Implementation of Saddle-Point Minimal Residual Solvers

Nicholas Hu*

April 30, 2019

1 Introduction

In this thesis, we describe the implementation of the Saddle-Point Minimal Residual (SPMR) solvers developed by Estrin and Greif [3]. These solvers compose a family of iterative methods for the solution of large and sparse *saddle-point systems*: linear systems of the form

$$\begin{bmatrix} A & G_1^T \\ G_2 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} f \\ g \end{bmatrix}, \tag{1}$$

where $A \in \mathbb{R}^{n\times n}$, $G_1, G_2 \in \mathbb{R}^{m\times n}$, $f \in \mathbb{R}^n$, and $g \in \mathbb{R}^m$ (with $n > m$).

The term "saddle-point" originates from the equality-constrained quadratic programming problem

$$\begin{aligned} \min \quad & \frac{1}{2}x^T Ax - f^T x \\ \text{subject to} \quad & Bx = g, \end{aligned}$$

where $A \in \mathbb{R}^{n\times n}$ is symmetric positive-definite and $B \in \mathbb{R}^{m\times n}$ has full rank. The Lagrangian of this problem is $\mathcal{L}(x, y) = (\frac{1}{2}x^T Ax - f^T x) - y^T(g - Bx)$, where y represents a vector of Lagrange multipliers and the critical points of the Lagrangian are the solutions to

$$\nabla\mathcal{L}(x, y) = \begin{bmatrix} \nabla_x \mathcal{L}(x, y) \\ \nabla_y \mathcal{L}(x, y) \end{bmatrix} = \begin{bmatrix} Ax - f + B^T y \\ Bx - g \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix},$$

which is precisely (1) with $G_1 = G_2 = B$. These critical points are in fact *saddle points* of $\mathcal{L}$, since its Hessian

$$H_{\mathcal{L}}(x, y) = \begin{bmatrix} A & B^T \\ B & 0 \end{bmatrix}$$

is indefinite, which can be seen by considering the congruence transformation

$$\begin{bmatrix} I & 0 \\ -BA^{-1} & I \end{bmatrix} \begin{bmatrix} A & B^T \\ B & 0 \end{bmatrix} \begin{bmatrix} I & 0 \\ -BA^{-1} & I \end{bmatrix}^T = \begin{bmatrix} A & 0 \\ 0 & -BA^{-1}B^T \end{bmatrix}.$$

To wit, $-BA^{-1}B^T$ is symmetric *negative*-definite, whence $H_{\mathcal{L}}$ is indefinite by Sylvester's law of inertia.

*Department of Computer Science, The University of British Columbia, Vancouver BC, Canada, njhu@cs.ubc.ca.

Besides constrained optimization, saddle-point systems naturally arise in fields such as computational fluid dynamics, economics, electromagnetism, dynamical systems, and optimal control, as well as in solving PDEs with constraints. For more examples and properties of saddle-point systems, we refer the reader to the comprehensive survey of the subject by Benzi, Golub, and Liesen [1].

In the general formulation of (1), no assumptions are made about A (not even invertibility!) and we allow $G_1 \neq G_2$. The novel feature of SPMR is the exploitation of the block structure of the *saddle-point matrix*

$$\mathcal{K} = \begin{bmatrix} A & G_1^T \\ G_2 & 0 \end{bmatrix},$$

which generic iterative solvers (e.g., GMRES) tend to disregard. The off-diagonal blocks G_1^T and G_2 are bidiagonalized while A is diagonalized, allowing the SPMR iterates to be updated by short-term recurrences based on residual minimization or quasi-minimization, a process which we describe in Section 2. In Section 3, we will discuss the implementation of these methods in MATLAB and Julia.

2 The SPMR Family of Methods

2.1 Schur Complement and Nullspace Methods

The SPMR family of methods can be divided into two subfamilies depending on whether the leading block A is readily invertible.

