Improving SIR
A Semi-Implicit Root Solver with Particular Applications to Global Solution of PDEs

Cristian Håkansson
Abstract

Improving SIR - A Semi-Implicit Root Solver with Particular Applications to Global Solution of PDEs

Cristian Håkansson

The computational core of the Generalized Weighted Residual Method (GWRM) is the Semi-implicit Root Solver (SIR), originally written in Maple.

In this paper, we improve the performance of the SIR algorithm. This improved version is then implemented in MATLAB. We incorporate this MATLAB version into GWRM and compare it to a similarly improved pure Maple implementation. Furthermore, Newton’s method with line search (NL) is also implemented in MATLAB and compared to SIR.

In order to perform this comparison we introduce convergence maps as a new, compact, way to visualize convergence properties as we vary the initial guess. These are employed in order to compare SIR and NL in terms of stability of convergence. The tests are carried out on a predefined set of standard problems.

We conclude that the MATLAB implementation of SIR is a substantial improvement over the Maple implementation when used in GWRM. Furthermore, compared to NL, SIR is shown to have a greatly reduced sensitivity to variations in the initial guess. Conversely, NL is shown to perform slightly better when using only the initial guesses defined along with the standard problems.

This paper ends with a few suggestions on how the algorithm can be further improved.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
</tr>
<tr>
<td>1.1</td>
<td>Introduction to GWRM</td>
</tr>
<tr>
<td>1.2</td>
<td>Introduction to the Semi-Implicit Root Solver</td>
</tr>
<tr>
<td>1.2.1</td>
<td>One-dimensional Algorithm</td>
</tr>
<tr>
<td>1.2.2</td>
<td>Generalization in Several Dimensions</td>
</tr>
<tr>
<td>2</td>
<td>Goals</td>
</tr>
<tr>
<td>2.1</td>
<td>Translating SIR into MATLAB</td>
</tr>
<tr>
<td>2.2</td>
<td>Improving Efficiency</td>
</tr>
<tr>
<td>2.3</td>
<td>Incorporating the MATLAB Translation into GWRM</td>
</tr>
<tr>
<td>2.4</td>
<td>Comparison to Line Search</td>
</tr>
<tr>
<td>2.4.1</td>
<td>Implementing Line Search in MATLAB</td>
</tr>
<tr>
<td>2.4.2</td>
<td>Introduction to Newton’s Method Using Line Search</td>
</tr>
<tr>
<td>2.4.3</td>
<td>Defining and Solving a Predefined Set of Problems</td>
</tr>
<tr>
<td>2.4.4</td>
<td>Measuring Performance</td>
</tr>
<tr>
<td>3</td>
<td>Realizing the Goals</td>
</tr>
<tr>
<td>3.1</td>
<td>Implementing SIR and NL in MATLAB</td>
</tr>
<tr>
<td>3.1.1</td>
<td>The Symbolic Toolbox</td>
</tr>
<tr>
<td>3.2</td>
<td>Efficiency Concerns when Using the Symbolic Toolbox</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Verifying the Implementation</td>
</tr>
<tr>
<td>3.3</td>
<td>Improving Efficiency of the Algorithm</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Utilizing Sparse Matrix Structures</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Decreasing the Number of Quasi-Monotonicity Checks</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Using Subiterations as a Last Resort</td>
</tr>
<tr>
<td>3.3.4</td>
<td>Removal of Mirroring</td>
</tr>
<tr>
<td>3.3.5</td>
<td>Making Subiterations Conditional</td>
</tr>
<tr>
<td>3.4</td>
<td>Incorporating SIR into GWRM</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Interfacing MATLAB and Maple Using the Maple Toolbox</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Modifying the Source Code</td>
</tr>
<tr>
<td>3.5</td>
<td>Comparison with Newton’s Method Using Line Search</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Measuring CPU Time and Number of Iterations</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Measuring Sensitivity to Variations in Initial Guess</td>
</tr>
<tr>
<td>4</td>
<td>Results</td>
</tr>
<tr>
<td>4.1</td>
<td>Translation of SIR into MATLAB, improved efficiency</td>
</tr>
<tr>
<td>4.2</td>
<td>Incorporation of SIR into GWRM</td>
</tr>
<tr>
<td>4.3</td>
<td>Comparison to NL</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Implementation of NL in MATLAB</td>
</tr>
<tr>
<td>4.3.2</td>
<td>CPU Time and Iterations Required</td>
</tr>
<tr>
<td>4.4</td>
<td>Sensitivity to Initial Guess</td>
</tr>
<tr>
<td>5</td>
<td>Discussion</td>
</tr>
<tr>
<td>5.1</td>
<td>Interpreting the Results</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Advantages</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Disadvantages</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Other Characteristics</td>
</tr>
<tr>
<td>5.2</td>
<td>Incorporating SIR into GWRM</td>
</tr>
</tbody>
</table>
5.3 Future Improvements ............................................. 20

A Complete List of Convergence Maps .................................. 22

B Source Code ......................................................... 25

B.1 Sparse Jacobian Definitions .................................... 25
B.2 MATLAB Implementation of SIR ................................ 25
  B.2.1 Pseudo Code for SIR ....................................... 26
  B.2.2 SIR.m .......................................................... 28
B.3 MATLAB/Maple Hybrid Implementation of GWRM .......... 32
  B.3.1 GWRM.m ...................................................... 32
  B.3.2 Burger.mpl .................................................. 32
  B.3.3 SIR.m .......................................................... 37
B.4 MATLAB Implementation of NL ................................ 40
  B.4.1 NL.m .......................................................... 41
  B.4.2 lnsrch.m ...................................................... 42
B.5 MATLAB/Maple Hybrid Implementation of SIR ............ 44
  B.5.1 runSIR.m ...................................................... 44
  B.5.2 prob_cos.mpl ............................................... 45
  B.5.3 jac.mpl ...................................................... 45
  B.5.4 jac_sp.mpl .................................................. 45
  B.5.5 SIR.m .......................................................... 46
B.6 Helper Functions .................................................. 49
  B.6.1 jac.m .......................................................... 49
  B.6.2 map2mat.m .................................................. 50
  B.6.3 map2sparse.m ............................................... 50
  B.6.4 addPar.m ...................................................... 52
1 Introduction

The overarching goal, of which this thesis is a part, is to improve the performance of the Generalized Weighted Residual Method (henceforth abbreviated GWRM) for solving partial differential equations. The computational core of this algorithm is the multidimensional non-linear root solver SIR (the Semi-Implicit Root solver). One of the most important ways of improving the performance of GWRM is thus to improve SIR.

The remainder of this introductory section will introduce GWRM and SIR. This brief introduction should cover enough to allow the reader to fully understand the remainder of the report. The interested reader will find references to more detailed descriptions.

The next section will detail the goals of this thesis, as well as outline how they are to be accomplished. Following this, the third section will detail how the aforementioned goals were realized. The fourth section will list the results achieved. Finally, in the fifth section, we will discuss these results.

1.1 Introduction to GWRM

What follows is a summary of the description given in [1], where more detailed information can also be found.

Consider a system of parabolic or hyperbolic initial-value partial differential equations:

\[
\frac{\partial u}{\partial t} = Du + f \tag{1}
\]

where \( u = u(t, x; p) \) is the solution vector, \( D \) is a linear or nonlinear matrix operator and \( f = f(t, x; p) \) is a forcing term assumed arbitrary but non-dependent on \( u \). Here, \( t \) denotes time, \( x \) spatial variables and \( p \) physical parameters. Initial as well as (Dirichlet, Neumann or Robin) boundary conditions are assumed known.

Now, eq. (1) is integrated in time. Note that, for simplicity, we restrict \( u \) to a single equation with only one physical parameter and spatial dimension. Schemes for several coupled equations and higher dimensions may subsequently be straightforwardly obtained.

\[
u(t, x; p) = u(t_0, x; p) + \int_{t_0}^{t} \{ Du(t', x; p) + f(t', x; p) \} \, dt' \tag{2}\]

The solution, \( u(t, x; p) \), is approximated by an expansion in Chebyshev polynomials, defined by \( T_n(x) = \cos(n \arccos x) \):

\[
u(t, x; p) = \sum_{k=0}^{K} \sum_{L=0}^{L} \sum_{M=0}^{M} a_{kLM} T_k(\tau)T_l(\xi)T_m(P) \tag{3}\]

where \( \tau, \xi \) and \( P \) are \( t, x \) and \( p \), respectively, normalized to have domains \([-1, 1]\).
This is required since the Chebyshev polynomials require arguments to be in that range.

Primes on summation signs indicate that each occurrence of a zero coefficient index should render a multiplicative factor of $1/2$.

Next, we determine the unknown coefficients $a_{k,lm}$ by using the Galerkin weighted residual method, given by the following equations:

$$
\int_{t_0}^{t_1} \int_{x_0}^{x_1} \int_{p_0}^{p_1} \left\{ u(t,x;p) - \left[ u(t_0,x;p) + \int_{t_0}^{t} \{ Du + f \} dt' \right] \right\} T_k(\tau)T_l(\xi)T_m(P) w_t w_x w_p dt dx dp = 0
$$

where $t_0$ and $t_1$ denote the lower and upper computational domain boundaries for $t$, and similarly for $x_0, x_1, p_0$ and $p_1$. $w_t, w_x$ and $w_p$ are weights given by:

$$w_t = (1 - \tau^2)^{-1/2}, \quad w_x = (1 - \xi^2)^{-1/2}, \quad w_p = (1 - P^2)^{-1/2}
$$

It can now be shown that this reduces to:

$$a_{q,rs} = 2\delta_{q0}b_{rs} + A_{q,rs} + F_{q,rs}
$$

where $\delta_{ik}$ is the Kronecker delta (being 1 if $i = k$ and 0 otherwise) and $A_{q,rs}, F_{q,rs}$ and $b_{rs}$ are the coefficients in the following expansions:

$$\int_{t_0}^{t} Du(t', x; p) dt' = \sum_{k=0}^{K} \sum_{l=0}^{L} \sum_{m=0}^{M} A_{k,lm} T_k(\tau) T_l(\xi) T_m(P)
$$

$$\int_{t_0}^{t} f(t', x; p) dt' = \sum_{k=0}^{K} \sum_{l=0}^{L} \sum_{m=0}^{M} F_{k,lm} T_k(\tau) T_l(\xi) T_m(P)
$$

$$u(t_0, x; p) = \sum_{l=0}^{L} \sum_{m=0}^{M} b_{lm} T_l(\xi) T_m(P)
$$

where

$$b_{lm} = \sum_{k=0}^{K} a_{k,lm} T_k(\tau_0) = \sum_{k=0}^{K} a_{k,lm} (-1)^k
$$

Eq. (5) is to be computed for all $0 \leq q \leq K + 1, 0 \leq r \leq L_{BC}, 0 \leq s \leq M$. Note that the number of modes in the temporal domain is extended to $M+1$ due to the time integration, and that the high end spatial modes with $r \geq L_{BC}$ are saved for implementation of boundary conditions.

Now, we solve eq. (7) for the coefficients $F_{k,lm}$. Likewise, solving eq. (6) for the coefficients $A_{k,lm}$ will yield these as functions of $a_{q,rs}$.
Finally, by inserting the coefficients \( a_{qrs} \) in the vector \( a \), eq. (5) can be written in vector notation as:

\[
a = \varphi(a)
\]

(10)

where \( \varphi \) is a vector valued function of several variables.

We will now focus our attention on the root solver used by GWRM to solve this implicit system of equations.

### 1.2 Introduction to the Semi-Implicit Root Solver

#### 1.2.1 One-dimensional Algorithm

We want to solve the equation \( f(x) = 0 \), or rearranged, \( x = x - f(x) = \varphi(x) \).

