = d_i$). The solutions set of (13) is a d_0-dimensional manifold ("hyper-surface") in the $(d_0 + d_i)$-dimensional space of the variables.

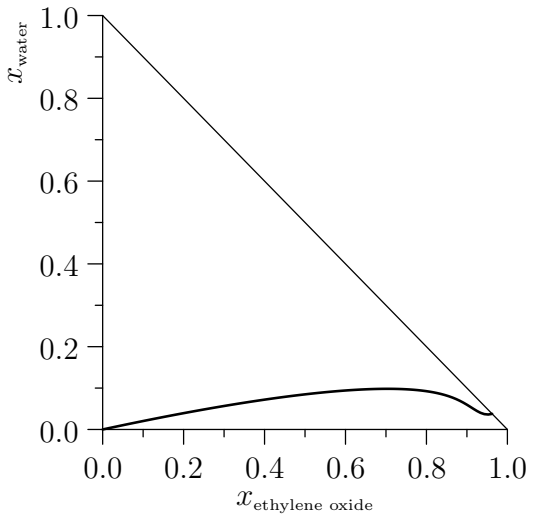

Figure 3: The solution set of block 1 (stage 1), projected to the 2D plane; mole fractions in the liquid phase entering stage 1 are on the axes.

We want to plot this manifold for the reactive distillation column in [1]. Since $d_0 = 1$, the solution set of (13) is just a curve for each F_i $(i = 1,\ldots,N)$. This curve must be appropriately projected to the 2D plane before it can be plotted since it is in a $(d_0 + d_i)$-dimensional space, where $d_1 = 9$ and $d_i = 15$

($i = 2, \ldots, N$). Hence, the mole fraction of the ethylene oxide and water in the liquid stream entering stage i (as determined by the solution set of block i) are selected for plotting. Figure 3 shows the results of this projection for stage 1, which is the stage just above the reboiler.

System (13) becomes properly-determined if d_0 of the variables are fixed; in other words, the solution set can be parameterized by d_0 parameters. The parameterizations $x_k^{(i-1)}(s_{i-1})$ $(k = 0, \ldots, i-1)$ appearing in (13) were obtained in the previous iteration step of the forward sweep (and the algorithm was initialized with a trivial linear parameterization before entering the forward sweep). First, let us assume that the solution set is parameterized by s_{i-1} ($\dim s_{i-1} = d_0$), as obtained from the previous iteration step. Figure 4a shows the points corresponding to $s_{i-1} = \{0.0,\ 0.25,\ 0.50,\ 0.75,\ 1.00\}$ ($i = 1$, $\lambda = 0.033$). It is obvious from the figure that this parameterization is undesirable: the distribution of these points are very uneven, the parameterization is sensitive to small changes in s_{i-1} for small s_{i-1} and insensitive for s_{i-1} values close to 1. However, if the curve is reparameterized by (normalized) arc length, we get a well-conditioned new parameterization by s_i. Figure 4b shows the points corresponding to $s_i = \{0.0,\ 0.25,\ 0.50,\ 0.75,\ 1.00\}$. This is the desirable parameterization. Figure 4c shows how the old and the new parameterizations are related.