If A is efficiently invertible and $\hat{x}$ is the solution to $A\hat{x} = f$, the substitution $x' := x - \hat{x}$ yields the system

$$\begin{bmatrix} A & G_1^T \\ G_2 & 0 \end{bmatrix} \begin{bmatrix} x' \\ y \end{bmatrix} = \begin{bmatrix} 0 \\ g - G_2\hat{x} \end{bmatrix}.$$

We may therefore assume in this case, without loss of generality, that the system is of the form

$$\begin{bmatrix} A & G_1^T \\ G_2 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0 \\ g \end{bmatrix}. \tag{2}$$

These methods bear the suffix "-SC", as they implicitly solve the system $(\mathcal{K}/A)y = g - G_2\hat{x}$, where

$$\mathcal{K}/A = -G_2A^{-1}G_1^T$$

denotes the *Schur complement* of A in $\mathcal{K}$. Indeed, it can be shown that they are mathematically equivalent to other known iterative methods (namely, USYMQR [7] and QMR [4]) applied to $S := -(\mathcal{K}/A)$. Even so, the SPMR family proves to be numerically superior to such methods when S is ill-conditioned, just as LSQR improves upon CG applied to the normal equations [3].

On the other hand, if we wish to avoid inverting A but are able to find a particular solution $\hat{x}$ to $G_2\hat{x} = g$, then the same substitution results in the system

$$\begin{bmatrix} A & G_1^T \\ G_2 & 0 \end{bmatrix} \begin{bmatrix} x' \\ y \end{bmatrix} = \begin{bmatrix} f - A\hat{x} \\ 0 \end{bmatrix}.$$

We may therefore assume that $g = 0$ in (1). For this subfamily of methods, we also assume that we have access to *nullspace bases* H_1, H_2 of G_1, G_2, respectively – that is, matrices $H_1, H_2 \in \mathbb{R}^{n\times(n-m)}$

such that $G_1 H_1 = G_2 H_2 = 0$*. Now if (1) has been reduced to the form

$$\begin{bmatrix} A & G_1^T \\ G_2 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} f \\ 0 \end{bmatrix},$$

the second equation $G_2 x = 0$ implies that $x = H_2 q$ for some $q \in \mathbb{R}^{n-m}$. Left-multiplying the first equation by H_1^T reveals that

$$\begin{aligned} H_1^T f &= H_1^T A x + H_1^T G_1^T y \\ &= H_1^T A H_2 q + (G_1 H_1)^T y \\ &= H_1^T A H_2 q. \end{aligned} \tag{3}$$

If A were invertible, this would be the Schur complement system implicitly solved when considering the saddle-point system

$$\begin{bmatrix} A & AH_2 \\ H_1^T A & 0 \end{bmatrix} \begin{bmatrix} p \\ q \end{bmatrix} = \begin{bmatrix} 0 \\ -H_1^T f \end{bmatrix} \tag{4}$$

(cf. equation (2)). More precisely, if we denote the saddle-point matrix in (4) by $\mathcal{K}_D$, then (3) is the equation $(\mathcal{K}_D/A)q = -H_1^T f$. The matrix

$$\mathcal{K}_D = \begin{bmatrix} A & AH_2 \\ H_1^T A & 0 \end{bmatrix}$$

is called the *inverse-free dual saddle-point matrix*, in reference to the *dual saddle-point matrix*

$$\mathcal{K}'_D = \begin{bmatrix} A^{-1} & H_2 \\ H_1^T & 0 \end{bmatrix},$$

which also satisfies $(\mathcal{K}'_D/A^{-1})q = -H_1^T f$ (henceforth, we will use "dual system" and "dual saddle-point matrix" exclusively in reference to (4) and $\mathcal{K}_D$) [1, 3].

Although A was not assumed to be invertible in this case, it turns out that this assumption is not necessary for this dual formulation to be meaningful. Under the assumption that the original saddle-point matrix $\mathcal{K}$ is nonsingular, Estrin and Greif show that there exists a solution to (4) such that $p \in \ker(G_2)$. Moreover, they show that x and y can be recovered by setting $x = -p$ and then solving the consistent overdetermined system $G_1^T y = f - Ax$ [3, Thm. 1]. In practice, this means that solvers based on (4) only compute x; the remaining block y must be recovered separately. However, in many applications, such as when y is a vector of Lagrange multipliers, y is not even needed, so this decoupling can be useful. Methods based on the dual system are given the suffix "-NS", indicating their usage of nullspace bases for the off-diagonal blocks.

2.2 Orthogonalization and Biorthogonalization Algorithms

A second dichotomy within the SPMR family lies in the nature of the bases with respect to which the off-diagonal blocks G_1^T and G_2 are bidiagonalized.