This can be done by the simple iteration scheme \( x_{i+1} = \varphi(x_i) \) given that the following conditions are met: Assume that the equation \( x = \varphi(x) \) has a root \( \chi \) and that \( \frac{d\varphi}{dx} < 1 \) in the interval \( I = \{ x : |x - \chi| \leq \rho \} \). It can then be shown that the scheme converges to \( \chi \) for each starting guess \( x^0 \in I \).

Now, in order to assure that the above conditions are met for any choice of \( \varphi \), we solve the problem \( x + \beta x = \varphi(x) + \beta x \), having the same roots as the initial equations.

If we apply the simple iteration scheme mentioned above, we arrive at \( x_{i+1} + \beta x_{i+1} = \varphi(x_i) + \beta x_i \). Substitute \( \beta = \frac{\alpha}{1 - \alpha} \) and rewrite as:

\[
x_{i+1} + \frac{\alpha x_{i+1}}{1 - \alpha} = \varphi(x_i) + \frac{\alpha x_i}{1 - \alpha} \\
(1 - \alpha)x_{i+1} + \alpha x_{i+1} = (1 - \alpha)\varphi(x_i) + \alpha x_i \\
x_{i+1} = \varphi(x_i) - \alpha \varphi(x_i) + \alpha x_i = \varphi(x_i) - \alpha (\varphi(x_i) + x_i) \\
x_{i+1} = \alpha (x_i - \varphi(x_i)) + \varphi(x_i)
\]

(11)

Now, define \( \Phi(x, \alpha) = \alpha (x - \varphi(x)) + \varphi(x) \). Setting \( \Phi' = R \), where the \( \cdot' \) denotes the derivative with respect to \( x \), and solving for \( \alpha \) yields:

\[
\Phi' = \alpha (1 - \varphi' (x)) + \varphi' (x) = R \\
\alpha = \frac{R - \varphi' (x)}{1 - \varphi' (x)}
\]

(12)

We can now use this formula to assure that \( R < 1 \) at each iteration. Note that \( R = 1 \Rightarrow x_{i+1} = x_i \). Furthermore, it can be proven [2] that by setting \( R = 0 \), we obtain Newton’s method. This is a desirable property because of the second order convergence of Newton’s method.

One drawback of Newton’s method is that it will only converge if the initial guess is sufficiently close to a root. The semi-implicit root solver (henceforth
abbreviated SIR) overcomes this by initially setting \( R \) close to 1, decreasing it as the algorithm moves closer to a root. Thus SIR will tend towards the behaviour of Newton’s method as the circumstances grow increasingly favourable for that algorithm.

Furthermore, SIR guarantees monotonic convergence by determining the direction of the root though root bracketing (see algorithm in [3]) and applying the following safeguards:

1. If the solution is moving away from the root, \( \Phi \) is mirrored in the line \( f(x) = x \), thus assuring it moves in the other direction. This is achieved by replacing \( \Phi(x, \alpha) \) with \( \Phi(x, 2 - \alpha) \).

2. If \( \Phi \) has changed sign somewhere between \( x^i \) and \( x^{i+1} \), the algorithm will have passed a root. \( R \) is then increased towards 1, using subiterations, until the step size is small enough to avoid this.

3. Singular, or near-singular, values of \( \alpha \) are avoided using subiterations when \( |\alpha| \) exceeds a critical value \( \alpha_c \). This will again increase \( R \) towards 1, thus reducing \( \alpha \) towards 1.

### 1.2.2 Generalization in Several Dimensions

In more than one dimension, we will instead solve a system of \( N \) equations. This can be written in vector notation as follows: find \( x \) such that \( \varphi(x) = 0 \), where \( x = (x_1, x_2, \ldots, x_N)^T \) and \( \varphi(x) = (\varphi_1(x), \varphi_2(x), \ldots, \varphi_N(x))^T \) where \( T \) denotes the transpose.

In analogy to the one-dimensional case, the problem is replaced by \( x = A(x - \varphi(x)) + \varphi(x) \) which gives us the iterative scheme \( x^{i+1} = A(x^i - \varphi(x^i)) + \varphi(x^i) \). Here, \( A \) is an \( N \times N \) matrix, corresponding to \( \alpha \) in the one-dimensional case.

The parameter matrix \( A \) is derived much like in the previous section: define \( \Phi(x, A) = A(x - \varphi(x)) + \varphi(x) \). Set \( \Phi' = R \) and let \( I \) be the identity matrix:

\[
\Phi' = A(I - J_\varphi) + J_\varphi = R
\]

Rewriting \( R = (A - I)(I - J_\varphi) + I \) and solving for \( A \) yields:

\[
A = I + (R - I)(I - J_\varphi)^{-1} = I + (R - I)J_f^{-1}
\]

where \( J_f \) is the Jacobian of \( f(x) = x - \varphi(x) \). Note that \( \varphi(x) = x \Leftrightarrow f(x) = 0 \). Thus, \( f \) is the function that Newton’s method would work upon in order to solve \( \varphi(x) = x \).

Like in the one-dimensional case, it can be shown that SIR is equivalent to Newton’s method when \( R = 0 \). Also, setting \( R = I \Rightarrow x^{i+1} = x^i \). Thus, it is still possible to initially set \( R \) close to the identity matrix, letting it approach the zero matrix.
Furthermore, $\mathbf{R}$ will be kept diagonal throughout the process, i.e. the derivative of $f_i(x)$ will be set to zero, except in the direction of $x_i$. The reasoning behind this is to construct $\Phi_i$ in such a way that it will have the same roots as $\varphi_i$, while remaining locally constant in all dimensions except the $i$:th.

However, monotonicity cannot be guaranteed for systems of equations, since it is no longer possible to determine the directions of roots. A way to mimic the behaviour of the one-dimensional algorithm is to replace the known direction towards the root with the direction of the Newton step: $-\mathbf{J}_f^{-1} f$. Since it can be proven [3] that the Newton step is a descent direction of $|f(x)|^2$, SIR will not move away from a nearby global minimum if one exists.

We may now summarize the SIR algorithm:

\[
\begin{align*}
x^{i+1} &= \mathbf{A} (x^i - \varphi(x^i)) + \varphi(x^i) = \Phi(x^i) \\
\mathbf{A} &= \mathbf{I} + (\mathbf{R} - \mathbf{I})\mathbf{J}_f^{-1} \\
\frac{\partial \Phi_i}{\partial x_j} &= \mathbf{R}_{ij} = \begin{cases} 
\mathbf{R}_{ii} & \text{if } i = j \\
0 & \text{if } i \neq j
\end{cases}
\end{align*}
\]

The values of $\mathbf{R}_{ii}$ are initialized close to 1 and decreased at the end of each iteration.

In this modified state, the safeguards from the previous section can be written as follows:

1. If the solution is moving in a direction that is at odds with the Newton step (as defined by their scalar product being negative), $\Phi$ is mirrored so that the step is taken in the opposite direction. Here, this is accomplished by replacing $\mathbf{A}_{ij}$ with $2 - \mathbf{A}_{ij}$ if $i = j$ and with $-\mathbf{A}_{ij}$ otherwise.

2. If $\Phi_i$ has changed sign in some dimension $i$, then the algorithm has passed a local root of $\Phi_i(x) = 0$, which means it may also have passed the global root of $\Phi(x) = 0$. $\mathbf{R}$ is then increased, by subiterating, towards $\mathbf{I}$ until the step size is small enough to avoid this.

3. If the largest $|\mathbf{A}_{ij}|$ exceeds a critical value $\mathbf{A}_c$, the algorithm will again increase $\mathbf{R}$ towards $\mathbf{I}$ until this is no longer the case.

For a description of SIR in greater depth, along with Maple source code, the reader is referred to [2].

2 Goals

This thesis work aims to achieve the following goals:

**Translate SIR into MATLAB.** By doing so we hope to improve the efficiency as well as make the algorithm more accessible to the scientific computing community.

**Improve efficiency** of SIR.
Incorporate the MATLAB Translation into GWRM in order to improve performance. We also want to accomplish this in a way that allows us to reuse as much of the original Maple GWRM code as possible.

Compare to line search. SIR attempts to strike a balance between speed and stability of convergence. We will compare it in both of these regards to Newton’s method using line search, one of the most widely used algorithms for solving systems of nonlinear equations.

The remainder of this chapter will detail, for each of these goals, how they are to be realized.

2.1 Translating SIR into MATLAB

There are two major considerations to be made when translating from Maple to MATLAB. The first is to identify the relevant differences between the languages (we know that the two languages differ at least in their handling of symbolic variables). The second is to accommodate for MATLABs own optimization scheme, which poses certain requirements on data structures and coding conventions if execution speed is to be maximized.

2.2 Improving Efficiency

There are two basic levels where efficiency may be improved. The first is in the MATLAB implementation, while the second is in the formulation of the algorithm.

In either case, changes will be implemented and performance measured and compared on a sample of problems described in section 2.4.3. The profiling tool built into MATLAB will be a useful resource in deciding where optimization is needed the most.

2.3 Incorporating the MATLAB Translation into GWRM

Since we want to reuse as much of the original Maple code as possible, only SIR should be defined in MATLAB. Outside of SIR, we want to make as small changes as possible in the Maple code to facilitate this incorporation.

Interfacing the MATLAB and Maple engines will result in some amount of computational overhead. We must keep this small enough so as to not overshadow any performance gained from the MATLAB implementation.

2.4 Comparison to Line Search

This part is broken down further into the following substeps:

2.4.1 Implementing Line Search in MATLAB

For the comparison to be valid, it is reasonable to expect the algorithms to be implemented and run in the same programming environment. For instance, a library implementation of line search would likely contain low level optimizations far beyond the scope of this thesis.

Thus, a standard implementation of line search described in [3] will be translated into MATLAB in much the same way as SIR.
2.4.2 Introduction to Newton’s Method Using Line Search

This section contains a brief description of Newton’s method using line search (henceforth abbreviated NL). For a more thorough description, the reader is directed to [3].

First of all, recall the formulation of Newton’s method in several dimensions:

\[ x^{i+1} = x^i - J^{-1}(x^i) f(x^i) \]  \hspace{1cm} (15)

The term added at each iteration, \( -J^{-1}(x^i) f(x^i) \), is sometimes referred to as the \textit{Newton step} and the corresponding direction the \textit{Newton direction}.

NL expands upon this premise by allowing the algorithm to take only fractions of the Newton step. Thus, NL can be formulated as follows:

\[ x^{i+1} = x^i - \alpha J^{-1}(x^i) f(x^i) \]  \hspace{1cm} (16)

In order to find a suitable value of \( \alpha \), we observe that \( \| f(x) \| = 0 \) exactly when \( f(x) = 0 \). The problem can thus be reduced to the one-dimensional problem of finding the minimum of the scalar-valued function \( \| f(x) \|^2 = f^2 = f \cdot f \) along the line defined by the Newton direction.

The implementation described in [3] accomplishes this by approximating \( \| f(x) \|^2 / 2 \) (The division by two merely serves to simplify computations) with a polynomial \( p(x) \). The polynomial is then derived in order to find potential minima. This process is repeated, each time using the newest points for the approximation, until a suitable \( \alpha \) is found.

As mentioned in section 1.2.2, it can be proven [3] that the Newton direction is a descent direction of \( \| f(x) \|^2 \). This means that once the algorithm comes close enough to a local minimum of \( \| f(x) \|^2 \), it will converge towards it rather than towards the solution to \( f(x) = 0 \).

2.4.3 Defining and Solving a Predefined Set of Problems

In order to compare the algorithms, we will solve a predefined set of problems, namely the ones used by Lukšan in [4] and also used by others in performance tests. However, some of the problems listed will cause the Jacobian matrix to become singular, preventing them from being solved by NL and SIR. We will thus restrict our investigation to the problems where this does not occur.