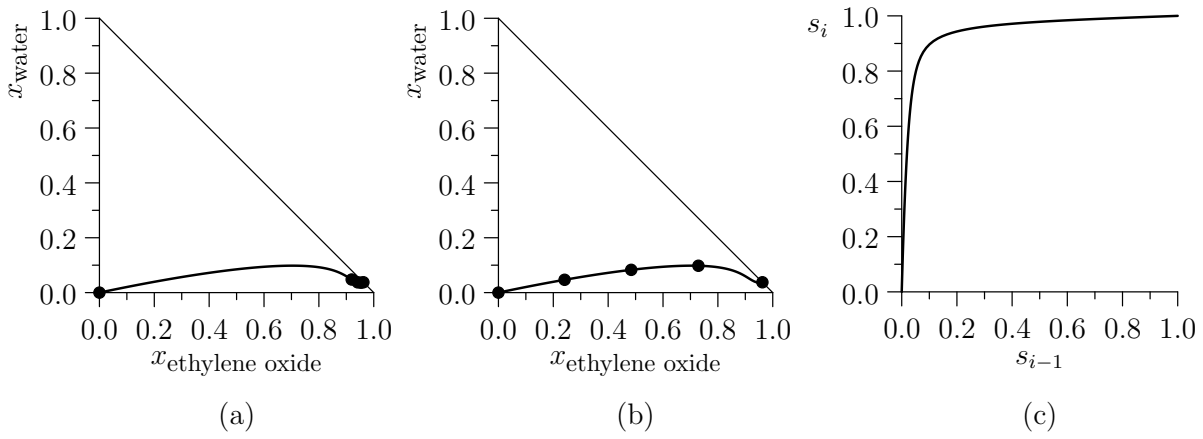

Figure 4: (a) The solution set of block 1, the points correspond to the parameter values $s_{i-1} = \{0.0,\ 0.25,\ 0.50,\ 0.75,\ 1.00\}$, where the $x(s_{i-1})$ parameterization was obtained in the previous iteration step of the forward sweep. (b) After reparameterization, the points correspond to the parameter values $s_i = \{0.0,\ 0.25,\ 0.50,\ 0.75,\ 1.00\}$. (c) The relation between the two parameterizations.

The forward sweep builds and saves the parameterizations $x_k = x_k^{(i)}(s_i)$ $(k = 0, \ldots, i)$ together with the transition maps $s_{i-1} = s_{i-1}(s_i)$ in a recursive fashion.

2.2 Notes on solving the final system

After the forward sweep has finished, the properly-determined $d_0 \times d_0$ system

$$F_{N+1}(x_0^{(N)}(s_N), \ldots, x_N^{(N)}(s_N)) = 0 \tag{14}$$

is obtained at block $N+1$. All solutions of this system are sought. We have $d_0 = 1$ for the reactive distillation column computed in [1], that is, the residual $F_{N+1}(x_0^{(N)}(s_N), \ldots, x_N^{(N)}(s_N))$ in (14), can be conveniently plotted as the function of s_N. Figure 5 shows the numerical results. Finding all solutions

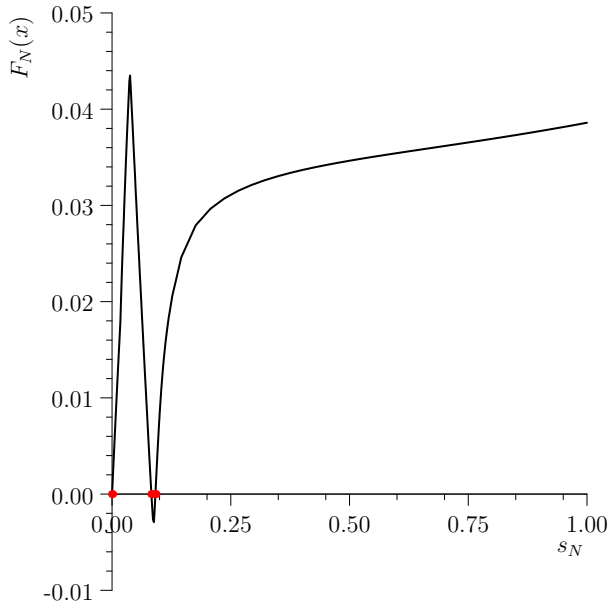

Figure 5: Solving the final equation, residual as the function of s_N. Red dots are shown at the solutions. The column has 3 steady-states at bifurcation parameter $\lambda = 0.033$.

is easy, only the sign changes have to be found. The absence of sign changes indicates infeasibility; the original system of equations has no solution in this case.

2.3 Implementation of the reparameterization step

For those interested in the low-level details, the C++ implementation is available in the online supplementary material of BAHAREV & NEUMAIER [1].

As stated earlier, $d_0 = 1$ for the reactive distillation column under study. Hence, a 1D manifold is reparameterized in each iteration step of the forward sweep. It is relatively easy to handle, as a 1D manifold is just a union of curves and the arc length is the natural parameter. Therefore, **the current implementation works only for 1D manifolds, parameterized by arc length.** However, it must be emphasized here, that the *algorithm* by BAHAREV & NEUMAIER [1] is not limited to the 1D case, only the current *implementation* is. Parameterizing multidimensional manifolds can be done [3] but is significantly more complex (work in progress).

A 1D manifold consists of infinitely many points but it has to be approximated with finitely many parameters in practice. That is, the manifold has to be discretized. A simple **piecewise linear approximation to 1D manifolds is implemented**: The 1D manifold is discretized as an ordered set of points and linear interpolation is applied between the neighboring points.

In the initialization step, the trivial linear parameterization of x_0

$$x_0 = x_0^{(0)}(s_0) = \underline{x}_0 + (\overline{x}_0 - \underline{x}_0)s_0, \qquad s_0 \in [0,1]^{d_0}, \tag{15}$$

is discretized as $m+1$ equidistant breakpoints at $s_0 = 0, \frac{1}{m}, \ldots, 1$. The only parameter that the user must specify is the number of pieces m of the initial piecewise linear manifold, defining the resolution of the method.