Before considering this distinction, let us recall the schema by which many other iterative methods – especially Krylov subspace methods – are derived. First, a matrix relation expressing the

*In Section 3.1.1, we will consider techniques for circumstances in which nullspace bases are not explicitly available.

desired reduction of the system matrix is stated. This relation describes the reduction that would be achieved if the iteration were to proceed until completion. Next, submatrices (typically rectangular) are defined so as to express the state of the iteration at an intermediate step. Finally, recurrence relations for vector iterates are determined from these submatrix relations, which dictate the updates performed in each iteration.

As an example, let us review the Full Orthogonalization Method (FOM), whose underlying iteration is the Arnoldi iteration [6]. The Arnoldi iteration is derived from the matrix relation

$$V^T A V = H,$$

where $A \in \mathbb{R}^{n\times n}$, $H \in \mathbb{R}^{n\times n}$ is upper Hessenberg, and $V \in \mathbb{R}^{n\times n}$ is orthogonal. Defining V_k as the submatrix consisting of the first k columns of V and $\tilde{H}_k$ as the upper-left $(k+1)\times k$ submatrix of H, we obtain

$$A V_k = V_{k+1}\tilde{H}_k, \qquad V_k^T V_k = I.$$

Labelling the columns of V and the entries of H in the usual manner, we determine therefrom that

$$h_{j+1,\,j}\, v_{j+1} = A v_j - \sum_{i=1}^{j} h_{ij} v_i, \qquad h_{ij} = v_i^T A v_j,$$

for $j = 1, 2, \ldots, k$.

The SPMR methods are more complex, as multiple blocks of $\mathcal{K}$ (for "-SC" methods) or $\mathcal{K}_D$ (for "-NS" methods) are simultaneously reduced. Nevertheless, their derivations conform to the same schema. For simplicity, we first consider the methods in the "-SC" subfamily, which are governed by the matrix relation

$$\begin{bmatrix} W & \\ & Z \end{bmatrix}^T \begin{bmatrix} A & G_1^T \\ G_2 & 0 \end{bmatrix} \begin{bmatrix} U & \\ & V \end{bmatrix} = \begin{bmatrix} J & B^T \\ C & 0 \end{bmatrix}, \tag{5}$$

where $J \in \mathbb{R}^{n\times n}$ is a diagonal matrix of signs (± 1) and $B, C \in \mathbb{R}^{m\times n}$ are lower bidiagonal. (The reason J is taken to be a matrix of signs rather than the identity matrix is that we wish to have $U = W$ and $V = Z$ when A is symmetric. The equation above therefore requires that J be congruent to A, which is impossible to satisfy if $J = I$ unless A is positive-definite, according to Sylvester's law of inertia. In this regard, the matrix J can "absorb the indefiniteness of A" [3].) Observe from (5) that U and W are A-biconjugate up to signs (i.e., $W^T A U = J$).

Notice that we have not yet imposed any requirements on the bases V and Z. It is here that the aforementioned dichotomy arises: *orthogonalization*-based methods require that V and Z both be orthogonal, whereas *biorthogonalization*-based methods require that V and Z be biorthogonal (i.e., $Z^T V = I$).

For methods in the "-NS" subfamily, we simply replace $\mathcal{K}$ by $\mathcal{K}_D$ in (5). This gives four different algorithms for the reduction of the system matrix, summarized in the table below.

	Reduction of $\mathcal{K}$	Reduction of $\mathcal{K}_D$
Orthogonalization	SIMBA-SC	SIMBA-NS
Biorthogonalization	SIMBO-SC	SIMBO-NS

Table 1: *Simultaneous bidiagonalization algorithms for SPMR methods. (The "A" in SIMBA stands for "A-biconjugacy"; the "O" in SIMBO stands for "biorthogonality".)*

We omit the statements of the submatrix and vector relations for the SIMBA/SIMBO algorithms, which can be found in the original paper [3, pp. 5–7, 15–16].

2.3 Residual Minimization and Residual Quasi-minimization

Having described the various ways in which the system matrix can be reduced, it remains to explain how iterative approximations to x and y (or p, when the dual system is being solved) are generated. It is common for iterative methods to determine successive iterates by solving a residual minimization/quasi-minimization problem over an affine space, and for this minimization problem to have a structure that is amenable to the development of recurrence relations for the iterates. An archetype of this methodology is GMRES, whose iterates lie in a translated Krylov subspace and are determined by solving a residual minimization problem (the "MRES" in "GMRES"), which turns out to be a Hessenberg least squares problem [6].