Each problem will be defined with both 20 and 100 variables. We will also solve \( x = \varphi(x) \) where

\[ \varphi_i = \begin{cases} \cos(x_{i+1}) & \text{if } i < N; \\ 3 \cos(x_i) & \text{if } i = N. \end{cases} \]  \hspace{1cm} (17)

We will create MATLAB functions corresponding to each of these problems. These functions will generate the relevant equations in a format that matches the input parameters of SIR and NL.

2.4.4 Measuring Performance

We will perform two separate sets of tests in order to compare the performance of the algorithms. Both are based on solving the systems of equations in the previous subsection.
In the first test, CPU time and number of subiterations will be measured when the equations are solved using the initial guess given in the reports from which they are taken. We will allow at most 100 iterations for the solution of each problem and if NL converges to a local minimum this will be noted.

In the second test, we will vary the initial guess and measure the number of iterations required to solve the problems. Again, we will allow at most 100 iterations and note if NL converges to a local minimum.

The first of these tests will determine the overall efficiency of the algorithms, while the second will determine their sensitivity to variations in the initial guess.

3 Realizing the Goals

3.1 Implementing SIR and NL in MATLAB

One thing that should be noted when writing in MATLAB is that it is optimized for the handling of vectors and matrices. One way in which this manifests itself is that many operations will become more efficient if one can write them as matrix or vector operations.

For instance, the following for loop:

```matlab
for i = 1:N
    x[i] = x[i]+5;
end
```

Can be more efficiently written as:

```matlab
x = x+5
```

This often improves not only performance, but also clarity.

3.1.1 The Symbolic Toolbox

One major difference between Maple and MATLAB is that the latter does not natively handle symbolic operations. This means, among other things, that while it can approximate the Jacobian at any given point, it cannot natively derive it analytically and generate a formula. This is a notable drawback, as both SIR and NL rely on Jacobian matrices.

It should be noted that MATLAB does handle symbolic variables via its symbolic toolbox, which can be used to analytically compute Jacobians, among other things. Depending on MATLAB version, this toolbox utilized either the Maple or the MuPad engine.

Unfortunately, using this toolbox turns out not to be feasible in practice, for reasons detailed below.

3.2 Efficiency Concerns when Using the Symbolic Toolbox

As mentioned in section 2.2, MATLAB has a built-in profiling tool, allowing us to study in detail how much time is spent executing each line of code.

Using this profiling tool we find that, for many problems, SIR spends more than 95% of its execution time at the function `subs`, which substitute numeric values into symbolic variables. The function `subs`, in turn, spends about half
its execution time converting symbolic expressions into character strings which can be passed as arguments to Maple.

Furthermore, the jacobian function spends about half its execution time converting the resulting Maple matrix into a symbolic matrix in MATLAB.

Thus, using the symbolic toolbox will lead to a significant amount of computational overhead, to the point that the overhead often amount to over 90% of the execution time!

The solution that was chosen for this problem was to define $\varphi$ and its Jacobian as character strings, describing functions depending on $x$. By assigning $x = p$ in the MATLAB environment and using the command eval, the function value at the point $p$ can be computed.

For instance: assume we wish to evaluate $\varphi(x) = \begin{bmatrix} \cos(x_2) \\ \cos(x_3) \\ 3 \cdot \cos(x_1) \end{bmatrix}$ at the point $x = \begin{bmatrix} -2 \\ -2 \\ -2 \end{bmatrix}$.

This is accomplished with the following MATLAB code:

```matlab
phi = ' [cos(x(2));cos(x(3));3*cos(x(1))]';
x = [-2;-2;-2];
phi_x = eval(phi);
```

The Jacobian will be derived from $\varphi$ by issuing a direct call to the Maple core contained in the symbolic toolbox. All other dependencies upon Maple or the symbolic toolbox is removed.

It should be noted that this solution is implemented in MATLAB 6.1. It will not work in recent versions of MATLAB, where the Maple core has been replaced with MuPad and may or may not work in intermediate versions.

A version that is more likely to work in current and future versions of MATLAB is based on the Maple toolbox (see section 3.4.1). Its source code can be found in appendix B.5.

### 3.2.1 Verifying the Implementation

In order to assure the correctness of the implementations, any roots found were tested against the condition $f(x) = x$.

Furthermore, the MATLAB and Maple implementations were compared. Due to differences in floating point roundoff, the convergence criterion that two successive solutions should differ by only a small amount, may fail to hold after the exact same number of iterations. Thus, a small variation in the number of iterations required to solve a particular problem is to be expected. The measured variations were deemed to be within reasonable margin.
3.3 Improving Efficiency of the Algorithm

The efficiency of the algorithm (as opposed to the efficiency of the MATLAB implementation, as detailed above) was improved in several ways, detailed below:

3.3.1 Utilizing Sparse Matrix Structures

Sparsity is often utilized to improve efficiency when inverting matrices. This is because matrix inversion has a time complexity of $N^3$ for a problem of size $N$, which often dominates the theoretical time complexity of the algorithm.

However, in practical applications of SIR during the work on this thesis, the time spent on Jacobian inversions was negligible. Since the Jacobian matrices often need to be transformed into a more suitable form before the sparsity can be exploited, this appears to be a wasted effort.

Conversely, Jacobian evaluations often dominate the execution time, and sparsity of the Jacobian matrix can be exploited in order to make these more efficient, since only the non-zero elements need to be considered.

The same considerations are relevant in the computations of the $A$ matrix, which requires multiplication with the diagonal matrix $R - I$ (see section 1.2.2).

Since sparse definitions provide computational overhead, they are undesirable for small or dense problems. It may also require more pre-processing to generate a sparse Jacobian than a dense one.

To accommodate for this, SIR was implemented as callable with either dense or sparse definitions. However, the definition of $R - I$ was set to sparse at all times, even though it slightly decreases performance for small problems. The rationale for this decision is that these small problems take relatively short time to solve anyway.

3.3.2 Decreasing the Number of Quasi-Monotonicity Checks

Subiterating is a costly process: in order to maintain quasi-monotonicity, $\varphi$ must be evaluated at a predetermined number of points between the current guess and the next.

The default value of additional points, aside from the already known value at the current guess, is 10. For this value, the additional function evaluations often dominate the execution time.

In the one-dimensional implementation of SIR, these evaluations are crucial for maintaining monotonicity. However, in the general case, monotonicity cannot be guaranteed by any means, making these computations less critical.

It should be noted, however, that for a global minimum to exist, it must also be a local minimum in each dimension. Thus it is still worthwhile to assure that we do not overshoot these local minima as we encounter them.

In summary, it is desirable to reduce the amount of additional evaluations, while we do not wish to eliminate them altogether. Experiments indicate that lowering the number of evaluations to a single additional point (for a total of two points) has little overall impact on convergence.

This also allows for implementation-specific optimizations since the value of the governing parameter can now be hard coded. This can also potentially improve the clarity of the code.
3.3.3 Using Subiterations as a Last Resort

Even with the above improvement, subiterations remain costly and, more importantly, often unnecessary in practice. Indeed, they may even prove detrimental to convergence.

For instance, when solving the problems listed in section 2.4.3, the algorithm always required more iterations when subiterations were used, sometimes preventing convergence within the allotted 100 iterations altogether. For other problems such as eq. 17, subiterations are crucial to finding a solution.

A good strategy, then, would be to attempt to find a solution without subiterating, resorting to subiterations if none is found. To accommodate this usage, the algorithm will not use subiterations unless the user specifies this by passing an additional input argument.

3.3.4 Removal of Mirroring

Another crucial part of maintaining monotonicity in the one-dimensional case is to mirror the iterating function if the solution is moving away from the root. In the general case, however, we cannot determine the directions of the root. We therefore resort to using the Newton step as a guideline, since it can be proven to be a descent direction [3]. Thus, much like was the case with the monotonicity checks discussed in section 3.3.2, this is less crucial in the general case.

Experiments indicate that, by removing the mirroring from the algorithm, convergence is made slightly more dependent on the value of the initial guess. However, the performance and clarity of code gained was decided to outweigh the negatives. Thus, mirroring is no longer part of the general algorithm.

3.3.5 Making Subiterations Conditional

Often when the algorithm fails to converge, it is because the solution starts to rapidly shoot off in some direction. This suggests that subiterations are needed the most when the step length is increasing faster than at a certain rate. This procedure also safeguards against near-singular values of the elements of $A$.

Mathematically, we subiterate when $\|x^{(i)} - x^{(i-1)}\| < c \cdot \|x^{(i+1)} - x^{(i)}\|$, where $x^{(i)}$ is the estimated root after the $i$:th iteration.

Experiments indicate that $c = 1$ is a suitable value, yielding fast and stable convergence.

3.4 Incorporating SIR into GWRM

The incorporation of SIR into GWRM can be broken down into the following steps:

3.4.1 Interfacing MATLAB and Maple Using the Maple Toolbox

Not to be confused with MATLAB’s own symbolic toolbox, the Maple toolbox, developed by Maplesoft, is available to Maple users. It requires an installation of Maple to be present on the computer, which is why it is not feasible in the pure MATLAB implementation.
Unlike the symbolic toolbox, the Maple toolbox utilizes data structures that can store symbolic variables in such a way that they are readable from memory by both Maple and MATLAB. Hence, the overhead caused by string manipulations mentioned in section 3.2 is removed.

It is now possible to define variables in the Maple environment, pass them to SIR in the MATLAB environment and make the results available in the Maple environment.

### 3.4.2 Modifying the Source Code

The data structures that are readable in both programs, however, are not the ones initially used in the Maple implementation of GWRM. Thus, minor changes must be made to it.

Variables defined using the aforementioned structures become available in the MATLAB environment by defining (MATLAB) symbolic variables with the same name there.

Furthermore, the MATLAB implementation of SIR must be modified in two ways. The first is to replace the strings with symbolic variables. Secondly, in order to avoid working with very large Jacobian matrices, we partition the spatial dimension of the problem into subdomains and iterate over these separately using an inner for loop. These essentially add another dimension to all vectors and matrices. MATLAB has a three-dimensional matrix data structure, and matrices allow the engine to optimize performance. Thus, it makes sense to use three-dimensional matrices for the Jacobian and $A$ matrix and a two-dimensional matrix for the function $\varphi$.

In the original Maple implementation, the result was returned as a vector whose elements were the approximations of $x$ after each iteration. Here, it will be replaced by a cell array consisting of $M$ cells. A cell array is essentially an array which can have arbitrary (and dissimilar) data types as their elements. Here, the m:th cell will contain a matrix which in turn has columns containing the solution after each iteration step.

The Maple toolbox allows Maple code to be executed from within MATLAB, but not vice versa. Thus, the following routine will be followed:

1. Execute Maple code defining the problem.
2. Define symbolic variables in MATLAB.
3. Call SIR, passing symbolic variables as arguments.

### 3.5 Comparison with Newton’s Method Using Line Search

Since the MATLAB implementation of SIR accepts function definitions as character strings, this is the way we must define the problems to be solved.

MATLAB functions were created corresponding to each of the problems described in section 2.4.3, with input parameters allowing the user to specify the problem size. The user is also able to decide whether the problem should be defined as $\varphi(x) = 0$, to be solved using NL, or as $\varphi(x) + x = x$ to be solved using SIR. The function then returns the appropriate $\varphi$ as a character string.
3.5.1 Measuring CPU Time and Number of Iterations

The problems mentioned above were then solved in both 20 and 100 dimensions, using both SIR and NL.

It should be noted that this was done while running Microsoft Windows, so a small amount of background computation could not be avoided. However, to reduce the impact of the operating system running in the background, the problems were solved 20 times, and the lowest execution time was tabulated.

In order to verify the above method, the test was repeated for some problems and results proved consistent even if they were gathered at different times or dates.

Apart from execution time, the required number of iterations was also tabulated.

3.5.2 Measuring Sensitivity to Variations in Initial Guess

In this work, a new scheme for visualizing sensitivity to the initial guess has been developed.