In the reparameterization step, given the i the block

$$F_i(x_0^{(i-1)}(s_{i-1}), \ldots, x_{i-1}^{(i-1)}(s_{i-1}), x_i) = 0 \qquad (i = 1, \ldots, N), \tag{16}$$

the goal is to construct the numerically well-conditioned, explicit parameterizations

$$x_k = x_k^{(i)}(s_i) \;\text{ for } k = 0, \ldots, i, \quad s_i \in [0,1]^{d_0}, \tag{17}$$

together with the transition map

$$s_{i-1} = s_{i-1}(s_i). \tag{18}$$

This **reparameterization is achieved by redistributing the breakpoints of the piecewise linear manifold so that the breakpoints are situated approximately uniformly**: infeasible and too close points are removed and new points are inserted between too distant points. (Points that satisfy $F_i(x) = 0$ but violate the variable bounds $\underline{x} \leq x \leq \overline{x}$ are infeasible.) This ensures that the piecewise linear approximation of the solution set is sufficiently accurate at each block; compare with Figure 4.

3 Comparison to the stage-by-stage or to the sequential modular approach

The stage-by-stage method [4] and the sequential modular approach to process flow sheeting and optimization [2, Chapter 8] will be referred to as **traditional methods** hereafter for short.

Basically all differences between the traditional methods and the proposed method are implied by one fundamental difference: The traditional methods involve computations with real numbers only, whereas the proposed method maintains an entire manifold representation at each block.

Since real numbers cannot be reparameterized, the traditional methods cannot involve any reparameterization. Given an initial guess for x_0, the estimates of all the remaining variables and the residuals of the final system of equations are determined in one forward pass only. Then the initial estimate is updated based on the residuals and all the variables are recomputed again in a forward pass. The forward pass is repeated until convergence.

The proposed method is guaranteed to reach convergence in just two passes: one forward sweep followed by one backward sweep. The traditional methods do not involve a backward sweep due to the

lack of reparameterization; they have to iterate on the tear variables instead. The convergence of the traditional methods greatly depends on the initial guess.

The traditional methods require an initial estimate for x_0 and can be extremely sensitive to it. This sensitivity is their most problematic feature. The proposed method does not require any initial estimate and is numerically stable thanks to the repeated reparameterizations.

This comparison is summarized in Table 1.

	traditional	proposed
operates on	real numbers	discretized manifolds
reparameterization	no	yes
backward sweep	no	yes
convergence	not guaranteed	guaranteed
initial guess	needed	none
sensitivity	yes	no

Table 1: Comparing the characteristic features of the traditional and the proposed method.