In keeping with praxis, the SPMR methods solve minimization/quasi-minimization problems over spaces spanned by the bases U and V. To state these problems, we define submatrices in accordance with the schema described in Section 2.2. Imitating the notation of Saad [6], let U_k denote the submatrix consisting of the first k columns of U (and likewise for V, W, and Z), and let B_k denote the upper-left $k \times k$ submatrix and $\tilde{B}_k$ the upper-left $(k+1) \times k$ submatrix of B (and likewise for C and J). Finally, we define

$$K_k := \begin{bmatrix} J_k & B_k^T \\ \tilde{C}_k & 0 \end{bmatrix}.$$

With this notation, the iterates are of the form $x_k = U_k \bar{x}_k$ and $y_k = V_k \bar{y}_k$ (i.e., $x_k \in \mathrm{im}(U_k)$ and $y_k \in \mathrm{im}(V_k)$). Again, we begin by illustrating the "-SC" subfamily of methods, for which the residual is

$$\begin{bmatrix} 0 \\ g \end{bmatrix} - \mathcal{K} \begin{bmatrix} x_k \\ y_k \end{bmatrix} = \begin{bmatrix} AU_k J_k & \\ & Z_{k+1} \end{bmatrix} \left(\begin{bmatrix} 0 \\ \delta_1 e_1 \end{bmatrix} - K_k \begin{bmatrix} \bar{x}_k \\ \bar{y}_k \end{bmatrix} \right), \tag{6}$$

where $\delta_1 = \|g\|$, which can be derived from (5). The vectors $\bar{x}_k$ and $\bar{y}_k$ are chosen to solve the residual quasi-minimization problem

$$\min_{\bar{x}, \bar{y}} \left\| \begin{bmatrix} 0 \\ \delta_1 e_1 \end{bmatrix} - K_k \begin{bmatrix} \bar{x} \\ \bar{y} \end{bmatrix} \right\|. \tag{7}$$

Observing that the upper block of the vector in (7) is $-(J_k \bar{x} + B_k^T \bar{y})$, we can equivalently solve

$$\min_{\bar{x}} \|\delta_1 e_1 - \tilde{C}_k \bar{x}\| \tag{8}$$

for $\bar{x}_k$ and recover $\bar{y}_k$ by taking $\bar{y}_k = -B_k^{-T} J_k \bar{x}_k$. With this approach, if $\bar{r}_k := \delta_1 e_1 - \tilde{C}_k \bar{x}_k$ is the quasi-minimal residual of (8), then

$$\begin{bmatrix} 0 \\ g \end{bmatrix} - \mathcal{K} \begin{bmatrix} x_k \\ y_k \end{bmatrix} = \begin{bmatrix} 0 \\ Z_{k+1} \bar{r}_k \end{bmatrix}.$$

Thus, if the method is *orthogonalization*-based, the norm of the quasi-minimal residual (i.e., $\|\bar{r}_k\|$) is in fact minimal, as the columns of Z_{k+1} are orthonormal! We therefore refer to this class of methods as 'true' SPMR methods, while biorthogonalization-based methods are named "SPQMR" instead (in SPQMR methods, the actual residual norm is $\|V_{k+1} \bar{r}_k\|$, which need not be equal to $\|\bar{r}_k\|$). We also observe that in either case, the norm of the actual residual is equal to that of $r_k := g - G_2 x_k$, the lower block of the vector in (6).

For the "-NS" subfamily of methods, we replace $\mathcal{K}$ by $\mathcal{K}_D$, g by $-H_1^T f$, and x_k, y_k by p_k, q_k in (6) (as per (4)). However, there is a catch: it is not apparent that the dual residual $r_k^{\mathrm{NS}} :=$