For each problem, a grid of $61 \times 61$ initial guesses was constructed. A surface plot defined by the function $|\phi|^2/2$ was added to help identify behaviour near local minima, since this is the function minimized by NL.

At each of the $61 \times 61$ grid points, the convergence characteristics of the algorithm measured was represented with a color:

- **Red**: no convergence within the allotted 100 iterations.
- **Orange**: convergence to a local minimum (only relevant for NL).
- **Cyan**: slow convergence.
- **Blue**: fast convergence.

Note that red and orange are discrete values, while convergence is denoted by a color somewhere between cyan and blue depending on speed.

This scheme has the benefit of allowing both convergence characteristics and the position of local minima to be accessible at a glance. However, in order for this section to be possible, the problems must be solved in only two dimensions. Since not all problems were defined for two dimensions, this part was restricted to those that were. A possible way to extend the idea to problems of larger dimensionality is discussed in section 5.3.

4 Results

4.1 Translation of SIR into MATLAB, improved efficiency.

Pseudo code for the improved version of the SIR algorithm, as well as MATLAB source code is found in appendix B.2. Source code for the helper functions used can be found in appendix B.6.
4.2 Incorporation of SIR into GWRM.

The Source code is found in appendix B.3, which contains complete listings of all files used to solve Burgers’ equations using the Maple/MATLAB hybrid implementation.

Timing both the pure Maple and Maple/MATLAB versions of the code yielded the following execution times (rounded to whole seconds):

- Pure Maple: 50 sec.
- MATLAB/Maple: 27 sec.

Note that the computer used is not the same as the one used when comparing SIR to NL, so comparisons between the two sets of measurements will not be valid.

4.3 Comparison to NL.

4.3.1 Implementation of NL in MATLAB.

Source code of the files specific to the MATLAB implementation of NL is found in appendix B.4, while source code of any helper functions used can be located in appendix B.6

4.3.2 CPU Time and Iterations Required.

Table 1 lists number of iterations required and time taken when solving the problems mentioned in section 2.4.3. This is done four times, once for each combination of problem size \((N = 20 \text{ or } N = 100)\) and algorithm used (NL or SIR).

In order to facilitate comparison between the four columns, sums and standard deviations are computed for both time and number of iterations.

Note that, as mentioned in section 2.4.3, problems 5, 10 and 13 are excluded, as MATLAB reported singular or near-singular Jacobian matrices for these problems. This affected both SIR and NL.

4.4 Sensitivity to Initial Guess.

A selection of convergence maps can be found in figures 1-4, in order to view convergence maps for all problems, the reader is referred to appendix A.

Figure 5 displays the path between successive iterations of SIR and NL near a local minimum. A "water level", parallel to and slightly above the xy-plane, has been added to help identify the global minimum.

5 Discussion

5.1 Interpreting the Results

5.1.1 Advantages

Figures 1-5 illustrate the two primary advantages that SIR has over NL:
\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c}
\text{prob.} & \text{NL} & \text{iter.} & \text{time} & \text{SIR} & \text{iter.} & \text{time} & \text{prob.} & \text{NL} & \text{iter.} & \text{time} & \text{SIR} & \text{iter.} & \text{time} \\
\hline
1 & 7 & 0.02 & 1 & 10 & 0.03 & 1 & 9 & 0.22 & 1 & 12 & 0.25 \\
2 & 51 & 0.14 & 2 & 15 & 0.03 & 2 & 51 & 0.83 & 2 & 15 & 0.23 \\
3 & 4 & 0.02 & 3 & 8 & 0.05 & 3 & 3 & 0.16 & 3 & 8 & 0.27 \\
4 & 7 & 0.08 & 4 & 9 & 0.09 & 4 & 7 & 0.55 & 4 & 9 & 0.61 \\
6 & 16 & 0.11 & 6 & 28 & 0.20 & 6 & 16 & 0.67 & 6 & 28 & 1.14 \\
7 & 55 & 0.27 & 7 & 16 & 0.06 & 7 & 24 & 0.67 & 7 & 16 & 0.38 \\
8 & 23 & 0.17 & 8 & 17 & 0.09 & 8 & 16 & 0.78 & 8 & 17 & 0.58 \\
9 & 19 & 0.17 & 9 & 26 & 0.19 & 9 & 17 & 0.98 & 9 & 26 & 1.13 \\
11 & 14 & 0.03 & 11 & 10 & 0.02 & 11 & 14 & 0.17 & 11 & 10 & 0.13 \\
12 & 16 & 0.05 & 12 & 29 & 0.08 & 12 & 16 & 0.27 & 12 & 29 & 0.47 \\
14 & 4 & 0.02 & 14 & 9 & 0.03 & 14 & 4 & 0.11 & 14 & 9 & 0.17 \\
15 & 5 & 0.05 & 15 & 9 & 0.08 & 15 & 5 & 0.33 & 15 & 9 & 0.47 \\
16 & 2 & 0.02 & 16 & 9 & 0.03 & 16 & 2 & 0.09 & 16 & 9 & 0.20 \\
17 & 4 & 0.02 & 17 & 9 & 0.03 & 17 & 4 & 0.11 & 17 & 9 & 0.19 \\
\hline
\Sigma & 227 & 1.17 & \Sigma & 204 & 1.01 & \Sigma & 188 & 5.94 & \Sigma & 206 & 6.22 \\
\text{stdev} & 16.9 & 0.08 & \text{stdev} & 7.7 & 0.06 & \text{stdev} & 13 & 0.31 & \text{stdev} & 7.6 & 0.33 \\
\end{array}
\]

Table 1: CPU time and iterations required for the solution of various problems.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure1.png}
\caption{Convergence maps for eq. 17 solved using NL, SIR and SIR with subiterations, respectively.}
\end{figure}
Figure 2: Convergence maps for problem 7 solved using NL and SIR, respectively.

Figure 3: Convergence maps for problem 14 solved using NL and SIR, respectively.

Figure 4: Convergence maps for problem 17 solved using NL, SIR and SIR with subiterations, respectively.
Reduced dependency on initial guess. From the convergence maps it is clear that SIR allows for convergence for a wider range of initial guesses for the problems attempted.

Some problems (4,7 and 14) converge nearly unconditionally in the illustrated region when solved with SIR, whereas this is not the case for NL. Other problems (equation 17 and problem 17) have similar characteristics when solved with NL and SIR without subiterations, but converge at almost every point in the region when subiterations are used.

Problems that converge unconditionally in the region when solved with NL (11,15), do so also when solved with SIR. Only two of the problems tested, namely problems 2 and 6, retain their high sensitivity to the initial guess for all three algorithms (NL, SIR and SIR using subiterations). Problem 2 also illustrates how subiterations can sometimes cause SIR to perform considerably worse.

Thus, judging from the problems tested, SIR performs at least as well as, and often much better than, NL in this respect, especially if subiterations are used.

Cannot get stuck in local minima. As noted in section 2.4.2, NL converges to a minimum of $f_{\text{min}} = |f(x)|^2$, even if this is a local minimum. In contrast, SIR, like the original Newton’s method, doesn’t rely on $f_{\text{min}}$ for convergence, and thus cannot be fooled by its local minima. As mentioned previously, this behaviour is illustrated in fig 5.
5.1.2 Disadvantages

Judging again from table 1, SIR is faster than NL in terms of total execution time when \( N = 20 \), but somewhat slower when \( N = 100 \). Thus, SIR appears to scale less favorably with problem size.

It should be noted that this, more so than the advantages stated in the previous subsection, is implementation dependent and that the version of NL is one that has likely seen a fair amount of tweaking. There is therefore a potential for future implementations of SIR to perform better in this respect.

5.1.3 Other Characteristics

A minor observation that can be made is that the standard deviation of number of iterations in table 1 is lower for SIR than for NL. Thus SIR appears to tend less towards extreme values of number of iterations.

5.2 Incorporating SIR into GWRM

Although the timing performed was far from exhaustive, it certainly appears as though the attempt to improve performance by calling a MATLAB implementation of SIR was successful. The time taken by the Maple/MATLAB hybrid implementation was roughly half of that taken by the pure Maple implementation.

5.3 Future Improvements

This section covers two ideas that have come up during this thesis work that are deemed to have potential if developed further.

Approximate Jacobian matrices. The most time consuming step is often the evaluation of the Jacobian matrix. We have experimented with two methods to allow SIR to work with approximate Jacobians at certain stages.

The first idea is to not update the Jacobian at each iteration step, but instead keep working with an old one. It has been hypothesized that having an accurate value of the Jacobian is the most useful when we approach the actual solution, as well as during a few iterations during the upstart.

This hypothesis is supported by the fact that it is possible to remove Jacobian updates from a certain number of "middle" iterations in cases where the number required was known, and still solve the problem in the same number of iterations. So far, there is no test to determine when updates are necessary, so this cannot at present be applied to problems which have not already been solved.

We have also attempted to stop updates of the \( A \) matrix, of which the Jacobian computation is a part, once its relative change between two successive iterations is below a certain threshold. So far, no definite conclusions have been reached.
Convergence maps for N-dimensional problems. One possible way to extend the concept of convergence maps to an arbitrary number of equations is to choose two dimensions, perhaps randomly, to map.

Acknowledgement

I would like to thank the people who helped me complete this thesis:

Jan Scheffel, my supervisor, for his encouragement, ideas and patient explanations.

Kjell Pernestål, for helping me shape this report out of the chaos that came before it.

Malin Drevin, for emotional and financial support.

References


Appendices

A Complete List of Convergence Maps

All maps are ordered, from left to right, by the algorithm used: NL, SIR and SIR with subiterations if used.

Eq. 17:

Problem 2:

Problem 4:
B  Source Code

This section will list the source code used to run the implementations of SIR, NL and GWRM generated in this thesis. Helper functions used will be listed last, but their intended purpose will be made clear in the sections where they are used.

B.1 Sparse Jacobian Definitions

Both SIR and NL allow the user to define the Jacobian as either dense or sparse. Since SIR and NL require functions to be represented as strings, it is not possible to simply use the built-in MATLAB command \texttt{sparse}(J) to convert J into a sparse representation.

However, another built-in representation can be used in a slightly modified fashion: MATLAB allows us to represent a matrix, A, with the three vectors r, c and B. Here, B contains the non-zero elements of A while r and c contain their row and column indices, respectively. Thus the following holds:

\[ A(r(i),c(i)) = B(i); \]

If we replace B with a string, we can allow it to contain functions which can be passed to SIR or NL. For instance, the following, dense, Jacobian:

\[ J = \begin{bmatrix} 0 & -\sin(x_2) & 0; 0 & 0 & -\sin(x_3); -3\sin(x_1) & 0 & 0 \end{bmatrix}; \]

can be represented in the following, sparse, fashion:

\[ r = [1,2,3]; \]
\[ c = [2,3,1]; \]
\[ J = \begin{bmatrix} -\sin(x_2), -\sin(x_3), -3\sin(x_1) \end{bmatrix}; \]

B.2 MATLAB Implementation of SIR

The pure MATLAB implementation of SIR is intended to be used in a fashion similar to the following:
% Define function
f = ' [cos(x2);cos(x3);3*cos(x1)]';
% Define initial guess
x0 = [-2;-2;-2];
% Compute Jacobian
J = jac(f);
% Find root, no subiterations
U = SIR(f,J,x0);

Here, \textit{jac} is a helper function designed to compute the Jacobian of a function defined as a string, it is listed in appendix B.6.1.

Subiterations are activated by passing an additional boolean \texttt{true} when calling \texttt{SIR}:

% Find root using subiterations
U = SIR(f,J,x0,true);

Furthermore, the \textit{jac} function returns a sparse representation if called like this:

% Compute sparse Jacobian
[r,c,J] = jac(f);

This can then be passed to \texttt{SIR} by putting the \texttt{r}, \texttt{c} and \texttt{J} in a cell array, like this:

% Find root, no subiterations, sparse Jacobian
U = SIR(f,{r,c,J},x0);

\textbf{B.2.1 Pseudo Code for SIR}

\textbf{Input:}

$\varphi$: A representation of the function $\varphi : \mathbb{R}^N \rightarrow \mathbb{R}^N$.