4 Appendix: ANSI C implementation of the illustrative linear example

Listing 4: Source code of the program used in the computations.

```
#include <math.h>
#include <stdio.h>

#define M 10.0
#define N_VARS 21
#define n (N_VARS-1)

void traditional_approach() {

        double x[N_VARS];
        double c[N_VARS];
        double residual;
        int i;

        c[0] = c[n] = (M+1);

        for (i=1; i<n; ++i) {
                c[i] = (1+M+1);
        }

        // Setup finished
        //---------------------------------------------------------------------



        x[0] = 1+1.0e-10;
        x[1] = -M*x[0] + c[0];

        for (i=2; i<=n;++i) {
                x[i] = -x[i-2]-M*x[i-1]+c[i-1];
        }

        // Final equation x_{n-1} + M*xn = cn
        residual = x[n-1] + M*x[n] - c[n];

        //---------------------------------------------------------------------


        // Done, printing x and the residual
        for (i=0; i<N_VARS; ++i) {
                printf("x%d = % .11g\n", i, x[i]);
        }

        printf("residual = %.3e\n", residual);
}

void traditional_sensitivity() {

        double x[N_VARS];
        double a[N_VARS] = { 0 };
        double b[N_VARS] = { 0 };
        double c[N_VARS];
        double residual;
        int i;

        c[0] = c[n] = (M+1);

        for (i=1; i<n; ++i) {
                c[i] = (1+M+1);
        }
        // Setup finished
        //---------------------------------------------------------------------
```

```
        // x0 = a0*x0 + b0 with a0=1, b0=0
        a[0] = 1;
        b[0] = 0;

        // M*x0 + x1 = c0
        // x1 = a1*x0+b1, where
        a[1] = -M;
        b[1] = c[0];

        // x0 + M*x1 + x2 = c1
        // (a0*x0+b0) + M(a1*x0+b1) + x2 = c1
        // x2 = a2*x0 + b2, where
        for (i=1; i<n; ++i) {
                a[i+1] = - M*a[i] - a[i-1];
                b[i+1] = c[i] - M*b[i] - b[i-1];
        }

        x[0] = 1;

        for (i=1; i<=n;++i) {
                x[i] = a[i]*x[0]+b[i];
        }

        // Final equation x_{n-1} + M*xn = cn
        residual = x[n-1] + M*x[n] - c[n];

        //-----------------------------------------------------------------


        // Done, printing x and the coefficients
        for (i=0; i<N_VARS; ++i) {
                printf("x%d = % .11g\n", i, x[i]);
        }

        printf("\n");

        for (i=0; i<N_VARS; ++i) {
                printf("a%d = % .16g\n", i, a[i]);
                printf("b%d = % .16g\n\n", i, b[i]);
        }

        printf("residual = %.3e\n", residual);
}

void proposed_method() {

        double x[N_VARS];
        double a[N_VARS] = { 0 };
        double b[N_VARS] = { 0 };
        double s[N_VARS];
        double c[N_VARS];
        double lambda[N_VARS];
        double mu[N_VARS];
        int i;

        c[0] = c[n] = (M+1);

        for (i=1; i<n; ++i) {
                c[i] = (1+M+1);
        }
        // Setup finished
        //-----------------------------------------------------------------


        // x0 = s0
        // x1 = s1
        // M*s0 + s1 = c0
        // s0 = a1*s1 + b1 where a1, b1
        a[1] = -1/M;
        b[1] = c[0]/M;
```

```
        // x2 = s2
        // (a1*s1+b1) + M*s1 + s2 = c1
        // s1 = a2*s2 + b2 where a2, b2

        // Forward sweep
        for (i=2; i<N_VARS; ++i) {
                a[i] = -1/(M+a[i-1]);
                b[i] = (c[i-1]-b[i-1])/(M+a[i-1]);
        }

        // Final equation
        // (an*sn+bn) + M*sn = cn, gives sn
        s[n] = (c[n]-b[n])/(M+a[n]);
        x[n] = s[n];

        // Backward sweep
        for (i=n; i>0; --i) {
                s[i-1] = a[i]*s[i] + b[i];
                x[i-1] = s[i-1];
        }

        //------------------------------------------------------------------


        // Done, printing solution
        for (i=0; i<N_VARS; ++i) {
                printf("x[%d] = %.14g\n", i, x[i]);
        }

        printf("\n");
        // Printing the coefficients of the transition maps
        for (i=0; i<N_VARS; ++i) {
                printf("a%d = % .15g\n", i, a[i]);
                printf("b%d = % .15g\n\n", i, b[i]);
        }

        //------------------------------------------------------------------


        // Sensitivity
        lambda[n] = 1;
        mu[n] = 0;

        for (i=n; i>=1; --i) {
                lambda[i-1] = a[i]*lambda[i];
                mu[i-1] = a[i]*mu[i] + b[i];
        }

        for (i=n; i>=0; --i) {
                printf("lambda%d = % .15e\t\tmu%d = % .16f\n", i, lambda[i], i, mu[i]);
        }
}

int main() {

        printf("-----------------------------------------------------\n");
        printf("Traditional approach\n");
        traditional_approach();
        printf("-----------------------------------------------------\n");
        printf("Sensitivity build up in the traditional approach\n");
        traditional_sensitivity();
        printf("-----------------------------------------------------\n");
        printf("Proposed method\n");
        proposed_method();

        return 0;
}
```

References

[1] A. Baharev and A. Neumaier. A globally convergent method for finding all steady-state solutions of distillation columns. submitted, 2013.

[2] L.T. Biegler, I.E. Grossmann, and A.W. Westerberg. *Systematic Methods of Chemical Process Design*. Prentice Hall PTR, Upper Saddle River, NJ, 1997.

[3] M.E. Henderson. Higher-dimensional continuation. In B. Krauskopf, H.M. Osinga, and J. Galán-Vioque, editors, *Numerical Continuation Methods for Dynamical Systems*, chapter 3, pp. 77–115. Dordrecht, The Netherlands: Springer, 2007.

[4] W.K. Lewis and G.L. Matheson. Studies in distillation. *Ind. Eng. Chem.*, 24:494–498, 1932.