$-H_1^T f - H_1^T A p_k$ is in any way related to the original residual as defined by (4) – that is, to r_k in the equation

$$\begin{bmatrix} f \\ 0 \end{bmatrix} - \mathcal{K} \begin{bmatrix} x_k \\ y_k \end{bmatrix} = \begin{bmatrix} f - (Ax_k + G_1^T y_k) \\ 0 \end{bmatrix} =: \begin{bmatrix} r_k \\ 0 \end{bmatrix},$$

where $x_k = -p_k$ and y_k is the least-squares solution to $G_1^T y = f - Ax_k$ (it can be shown that p_k, and hence x_k, will lie in $\ker(G_2)$; see also Section 2.1 [3, Thm. 1]). Estrin and Greif show in this case that $\|r_k^{\mathrm{NS}}\| = |r_k|_{H_1 H_1^T}$, where $|\cdot|_{H_1 H_1^T}$ is the seminorm $|r|_{H_1 H_1^T} := (r^T (H_1 H_1^T) r)^{1/2}$ [3, Thm. 5]. But if $r \in \ker(G_1)$, then $|r|_{H_1 H_1^T} = 0$ if *and only if* $r = 0$, since $\ker(H_1^T) = \ker(G_1)^{\perp}$. In other words, $|\cdot|_{H_1 H_1^T}$ is a norm on $\ker(G_1)$. Since the r_k lie in $\ker(G_1)$ by definition, $r_k^{\mathrm{NS}} \to 0$ implies that $r_k \to 0$, so $\|r_k^{\mathrm{NS}}\|$ is a consistent measure of convergence.

The following table (cf. Table 1) lists the four methods of the SPMR family that we have derived.

	Schur complement method	Nullspace method
Residual minimization	SPMR-SC	SPMR-NS
Residual quasi-minimization	SPQMR-SC	SPQMR-NS

Table 2: *The SPMR family of methods.*

3 Implementations of the SPMR Solvers

In our implementations of the SPMR solvers, we distinguish between three phases of computation:

1. *Assembly* of the saddle-point matrix
2. *Iteration* of one of the four methods in the SPMR family
   (a) Iteration of the corresponding simultaneous bidiagonalization algorithm
   (b) Update of the approximate solution
   (c) Computation/estimation of the residual norm
3. *Recovery* of the results (e.g., x, y, residual norms/residual norm estimates)

The separation of the assembly and iteration phases allows multiple SPMR methods to be applied (and/or for a single method to be invoked multiple times with different arguments) using the same saddle-point matrix without redundancy in computation. While the recovery phase merely marks the termination of the iteration, we find it convenient to regard it as a separate step, as it involves the creation of a data structure to encapsulate the results of the iteration.

We begin by describing the MATLAB implementation since it is conceptually simpler than the Julia implementation but possesses the latter's key elements.

3.1 MATLAB Implementation

The following code excerpt, taken from `grcar_sc.m` in the SPMR-MATLAB toolbox (`https://github.com/nick-hu/SPMR-MATLAB`), reproduces one of the numerical experiments conducted by Estrin and Greif [3, p. 23].

```
1   n = 2000;
2   m = 1000;
3
4   A = gallery('grcar', n);
5   F = 100 * speye(m, m);
6   G1 = [F F];
7
8   [L1, U1, P1] = lu(A);
9   [L2, U2, P2] = lu(A');
10
11  f = transfunc_wrapper(@(x) U1 \ (L1 \ (P1 * x)), ...
12                        @(x) U2 \ (L2 \ (P2 * x)));
13
14  K = spmr_sc_matrix(f, G1, G1, n, m);
15
16  g = ones(m, 1);
17
18  result = spmr_sc(K, g, 'tol', 1e-10, 'maxit', 2*m);
```

Code Excerpt 1: *Usage of SPMR-SC in* MATLAB.

We will use this excerpt to explain the implementations of each of the three phases in turn. (Note that on lines 8 and 9, the user may select any factorization they wish for the problem at hand; it is not necessary to perform an LU factorization.)

3.1.1 Assembly of the Saddle-point Matrix

The functions `spmr_sc_matrix` and `spmr_ns_matrix` assemble structs corresponding to the two types of saddle-point matrices described in Section 2.1 (namely, $\mathcal{K}$ and $\mathcal{K}_D$ respectively). Their prototypes are

```
spmr_sc_matrix(A, G1, G2, n, m)
spmr_ns_matrix(A, H1, H2, n, m, l)
```

where the parameters have the same meanings as in equations (2) and (4). The additional parameter `l` of `spmr_ns_matrix` is the size of the nullspace bases H_1 and H_2, which is $n - m$ when explicit nullspace bases are used. However, as Estrin and Greif explain, it is also possible to use *orthogonal projectors* onto $\ker(G_i)$ in place of the H_i when explicit bases are unavailable or unwanted (because they are expensive to compute and/or dense) [3, pp. 12–13]. In this case, the size of the 'nullspace bases' is effectively n, which is supplied as `l` to `spmr_ns_matrix`.