$J$: A representation of the Jacobian $J_\varphi$.

$x^0$: An initial estimate for the root of $x = \varphi(x)$.

$\text{sub}$: A boolean variable indicating whether or not subiterations are to be used.

\textbf{Output:}

$x^*$: The root of $x = \varphi(x)$.

\textbf{Parameters:}

$N$: Number of equations.
$d\Phi/dx_0$: Initial value of all $\partial\Phi_i/\partial x_i$.

**Default values:**
- 0.9 when not subiterating.
- 0.9999 when subiterating.

$R_{fac}$: Vector determining the rate at which $\partial\Phi_i/\partial x_i$ is decreased at each iteration.

**Default values:**
- 0.5 when not subiterating.
- 0.8 when subiterating.

$tol$: Solution accuracy.

**Default value:** $10^{-8}$

$I_{max}$: Maximum number of iterations.

**Default value:** 100

$I_s$: Maximum number of subiterations.

**Default value:** 1000 (This is intentionally set very high, since subiterations will still cease when certain conditions are met)

$\alpha_{max}$: Maximum allowed magnitude of $A$ matrix elements.

**Default value:** 2

$M_c$: Parameter for monotonicity check.

**Default value:** $-5 \cdot 10^{-2}$

**Functions:**

$A(R, J^{-1}) = I + (R - I)J^{-1}$

$\Phi(x, \varphi, A) = A(x - \varphi) + \varphi$

**Iteration loop:**

```matlab
% Initialize
R := d\Phi/dx_0 \cdot I
x\(^{(i-1)}\) := 0
% Computations are to be carried out on $J_f$ rather than $J_\varphi$
J_f := I - J
for i from 1 to $I_{max}$ do
  % Update $A$
  phi\(^{(i-1)}\) := $\varphi(x^{(i-1)})$
  J_f\(^{(i-1)}\) := $J_f(x^{(i-1)})$
  J_inv := inv(J_f\(^{(i-1)}\))
  A := A(R, J_inv)
  % Compute new guess, $x$
  x\(^{(i)}\) := $\Phi(x^{(i-1)}, phi\(^{(i-1)}\), A)
```

27
if $sub = true$ and $\max([|x(i) - x(i-1)| - |x(i-1) - x(i-2)|]) > 0$ then
  for $j$ from 1 to $I_s$ do
    % Compute Phi at current $x$
    $phi(i) := \varphi(x(i))$
    $Phi(i) := \Phi(x(i), phi(i), A)$
  endfor
  % Check validity of $x$
  for $n$ from 1 to $N$ do
    % Test for root between last and current guess
    $test_1 := x^{(i-1)} - x^{(i)}, (x^{(i)} - Phi_n^{(i)}) < M_c$
    % Test for critically large $A$
    $test_2 := \exists m \in [1, N]: |A_{n,m}| < \alpha_{max}$
    % Increase $R$ if necessary
    if $test_1 = true$ or $test_2 = true$ then
      $R_{n,n} := \frac{3R_{n,n}^2 + 1}{4}$
    else
      % Exit subiterations
      break
    endif
  endfor
  % Update $A$ and $x$
  $A := A(R, J_{inv})$
  $x^{(i)} := \Phi(x^{(i-1)}, phi^{(i-1)}, A)$
endfor
% Subiterations end
% Test for convergence
if $\text{mean}([|x(i) - x(i-1)|]) < tol$ then break
% Decrease $R$
$R := R_{fac} \cdot R$
endfor
return $x^{(n)}$

B.2.2 SIR.m

Here, we return not only the final solution, but a matrix containing the solution after each iterations step.

It should also be noted that the algorithm uses the helper function addPar, listed in B.6.4, to replace every occurrence of, for instance $x10$ with $x(10)$. This allows the command eval to properly insert the tenth element of the vector $x$. The reason that the string aren’t defined like this in the first place is that the jac function, listed in B.6.1, needs to construct symbolic variables with names identical to the elements of $x$, and parentheses are not allowed in variable names.
function U = SIR(fstr,Jstr,x0,sub)
    \% U = SIR(fstr,Jstr,x0)
    \% Solves the problem f(x) = x without using subiterations.
    \% \% fstr is a string containing the function definition for f
    \% \% Jstr has either of the following forms:
    \% \% - dense representation: Jstr is a string containing the
    \% \% function definition of the matrix J
    \% \% - sparse representation: Jstr = \{r,c,Jsp\} where r contains
    \% \% row indices, c contains column indices and Jsp is a string
    \% \% representation of the vector containing the non-zero
    \% \% elements of J.
    \% \% x0 is the initial guess.
    \% \% \% U = SIR(fstr,Jstr,x0,true)
    \% Solves the problem f(x) = x using subiterations.
    \% \% fstr, Jstr and x0 as above.

    if nargin < 4
        sub = 0; \% Default to no subiterations
    end
    if sub
        \% Values when subiterating
        Rfac=0.8; \% Reduction of R at each iteration
        dPdx=0.9999; \% Initial values of R
    else
        \% Values when not
        Rfac=0.5; \% Reduction of R at each iteration
        dPdx=0.9; \% Initial values of R
    end

    N=length(x0); \% Number of equations
    tol=1e-8; \% Solution accuracy
    imax=100; \% Maximum number of iterations
    Js=1000; \% Maximum number of subiterations
    a_c=2; \% Critical magnitude for alpha
    S1_min=-5e-2; \% Parameter for monotonicity check

    xold = zeros(size(x0)); \% Store the old x0
    R = dPdx*ones(N,1); \% RO
    U = []; \% Solution matrix

    \% If Jstr is a cell, we extract its component parts
    if iscell(Jstr)
        r = Jstr{1};
        c = Jstr{2};
        Jstr = Jstr{3};
        sparse_def = true;
    else
        sparse_def = false;
    end
for n = 1:imax
  % Append current guess to U
  U = [U x0];

  % ---------------
  % I: Compute new alpha
  % ---------------
  % Since fstr has x as its variable name, we assign x = x0 in order
  % to compute f(x0) and J(x0)
  x = x0;
  phi0 = eval(fstr);
  J = eval(Jstr);

  if sparse_def
    % Sparse version
    J = sparse(r,c,J,N,N);
    J = speye(N)-J;
  else
    % Dense version
    J = eye(N)-J;
  end

  % Test for singularity
  if issparse(J)
    condJ = rcond(full(J));
  else
    condJ = rcond(J);
  end
  if condJ == 0
    warning('Jacobian singular in SIR');
    return;
  end

  % Invert jacobian
  invJ = inv(J);

  % Compute R-I
  RI = RI_proc(R,N);

  % Compute alpha
  alpha = alpha_proc(RI,invJ,N);

  % ---------------
  % II: Compute x1
  % ---------------
  x1 = Phi_proc(x0,phi0,alpha);

% Only subiterate if step length is increasing
if sub && max(abs(x1-x0) - abs(x0-xold)) > 0
  % Precompute x0-Phi(x0) = x0-x1
  mon = x0-x1;
end
% III: Check validity of \( x_1 \)
%#-------------------------
for \( m=1:J_s \)
  % Evaluate \( \phi(x_1) \)
  \( x = x_1; \)
  \( \phi_{1} = \text{eval(fstr)}; \)
  \( n \) of \( S_1 \) is negative only if \( \phi \) has opposite signs at \( x_0 \) and \( x_1 \).
  \( S_1 = \text{mon} \odot (x_1 - \text{Phi}_\text{proc}(x_1,\phi_{1},\alpha)); \)
  % Check for critically large \( \alpha \) by taking the row-wise max
  \( S_2 = \text{max}(|\alpha|,[],2); \)
  % Perform all comparisons at once, store in boolean vector \( \text{ifR} \)
  % A particular row of \( \text{ifR} \) will be true if subiteration is
  % required in that dimension.
  \( \text{ifR} = (S_2 >= a_c \land S_1 < S_1_{\text{min}}); \)
  % Break if no subiterations are needed
  if max(\( \text{ifR} \)) == 0 break end
  % Increase \( R \) towards \( I \)
  for \( i=1:N \)
    % If subiterations are required in the \( i \):th dimension,
    % increase the \( i \):th row of \( R \)
    if \( \text{ifR}(i) = 1 \)
      \( R(i) = (3*R(i)+1)*0.25; \)
    end
  end
  % Update \( RI \)
  \( RI = \text{RI}_\text{proc}(R,N); \)
  % Recompute \( \alpha \)
  \( \alpha = \text{alpha}_\text{proc}(\text{RI},\text{invJ},N); \)
  \( x_1 = \text{Phi}_\text{proc}(x_0,\phi_{0},\alpha); \)
end %-------------------------
% Subiterations end here:
%-------------------------
%xold = x0; % Backup \( x_0 \) so that it can be used to compare step lengths.
x0 = x1;
if eps < tol
  break end
% Decrease \( R \)
\( R = R_{\text{fac}}*R; \)
% Append final guess to \( U \)
\( U = [U \ x_0]; \)
functions

function alpha = alpha_proc(RI,invJ,N)
    alpha = eye(N)+RI*invJ;

function Phi = Phi_proc(x,phi,alpha)
    Phi = alpha*(x-phi)+phi;

function RI = RI_proc(Rd,N)
    RI = spalloc(N,N,N);
    for i=1:N
        RI(i,i) = Rd(i)-1;
    end

B.3 MATLAB/Maple Hybrid Implementation of GWRM

This implementation consists of three parts:

Burger.mpl: Maple code defining all input arguments necessary to solve Burgers’ equations using GWRM.

SIR.m: A MATLAB implementation of SIR, modified to handle spatial sub-domains.

GWRM.m: MATLAB code tying the two together.

B.3.1 GWRM.m

% Restart maple engine
maple('restart');

% Define problem in Maple
procread('Burger.mpl');

% Extract symbolic variables
clear phi X Ns J;

% Convert X to double
X = double(X);

% Call SIR
U = SIR(phi,copy(J),X,0);

B.3.2 Burger.mpl

This is a largely unmodified version of the GWRM code for solving Burger’s equations given in [1]. Minor changes has been made near the end to use the Matrix and Array data structures.
Digits:=14:
with(linalg):
with(numapprox):
with(orthopoly):

# Parameters.
Ns:=2:  #Number of spatial subdomains
m_max:=6:  #Order of temporal modes
n_max:=8:  #Order of spatial modes
kappa:=0.01:  #Viscosity

# Subdomain stuff.
Lt:=0:  Rt:=5.0:

for j from 0 to Ns+1 do
    Rx[j]:=evalf(j/(Ns+1)):
    xb[j]:=Rx[j]:
    Lx[j]:=Rx[j-1]:
    epsBC[j]:=0.0:
od:

for j from 0 to Ns+1 do
    xb[j]:=Rx[j]:
    Lx[j]:=Rx[j-1]:
    epsBC[j]:=0.0:
od:

BMAt:=0.5*(Rt-Lt):
BPAt:=0.5*(Rt+Lt):

for j from 1 to Ns do
    BMAx[j]:=0.5*(Rx[j+1]-Rx[j-1]):
    BPAx[j]:=0.5*(Rx[j+1]+Rx[j-1]):
od:

# Support functions to Chebyshev expansions.
for j from 1 to 2*max(m_max,n_max) do fp[j]:=1 od:
fp[0]:=1/2:
for k from -2*max(m_max,n_max) to 2*max(m_max,n_max) do delta[k]:=0 od:
delta[0]:=1:

# The Chebyshev expansion.
for j from 1 to Ns do
    test_u[j]:=simplify(sum(sum(u[j][m,n]*fp[m]*fp[n]*T(m,(t-BPAt)/BMAt)*T(n,(x-BPAx[j])/BMAx[j]),m=0..m_max),n=0..n_max)):
od:

# Procedures for computation of PWRM-coefficients for Burger's equation.
Derivative_2x:=proc(a,BMA,A)
    local Asum,k,n,M,n,N;
    global m_max,n_max:
    N:=m_max+1;  #Derivatan minskar ordningen.
    for n from 0 to n_max do
        A[m,N+1]:=0;
        A[m,N+2]:=0;
        for n from N+1 by -1 to 1 do
            A[m,n-1]:=A[m,n+1]+2*n*a[m,n]/BMA;
        od;
    od;
end:
Product_2D:=proc(A,B,C,K,L)
local k,l,m,n,r,s,Ax,Bx;
global delta,fp;

for k from 0 to 3*K do
for l from 0 to 3*L do
Ax[k,l]:=0;
Bx[k,l]:=0; od: od:

for k from 0 to K do
for l from 0 to L do
Ax[k,l]:=fp[k]*fp[l]*A[k,l];
Bx[k,l]:=fp[k]*fp[l]*B[k,l];
od: od:

for r from 0 to 1*K do
for s from 0 to 1*L do
C[r,s]:=((1/4)*add(add(Ax[m,n]*
((1+delta[r-m])*(1+delta[s-n])*Bx[abs(r-m),abs(s-n)]+
(1+delta[r-m])*(1-delta[s])* Bx[abs(r-m),s+n]+(1-delta[r])* (1+delta[s-n])*Bx[r+m,abs(s-n)]+
(1-delta[r])* (1-delta[s])* Bx[r+m,s+n])
,m=0..K),n=0..L));
od: od:

for k from 0 to 1*K do
C[k,0]:=2*C[k,0] od:
for l from 0 to 1*L do
C[0,l]:=2*C[0,l] od:
end:

Integral_2t:=proc(a,BMA,A)
local Asum,m,n;
global m_max,n_max;
for n from 0 to n_max do
Asum:=0:
a[m_max+1,n]:=0:
a[m_max+2,n]:=0:
for m from 1 to m_max+1 do
A[m,n]:=BMA*(a[m-1,n]-a[m+1,n])/(2*m);
Asum:=Asum+cos(m*Pi)*A[m,n];
od:
A[0,n]:=-2*Asum;
od:
end:

# The PDE is transformed to Chebyshev space
# and distributed into subdomains.
for Is from 1 to Ns do
Derivative_2x(a, BMAx[Is],a1):
Derivative_2x(a1,BMAx[Is],a2):
Product_2D(a,a1,a2,m_max,n_max):
for m from 0 to n_max do
for n from 0 to n_max do
a4[m,n]:=-a3[m,n]+kappa*a2[m,n];
od: od:
Integral_2t(a4,BMAt,A0):
assign(a,u[Is]):
for m from 0 to m_max do
for n from 0 to n_max do
A[Is][m,n] := A0[m,n]:
od: od:

# BC's are applied.
BC_proc := proc(u)
# BC:s are imposed at modal points n_max-1 and n_max.
local ii,j,jx,k,mx,m,n,BCtest,del,S,Sd,i_Eq,Eq:
unassign('u'):
for m from 0 to m_max do
if Ns >= 2 then
S[1,0] := sum(fp[q]*u[1][m,q]*T(q,(xb[0]-BPax[1])/BMAx[1]),q=0..n_max):
S[1,1] := sum(fp[q]*u[1][m,q]*T(q,(xb[1]+epsBC[2]-BPax[1])/BMAx[1]),q=0..n_max):
S[1,2] := sum(fp[q]*u[1][m,q]*T(q,(xb[2]-epsBC[1]-BPax[1])/BMAx[1]),q=0..n_max):
for n from 2 to Ns-1 do
S[n,0] := sum(fp[q]*u[n][m,q]*T(q,(xb[n-1]+epsBC[n]-BPax[n])/BMAx[n]),q=0..n_max):
S[n,1] := sum(fp[q]*u[n][m,q]*T(q,(xb[n]-BPax[n])/BMAx[n]),q=0..n_max):
S[n,2] := sum(fp[q]*u[n][m,q]*T(q,(xb[n]+epsBC[n+1]-BPax[n])/BMAx[n]),q=0..n_max):
S[n,3] := sum(fp[q]*u[n][m,q]*T(q,(xb[n+1]-epsBC[n]-BPax[n])/BMAx[n]),q=0..n_max):
fi:
if Ns = 1 then
S[1,0] := sum(fp[q]*u[1][m,q]*T(q,(xb[0]-BPax[1])/BMAx[1]),q=0..n_max):
S[1,2] := sum(fp[q]*u[1][m,q]*T(q,(xb[2]-BPax[1])/BMAx[1]),q=0..n_max):
Eq[1] := subs(u=ux,S[1,0]):
Eq[2] := subs(u=ux,S[1,2]):
fi:
if Ns = 2 then
Eq[1] := subs(u=ux,S[1,0]):
Eq[2] := subs(u=ux,S[1,2])-S[2,1]:
Eq[3] := subs(u=ux,S[2,0])-S[1,1]:
Eq[4] := subs(u=ux,S[2,2]):
fi:
if Ns >= 3 then
Eq[1] := subs(u=ux,S[1,0]):
Eq[2] := subs(u=ux,S[1,2])-S[2,1]:
for n from 2 to Ns-1 do
Eq[2+n-1] := subs(u=ux,S[n,0])-S[n-1,1]:
Eq[2+n] := subs(u=ux,S[n,3])-S[n+1,1]:
od:
Eq[2+Ns-1] := subs(u=ux,S[Ns,0])-S[Ns-1,2]:
Eq[2+Ns] := subs(u=ux,S[Ns,2]):
fi:
for n from 1 to Ns do
Sol[n]:=solve({seq(Eq[ii],ii=2*n-1..2*n)},{seq(ux[n][m,j],j=n_max-1..n_max)});
assign(Sol[n]):
end:
end:

# Initial conditions (IC) are determined.
# Here we use u(0,x) = x*(1-x).
# Note, that the coeffs are different for different intervals.
for Is from 1 to Ns do
for n from 0 to n_max do
b_IC[Is][n]:=0:
od: od:
for j from 1 to Ns do
b_IC[j][0]:=-2.*BPAx[j]^2+2.*BPAx[j]-1.*BMAx[j]^2:
b_IC[j][1]:=-2.*BMAx[j]*BPAx[j]+BMAx[j]:
b_IC[j][2]:=-.500000000000*BMAx[j]^2:
od: od:

# All Chebyshev coefficients for the TP-WRM iterations are computed, # including BC and IC.
for Is from 1 to Ns do
for n from 0 to n_max-2 do
for m from 0 to m_max do
q[Is][m,n]:=A[Is][m,n]:
od: od: od:
for j from 1 to Ns do
for k from 1 to m_max+1 do
for j from 1 to n_max+1-0*2 do
q[Is][k-1,j-1]:=subs(ux=u,ux[Is][m,j]):
od: od: od:

# Create phi
for Is from 1 to Ns do
for k from 1 to m_max+1 do
for j from 1 to n_max+1-0*2 do
u[Is][k-1,j-1]:=x[(m_max+1)+(k-1)+j,Is]:
od: od: od:

# Create phi
for Is from 1 to Ns do
for k from 1 to m_max+1 do
for j from 1 to n_max+1 do
phi[(m_max+1)*(k-1)+j, Is]:=expand(q[Is][k-1,j-1]):
od: od: od:
# Create Jacobians
for Is from 1 to Ns do
for j from 1 to N do
for k from 1 to N do
    J[j,k,Is]:=diff(phi[j,Is],x[k,Is]):
od:
end:
end:
end:
%
% Create x0
for Is from 1 to Ns do
for j from 1 to n_max+1 do
    X[j,Is]:=2.0*b_IC[Is][j-1]
end:
end:

B.3.3 SIR.m

There are a few ways in which this implementation differs from the pure MATLAB implementation listed in appendix B.2.2:

- An inner loop over spatial subdomains has been added.
- As mentioned in section 3.4.2, another dimension is added to matrices and vectors.
- \( \phi \) and \( J \) are given as symbolic expressions rather as character strings. These symbolic expressions are evaluated as follows: First, a value is assigned to the dependent variable, \( x \), in the Maple environment using the setmaple command. Then the expression is converted to a numeric representation using the double command.
- Since \( J \) is now a symbolic expression, it can be transformed into \( I - J \) once in the initialization step. In the pure MATLAB implementation, this was done after \( J \) was evaluated at each iteration loop.

function U = SIR(phi,J,x0,sub)
% U = SIR(phi,J,x0)
% Solves the problem \( \phi(x) = x \), with starting guess \( x0 \) without using
% subiterations.
% % phi is a symbolic NxM matrix, with M being the number of
% % spatial subdomains.
% % x0 is a constant NxM matrix.
% % J is a 3D NxNxM matrix, where the m:th slice contains
% % the NxN Jacobian of the m:th column of \( f \) with respect to the m:th column
% % of \( x \)
% % U is a cell array consisting of M cells, The m:th cell contains a matrix
% with columns containing the solution after each iteration step.
% % U = SIR(phi_func,J_func,x0,sub)
% As above, with the value of sub (true or false) determining whether or
% not to use subiterations.
% Parameters

if nargin < 4
    sub = false; % Default to no subiterations
end
if sub
    % Values when subiterating
    Rfac=0.8; % Reduction of R at each iteration
    dPdx=0.9999; % Initial values of R
else
    % Values when not
    Rfac=0.5; % Reduction of R at each iteration
    dPdx=0.9; % Initial values of R
end

[N M] = size(x0); % Number of equations

tol=1e-8; % Solution accuracy
imax=100; % Maximum number of iterations
Js=1000; % Maximum number of subiterations
a_c=2; % Critical magnitude for alpha
S1_min=-5e-2; % Parameter for monotonicity check

% Initialization

xold = zeros(size(x0)); % Store the old x0
R = dPdx*ones(N,M); % R0, now also an N x M matrix
eps_acc = 1; % Dummy value for accuracy test
x1 = x0; % x1 will be updated one spatial subdomain at a time
for i = 1:M
    U{i} = []; % Initialize solution array
end

% Transform J into I-J
for i=1:M
    if issparse(J(:,:,i))
        % Sparse version
        J(:,:,i) = speye(N)-J(:,:,i);
    else
        % Dense version
        J(:,:,i) = eye(N)-J(:,:,i);
    end
end
IJ = J;

% Iteration Loop

for n = 1:imax
    % Inner loop over spatial subdomains
    for m = 1:M
        % Update U
        U{m} = [U{m} x0(:,m)];
        % -------------------
        % I: Compute new alpha
        % -------------------
        % Evaluate functions at x=x0
        setmaple('x',x0);
% Evaluate phi(x0) and IJ(x0)
% Note that all xs and x0s may be present in each subdomain
phi0 = double(phi(:,m));
IJ0 = double(IJ(:,:,m));
% Invert Jacobian
invIJ = inv(IJ0);

% Compute R-I, here we need to use the m:th column of R
RI = RI_proc(R(:,m),N);
% Compute alpha
alpha = alpha_proc(RI,invIJ,N);

% -------------------
% II: Compute x1
% -------------------
% Update the m:th subdomain of x1
x1(:,m) = Phi_proc(x0(:,m),phi0,alpha);

%%%%%%%%%%%%%%%%%%%
%% Subiterations %%
%%%%%%%%%%%%%%%%%%%
% Only subiterate if step length is increasing in this subdomain
if sub && max(abs(x1(:,m)-x0(:,m)) - abs(x0(:,m)-xold(:,m))) > 0

% Precompute x0-Phi(x0) = x0-x1 in this subdomain
mon = x0(:,m)-x1(:,m);