Although `n` and `m` could conceivably be inferred from the sizes of `A`, `G1`/`H1`, and `G2`/`H2`, we allow function handles for matrix-vector products/linear system solves to be passed in place of any of the matrices. Specifically, if `f` is a function handle given in place of `G1`, we require that `f(x, 1)` return $G_1 x$ and that `f(x, 2)` return $G_1^T x$, and likewise for `H1`, `G2`, and `H2`. If given in place of `A`, we require that `f(x, 1)` return $A^{-1}x$ for `spmr_sc_matrix` but Ax for `spmr_ns_matrix`, and likewise for `f(x, 2)` with A^T (note that Schur complement methods solve systems with A, whereas nullspace methods compute matrix-vector products with A). With all these dimensions provided, it is straightforward to verify that the matrices are of the appropriate sizes and that the functions return vectors of the appropriate lengths before the iteration commences.

In the MATLAB implementation, `spmr_sc_matrix` and `spmr_ns_matrix` simply amalgamate the provided arguments into a struct with fields identical to the parameters of the functions. We will see later that the Julia analogues of these functions have more complicated behaviour since they serve as *constructors* for saddle-point matrix objects.

The usage of `spmr_sc_matrix` is illustrated by line 14 of Code Excerpt 1, wherein `G1` and `G2` are both equal to the matrix defined on line 6 and the function handle created on line 11 is provided in place of `A`. The helper function `transfunc_wrapper` is a higher-order function h defined such that $(h(f_1, f_2))(\cdot, i) = f_i$ for $i \in \{1, 2\}$.

3.1.2 Iteration of the SPMR Method

Once a saddle-point matrix has been created, an appropriate method of the SPMR family may be applied using this matrix, given a right-hand side vector f or g. Naturally, there is a function for each of the four SPMR methods in Table 2:

```
spmr_sc(K, g[, 'tol', tol][, 'maxit', maxit][, 'precond', M])
spqmr_sc(K, g[, 'tol', tol][, 'maxit', maxit][, 'precond', M])
spmr_ns(KD, f[, 'tol', tol][, 'maxit', maxit][, 'precond', M])
spqmr_ns(KD, f[, 'tol', tol][, 'maxit', maxit][, 'precond', M])
```

where we have demarcated optional parameters by brackets (as in extended Backus-Naur form). As their names suggest, `K` is a struct assembled by `spmr_sc_matrix` and `KD` is a struct assembled by `spmr_ns_matrix` (see Section 3.1.1). The syntax of an invocation of `spmr_sc` is shown in line 18 of Code Excerpt 1.

The optional parameter `tol` specifies a relative residual norm tolerance and `maxit` sets a threshold on the maximum number of iterations performed. In addition, it is possible to incorporate a right preconditioner $\mathcal{P}$ of the form

$$\begin{bmatrix} I_n & \\ & \mathcal{M} \end{bmatrix}$$

in any of the methods for solving (2) or (4) [3, pp. 18–21]. If $\mathcal{M}$ is symmetric positive definite (so that its associated bilinear form defines an inner product), it suffices to alter SIMBA and SIMBO such that V and Z are $\mathcal{M}^{-1}$-orthogonal and $\mathcal{M}^{-1}$-biorthogonal, respectively. This can be achieved by furnishing the optional argument `M` as a matrix or function handle. Otherwise, we must essentially replace $\mathcal{K}$ and $\mathcal{K}_D$ by $\mathcal{K}\mathcal{P}^{-1}$ and $\mathcal{K}_D\mathcal{P}^{-1}$, respectively. For this, we can make use of the functionality described in Section 3.1.1 (e.g., to provide a function handle in place of $G_1^T\mathcal{M}^{-1}$).