% -------------------------
% III: Check validity of x1
% -------------------------
for j=1:Js
% Evaluate functions at x=x1
setmaple('x',x1);
% Evaluate phi(x1)
phi1 = double(phi(:,m));

% - The element-wise multiplication means that each element
%  n of S1 is negative only if, in the n:th dimension,
%  phi has opposite signs at x0 and x1.
S1 = mon .* (x1(:,m)-Phi_proc(x1(:,m),phi1,alpha));

% Check for critically large alpha by taking the row-wise max
S2 = max(abs(alpha),[],2); % Check for critically large alpha
% Perform all comparisons at once, store in boolean vector ifR
% A particular row of ifR will be true if subiteration is
% required in that dimension.
ifR = (S2 >= a_c | S1 < S1_min);
% Break if no subiterations are needed
if max(ifR) == 0
    break;
end
for i=1:N
    % Increase the m:th column of R towards I
    % If subiterations are required in the i:th dimension,
    % increase the i:th row of R
    if ifR(i) == 1
        R(i,m) = (3*R(i,m)+1)*0.25;
    end
end
% Update RI
RI = RI_proc(R(:,m),N);

% Recompute alpha
alpha = alpha_proc(RI,invJ,N);
xi(:,m) = Phi_proc(x0(:,m),phi0,alpha);
end
% Subiterations end here:
% ----------------------------------------

% ------------------------
% Accuracy test and update
% ------------------------
% compute accuracy in each subdomain, note that
% eps_acc is NOT a cell, but a vector of size M
eps_acc(m) = mean(abs(xi(:,m)-x0(:,m)));
xold(:,m) = x0(:,m); % Backup x0 so that it can be used to
% compare step lengths.
x0(:,m) = xi(:,m);

% Decrease R
R(:,m) = Rfac*R(:,m);
end
if max(eps_acc) < tol
break
end
% Add last point to U
for m = 1:M
    U(m) = [U(m) x0(:,m)];
end

function alpha = alpha_proc(RI,invJ,N)
% Compute alpha = I + (R-I)*J^(-1)
alpha = eye(N)+RI*invJ;
end

function Phi = Phi_proc(x,phi,alpha)
% Computes Phi = alpha*(x-phi)+phi
% x and phi are allowed to be matrices, in which case Phi will be also
Phi = alpha*(x-phi)+phi;
end

function RI = RI_proc(Rd,N)
% Computes R-I
% Rd is a vector containing the diagonal elements of R
RI = spalloc(N,N,N);
for i=1:N
    RI(i,i) = Rd(i)-1;
end

B.4 MATLAB Implementation of NL
For a thorough description of this algorithm, the reader is directed to [3], which
covers the C++ version this is adapted from.
The translation is in many ways similar to the translation of SIR from Maple
to MATLAB. For instance, functions are implemented as strings and evaluated
in the same fashion and for loops have been vectorized. A proper Jacobian has also replaced the finite difference approximations used in the C++ implementation.

The implementation consists of two files, \texttt{NL.m} and \texttt{lnsrch.m}.

**B.4.1 NL.m**

```matlab
function U = newtNR(fstr,Jstr,x)
% \[ U, check \] = newtNR(fstr,Jstr,x0)
% % Solve f(x) = 0, using Newton's Method with Line Search.
% % - x0 is the initial guess.
% % - fstr and Jstr are string representations of f and its
% % Jacobian, respectively.
% % - check stores whether or not a spurious root (i.e. local
% % minimum) was encountered.

MAXITS = 100; % Maximum number of iterations
TOLF=1e-8; % Convergence criterion on function values
TOLMIN=1e-12; % Criterion for spurious roots
STPMX=100; % Scaled maximum step length in line searches
TOLX=1e-8; % Convergence criterion on x

% If Jstr is a cell, we extract its component parts
if iscell(Jstr)
    r = Jstr{1};
    c = Jstr{2};
    Jstr = Jstr{3};
    J_sparse = true;
else
    J_sparse = false;
end;

U = x; % Initialize solution matrix
n = length(x); % Derive N from the length of x

% replace every occurrence of xi with x(i)
Jstr = addPar(Jstr,n);
frstr = addPar(fstr,n);

% compute f(x)
fvec = eval(fstr);
% compute fmin, the function to be minimized
f = fmin_func(fvec);
% Test for initial guess being a root.
% Use more stringent test than simply TOLF
if max(abs(fvec)) < 0.01*TOLF
    check = false;
    return;
end
% Calculate stpmax for line searches
stpmax = STPMX*max(norm(x),n);
% Start of iteration loop
for its=1:MAXITS
    % Compute jacobian at x
    fjac = eval(Jstr);
    if J_sparse
```
fjac = sparse(r,c,fjac,n,n);
end
%
% Test for singularity
if J_sparse
    condJ = rcond(full(fjac));
else
    condJ = rcond(fjac);
end
if condJ == 0
    warning('Jacobian singular in newtNR');
    check = false;
    return
end
%
% Compute the gradient of f for the line search
% NOTE: for column vectors, the jacobian vector needs to be transposed
% g = fjac'*fvec;
% xold = x; % Store x,
% fold = f; % and f
% Compute Newton step
p = -inv(fjac)*fvec;
%
% Perform line search
[x,f,fvec,check] = lnsrch(xold,fold,g,p,stpmax,fstr,@fmin_func);
%
% Append guess to solution
U = [U x];
%
% Test for convergence on function values
if max(abs(fvec)) < TOLF
    check = false;
    return;
end
%
% Check for gradient of f zero, i.e. spurious convergence.
if check
    den = max(f,0.5*n);
    test = max(abs(g).*max(abs(x),1)/den);
    check = (test < TOLMIN);
    return;
end
%
% Test for convergence on x
if max(abs(x-xold)./max(abs(x),1)) < TOLX
    return;
end
end
error('MAXITS exceeded in newtNR');
%
% The function of which to find a global minimum: \langle f, f \rangle / 2
function fmin = fmin_func(fvec)
    fmin = 0.5*dot(fvec,fvec);
end

B.4.2 lnsrch.m
function [x1, f1, fvec1, check] = lnsrch(x0, f0, grad, dx, stpmax, fstr, fmin)
% The heart of the line search function from numerical recipes
alpha = 1e-4; % Ensures sufficient decrease in function value
tolx = eps; % Convergence criterion on x
lambda2 = 0;
f2 = 0;
check = false;
% Scale if attempted step is too big
if norm(dx) > stpmax
    dx = dx .* stpmax/norm(dx);
end

slope = dot(grad,dx);
if slope >= 0
    error('Roundoff problem in lnsrch');
end

% Compute lambda_min
test = max(abs(dx)./max(abs(x0),1));
lambda_min = tolx/test;
% Always try the full Newton step first
lambda = 1;
% Start of iteration loop
while true
    x1 = x0 + lambda*dx;
    x = x1;
    fvec1 = eval(fstr);
    f1 = feval(fmin,fvec1);
    % Convergence on x. For zero finding,
    % the calling program should verify the convergence
    if lambda < lambda_min
        x1 = x0;
        check = true;
        return;
    % Sufficient function decrease (first Wolfe cond.)
    elseif f1 < f0 + alpha*lambda*slope
        return;
    % Backtrack
    else
        % First time
        if lambda == 1
            tmplam = -slope/(2*(f1-f0-slope));
        % Subsequent backtracks
        else
            rhs1 = f1-f0-lambda*slope;
            rhs2 = f2-f0-lambda2*slope;
            X(1) = (rhs1/(lambda*lambda) - rhs2/(lambda2*lambda2))/(lambda-lambda2);
            X(2) = (-lambda2*rhs1/(lambda*lambda)+lambda*rhs2/(lambda2*lambda2))/(lambda-lambda2);
            % Update alpha
            % A = 0
            if(X(1) == 0)
                tmplam = -slope/(2*X(2));
            % A != 0
            else
                disc = X(2)*X(2) - 3*X(1)*slope;
                if (disc < 0)
                    % Imaginary roots, set lambda = 0.5*lambda1
                    tmplam = 0.5*lambda;
                elseif X(2) <= 0
                    tmplam = (-X(2) + sqrt(disc))/(3*X(1));
                else
                    tmplam = -slope/(X(2) + sqrt(disc));
                end
            end
            lambda <= 0.5*lambda1
            if tmplam > 0.5*lambda
                tmplam = 0.5*lambda;
            end
        end
    end
end

43
B.5 MATLAB/Maple Hybrid Implementation of SIR

In the pure MATLAB implementation of SIR, limitations of the symbolic toolbox were circumvented using methods that are only viable in specific (older) versions of MATLAB. For readers who wish to implement SIR in a more recent version, we present a version utilising the Maple toolbox, as mentioned in section 3.4.1.

This implementation will have many similarities to the implementation of GWRM given in appendix B.3. It consists of the following parts:

prob\_x0\_mpl: Maple code defining $\varphi$ and the initial guess $x_0$.
jac.mpl: Maple code computing the Jacobian of $\varphi$.
jac\_sp.mpl: Maple code computing the Jacobian of $\varphi$ and storing it as a sparse definition (see appendix B.1).
SIR.m: A MATLAB implementation of SIR.
runSIR.m: MATLAB code tying the above together.

B.5.1 runSIR.m

When executed in MATLAB, runSIR.m will solve the problem specified in its parameters and store the result in the matrix $U$.

Note that we set the value of $N$ in the Maple environment, so that it can be used in the function definitions and Jacobian computations.

```matlab
% Parameters
prob = 'prob_cos'; % Which problem to solve
sparse_def = true; % Whether or not to define jacobian as sparse
sub = true; % Whether or not to subiterate
N = 2; % Number of dimensions

% Restart Maple engine
maple('restart');
setmaple('N',N);

% Define problem
% Run the specified problem definition
procread([prob '.mpl']);

% Compute jacobian
if sparse_def
    % Sparse definition
    procread('jac_sp.mpl');
else
    % Dense definition
    procread('jac.mpl');
end
```

44
% Extract symbolic variables
clear phi J x0 r c;
syms phi J x0;
if sparse_def
    syms r c;  % Extract additional variables required
    J = {r,c,J};  % Store entire jacobian definition in cell
end

% Execute SIR
U = SIR(phi,J,x0,sub);

B.5.2 prob.cos.mpl
This is an example of how a problem definition can be implemented, which will define the problem given in eq. 17. It is important that the variables are named phi and x0, since these names will be referenced later on.

As was the case in appendix B.3, we use the Matrix data structure, making variables available in both the Maple and MATLAB environments.

x0 := Matrix(N,1):
phi := Matrix(N,1):
for j from 1 to N-1 do
    phi[j] := cos(x[j+1,1]):
od:
phi[N] := 3*cos(x[1,1]):
for j from 1 to N do
    x0[j] := -2:
od:

B.5.3 jac.mpl
Here we compute the Jacobian of the function stored in phi with respect to x and store the result in the Matrix J.

J := Matrix(N,N):
for j from 1 to N do
    for k from 1 to N do
        J[j,k] := diff(phi[j,1],x[k,1]):
od:
od:

B.5.4 jac_sp.mpl
This time, we store only the nonzero elements of the Jacobian in the Array J, and their column and row indices in the Arrays r and c, respectively.

# Define Arrays that are sure to fit the at most
# N*N nonzero elements
J := Array(1..N*N):
r := Array(1..N*N):
c := Array(1..N*N):
# Counter for nonzero elements
count := 0:
for j from 1 to N do
    for k from 1 to N do
        # Test if derivative is nonzero
        d := diff(phi[j,1],x[k,1]):
if d <> 0 then
    # Increase counter
    count := count + 1;
    # Store indices
    r[count] := j;
    c[count] := k;
    # Store the current derivative
    J[count] := d:
fi:
od:
od:
# Shrink the arrays to fit the data
J := J[1..count];
r := r[1..count];
c := c[1..count]:

B.5.5 SIR.m

This version of SIR resembles the pure MATLAB implementation listed in appendix B.2.2, except that we allow it to handle symbolic variables in the same way as the GWRM implementation listed in appendix B.3.3.

function U = SIR(phi,J,x0,sub)

%%%%%%%%%%%%%%%%
%% Parameters %
%%%%%%%%%%%%%%%%
if nargin < 4
    sub = false; % Default to no subiterations
end
if sub
    % Values when subiterating
    Rfac=0.8; % Reduction of R at each iteration
    dPdx=0.9999; % Initial values of R
else
    % Values when not
    Rfac=0.5; % Reduction of R at each iteration
    dPdx=0.9; % Initial values of R
end

N = length(x0); % Number of equations
tol=1e-8; % Solution accuracy
imax=100; % Maximum number of iterations
Js=1000; % Maximum number of subiterations
a_c=2; % Critical magnitude for alpha
S1_min=-5e-2; % Parameter for monotonicity check

%%%%%%%%%%%%%%%%%%%
%% Initialization %
%%%%%%%%%%%%%%%%%%%
xold = zeros(size(x0)); % Store the old x0
R = dPdx*ones(N,1); % R0
U = []; % Solution matrix

46
% If J is a cell array, we extract its component parts
if iscell(J)
    r = double(J{1});
    c = double(J{2});
    J = J{3};
    sparse_def = true; % Flag sparse version
else
    sparse_def = false; % Flag dense version
end

%%%%%%%%%%%%%%%%%%%%
%% Iteration Loop %
%%%%%%%%%%%%%%%%%%%%
for n = 1:imax
    U = [U double(x0)];

    % ---------------------
    % I: Compute new alpha
    % ---------------------
    setmaple('x',x0);
    phi0 = double(phi);
    J0 = double(J);

    % Transform J0 into I-J0
    if sparse_def
        % Sparse version
        J0 = sparse(r,c,J0,N,N);
        J0 = speye(N)-J0;
    else
        % Dense version
        J0 = eye(N)-J0;
    end

    % Test for singularity
    condJ = rcond(full(J0)); % Convert to full matrix, in case J is sparse.
    if condJ <= eps
        error('Jacobian singular in SIR');
    end
    invJ = inv(J0);

    % Compute R-I
    RI = RI_proc(R,N);
    % Compute alpha
    alpha = alpha_proc(RI,invJ,N);

    % ------------------
    % II: Compute x1
    % ------------------
    x1 = Phi_proc(x0,phi0,alpha);

    %-------------------------------------------
    % Subiterations %
    %-------------------------------------------
    % Only subiterate if step length is increasing
    if sub && max(abs(double(x1-x0)) - abs(double(x0-xold))) > 0
        mon = x0-x1;
for j=1:Js
  % Evaluate phi(x1)
  setmaple('x',x1);
  phi1 = double(phi);
  S1 = mon .* (x1-Phi_proc(x1,phi1,alpha));
  S2 = max(abs(alpha),[],2);
  % Perform all comparisons at once, store in boolean vector ifR
  % A particular row of ifR will be true if subiteration is
  % required in that dimension.
  ifR = (S2 >= a_c | S1 < S1_min);
  % Break if no subiterations are needed
  if max(ifR) == 0
    break;
  end
  % Increase R towards I
  for i=1:N
    % Increase i:th dimension
    if ifR(i) == 1
      R(i) = (3*R(i)+1)*0.25;
    end
  end
  % Update RI
  RI = RI_proc(R,N);
  % Recompute alpha
  alpha = alpha_proc(RI,invJ,N);
  x1 = Phi_proc(x0,phi0,alpha);
end

% Accuracy test and update
% ------------------------
eps_acc = mean(abs(double(x1-x0))); xold = x0; % Backup x0 so that it can be used to compare step lengths.
x0 = x1;
if max(eps_acc) < tol
  break
end

% Decrease R
% ----------
% This was moved to the end of the iteration step, rather than being % performed at the beginning of all steps but the first. This removes % the need to test if n>1 at each step.
R = Rfac*R;
% Add last point to U
U = [U double(x0)];
% Functions

function alpha = alpha_proc(RI,invJ,N)
alpha = eye(N)+RI*invJ;

function Phi = Phi_proc(x,phi,alpha)
% Computes Phi (x and phi are allowed to be matrices)
Phi = alpha*(x-phi)+phi;

function RI = RI_proc(R,N)
% NOTE: for N =~1000, this function takes up a fair bit of the execution
% time if RI is not declared sparse.
% For smaller values of N (< ~500) this is less efficient than
% the normal, dense, declaration.
RI = spalloc(N,N,N);
for i=1:N
    RI(i,i) = R(i)-1;
end

B.6 Helper Functions

This section lists the various helper functions used. Each begins with a summary of its purpose and the comments included in the code should help the reader get the gist of how they work.

B.6.1 jac.m

Computes the Jacobian of a function defined in a character string. As described in appendix B.1, depending on number of output parameters, this is either returned as a dense or sparse representation.

Depending on representation, one of the two helper functions map2mat (see appendix B.6.2) or map2sparse (see appendix B.6.3) are used to transform the Maple result into MATLAB format.

function varargout = jac(f)
% Transpose f by replacing semicolons with commas
semis = findstr(f,';');
f(semis) = ',';
% Determine N as one more than the number of semicolons in f
N = length(semis)+1;
if N == 1
    error('f is either zero-dimensional or incorrectly specified');
end
% generate the string [x1,...,xN]
x = '['; % Initialize
for i = 1:N
    x = [x 'x' num2str(i) ',']; % Append 'xi,'
end
% replace trailing comma with a closing bracket
x(end) = '];';

% Let Maple compute jacobian
J = maple('jacobian',f,x);
% Sparse matrix output
if nargout == 3
    [J r c] = map2sparse(J);
    varargout = {J,r,c};
% Dense matrix output
elseif nargout == 1
    varargout = {map2mat(J)};
else
    error('Wrong number of output arguments');
end

B.6.2 map2mat.m

Lifted directly from the source code of the symbolic toolbox. Converts a Maple string containing a matrix, vector or array into a MATLAB string by performing search-and-replace operations.

function r = map2mat(r)
    % Deblank.
    r(findstr(r,' ')) = []; % Special case of the empty matrix or vector
    if strcmp(r,'vector([[]])') | strcmp(r,'matrix([[]])') | ... % Remove matrix, vector, or array from the string.
        r = [];
    else
        r = strrep(r,'matrix([[[','['); r = strrep(r,'array([[[','[');
        r = strrep(r,'vector([','['); r = strrep(r,'][',']');
        r = strrep(r,']],[','],']); r = strrep(r,']],[',']);
    end

B.6.3 map2sparse.m

A more complicated version of map2mat. map2sparse will only work on square matrices and returns a sparse representation as described in appendix B.1.

function [r,c,S] = map2sparse(D)
    % We assume D is a square maple-formatted matrix string:
    % D = 'matrix([[...],[...],[...]])'
    % First, we convert the matrix D into a matlab row vector:
    % Does the string start with a unary minus sign?
    if D(1) == '-'
        % If so, S should also
        S = '-[[';
        % Remove the leading '-'-matrix([' and trailing ']')
        D([1:9 end-1 end]) = [];
    else
        % Otherwise
        S = '[[
        % Remove the leading 'matrix([' and trailing ']')
        D([1:8 end-1 end]) = [];
    end
    % After this, the result will be a Matlab row vector
    D = strrep(D, [',',','],[',',']);
    % Deblank
    D(findstr(r,' ')) = [];

50
Now, we find the commas that separate the columns:

% Now, we find the commas that separate the columns:
cs = findstr(D,'
,');

% Also find the (middle) zeros
zs = findstr(D,',0,');

% Compute \( N \) from the size of \( cs \) (number of commas = number of elements - 1)
N = sqrt(length(cs)+1);

r = []; % Initiate row indices
c = []; % Initiate column indices
z = 1; % Zero counter

% First we check the very first element:
% We know where the first comma occurs, if we look at the two preceding
% characters and find that they are '0' we know that the first element is
% a zero. Note that we cannot just check one character back, since we might
% have, for instance, D starting with '-matrix[100,...'
if ~strcmp(D(cs(1)-2:cs(1)-1),'
,0')
    % If the first element is nonzero, then we append it to S.
    % We know that the first element starts at the second character (the
    % first being the opening bracket) and ends just before the first
    % comma. We also store the coordinates of the element (1,1) into
    % r and c
    S = [S D(2:cs(1)-1) ','];
    r = 1;
    c = 1;
end

% Now we check the middle elements (all but the first and last one).
% Note that the index of the last ',0,' corresponds to the second
% last comma, so we can't use the last comma this way.
% First we check if there are any middle zeros at all.
if length(zs) > 0
    for k = 1:length(cs)-1
        % Does this comma match up with a zero?
        if cs(k) == zs(z)
            % If so, we move the zero counter
            if z == length(zs)
                % If we're already at the end, we copy the remaining
                % middle elements right away and exit the for loop.
                % This corresponds to every char after the *next* comma
                % (since this was a zero element) up to and including the
                % last one.
                S = [S D(cs(k+1):end)];
            % Compute the coordinates: Since k indexes a row matrix,
            % we must swap i and j when using ind2sub. Since we start
            % counting at the second element, we add one to the index.
            % Note that we're starting at k+1 where the first nonzero
            % element is, so the first index number will be k+2.
            % Finally, we're adding all the remaining middle elements,
            % fortunately ind2sub has no problem returning i and j as
            % vectors.
                [j,i] = ind2sub([N,N],k+2:length(cs));
                r = [r i];
                c = [c j];
                break;
            end
        % Otherwise, we can safely move the zero counter forward
            z = z + 1;
        end
    end
end
else
    \% If not, we store this element in \( S \) by copying everything
    \% after the current comma up to and including the next one.
    \( S = [S \ D(cs(k)+1:cs(k+1))]; \)
    \% We also store the row and column indices of this element.
    \([j,i] = \text{ind2sub}(\lfloor N,N\rfloor,k+1); \)
    \( r = [r \ i]; \)
    \( c = [c \ j]; \)
end
else
    \% If there were no middle zeros, we know that we can just copy all the
    \% middle elements from \( D \) into \( S \). This is every char between the first
    \% and last comma.
    \( S = [S \ D(cs(1)+1:cs(end))]; \)
    \% Store all indices into \( r \) and \( c \), as above.
    \([j,i] = \text{ind2sub}(\lfloor N,N\rfloor,2:length(cs)); \)
    \( r = [r \ i]; \)
    \( c = [c \ j]; \)
    \% We also warn that this might not be the most efficient storage method
    \% for such a matrix.
    \text{warning}('Matrix has at most two zero elements, sparse storage might be inefficient.');
end
\% Now, all that remains is to check the last element:
if strcmp(D(end-2:end),',0\]')
    \% If this element was zero, we remove the last comma (if this isn't the
    \% empty matrix) and close with a bracket.
    if S(end) == '
        S(end) == []; \)
    end
    \( S = [S '\']; \)
else
    \% If this element is nonzero, we store everything after the last comma
    \% in \( S \), and we also store the coordinates of the element \( (N,N) \) into
    \% \( r \) and \( c \).
    \( S = [S \ D(cs(end)+1:end)]; \)
    \( r = [r \ N]; \)
    \( c = [c \ N]; \)
end
B.6.4 addPar.m

Replaces every occurrence of \( x_i \) with \( x(i) \) in a given string, where \( i \in [1, N] \).

function str = addPar(str,N)
\% replace every occurrence of \( x(i) \)
\% note that we do this backwards to avoid \( x10 \) being
\% replaced with \( x(1)0 \)
for i=N:-1:1
    \% is = num2str(i);
    is = ['x' is];
    xpis = ['x(' is ')'];
    str = strrep(str,is,xpis);
end