We also note that a cheap computation of the residual norm at each step is only possible when using the 'true' SPMR methods; for SPQMR methods, we can only obtain an upper bound (cf. residual norm estimation in GMRES vs. DQGMRES [6, pp. 178, 182]) [3, pp. 11, 17]. When this upper bound falls below the specified tolerance, $\|r_k\|$ or $\|r_k^{\mathrm{NS}}\|$ is computed exactly (which involves computing matrix-vector products). If this quantity is also below the specified tolerance, the iteration is terminated with success; otherwise, it is resumed.

While the MATLAB implementations of these methods are mostly straightforward translations of the algorithms given by Estrin and Greif, it is worth mentioning that some structural redundancies in the prototype implementations have been eliminated by method extraction. For instance, conditional blocks that either compute `K.G1 * x` or `K.G1(x, 1)` depending on the type of `K.G1` have been replaced by calls of the form `mul(K.G1, x)`. Although the conditionals are still present in the extracted methods, we will see in the Julia implementation that they can be dispensed with altogether by using polymorphism.

3.1.3 Recovery of the Results

Upon termination, the functions of Section 3.1.2 return a struct with fields `x`, (`y` for Schur complement methods,) the iteration count `iter`, a vector `resvec` of relative residual norms or estimates thereof, and a termination flag `flag`. (This struct is `result` in line 18 of Code Excerpt 1.) The

flag is an element of the enumerated type `SpmrFlag` that describes the circumstance in which termination occurred and is one of:

- `CONVERGED`, indicating that the relative residual norm fell below the prescribed tolerance (within the maximum number of iterations)
- `MAXIT_EXCEEDED`, indicating that the maximum number of iterations was performed (without convergence being achieved)
- `OTHER`, indicating that some computed quantity became too small

3.2 Julia Implementation

In addition to our MATLAB implementation, we implemented the SPMR family of methods in Julia, an emerging programming language designed for high-performance numerical and scientific computing [2]. While strongly influenced by MATLAB, we stress that Julia is not merely an open-source clone of the former. In particular, Julia possesses multiple dispatch and a sophisticated type system with features such as parametric polymorphism. These features, combined with its just-in-time compilation, allow it to rival the performance of statically-typed languages such as C while retaining the clarity of high-level languages such as Python. Metaprogramming with Lisp-style macros (operating at the level of abstract syntax trees) and code generation is also possible in Julia. All the aforementioned features are central to our implementation and render it more understandable, flexible, maintainable, and often more efficient than the MATLAB version.

The following code excerpt is adapted from `test/grcar_sc.jl` in the Julia SPMR package (`https://github.com/nick-hu/SPMR`) and is the Julia equivalent of Code Excerpt 1.

```
using LinearAlgebra, SparseArrays
using SPMR, LinearMaps

include("grcar.jl")

n, m = 2000, 1000

A = grcar(n)
F = sparse(100I, m, m)
G₁ = [F F]

K = SpmrScMatrix(A, G₁', G₁)

g = ones(m)

result = spmr_sc(K, g, tol=1e-10, maxit=2m)
```

Code Excerpt 2: *Usage of SPMR-SC in Julia.*

The included file `test/grcar.jl` contains the function `grcar`, which reproduces the effect of MATLAB's `gallery('grcar', n)`. The SPMR package depends on and is interoperable with the LinearMaps.jl package (`https://github.com/Jutho/LinearMaps.jl`), which allows Julia functions (for black-box matrix-vector products) to behave as linear maps (matrices).

As our implementation utilizes several advanced capabilities of Julia, we assume that the reader is familiar with the language (whose documentation can be found at `https://docs.julialang.org/en/v1/`). In particular, multiple dispatch, parametric polymorphism, and metaprogramming are used extensively.

To exploit the functionality of the LinearMaps.jl package, we declare the following types:

```
const FloatOperator = Union{AbstractMatrix{Float64}, LinearMap{Float64}}
const FloatInvOperator = Union{Factorization{Float64}, InvLinearMap{Float64},
                               UniformScaling}
const RealOperator = Union{AbstractMatrix{<:Real}, LinearMap{<:Real}}
const RealInvOperator = Union{Factorization{<:Real}, InvLinearMap{<:Real},
                              UniformScaling}
```

where `InvLinearMap` is a parametric type representing the inverse of a linear map. Internally, `InvLinearMap` is simply

```
struct InvLinearMap{T}
    map::LinearMap{T}
end
```

and left-divisions by an `InvLinearMap`, `A`, are interpreted as left-multiplications by `A.map`. In summary, types that end in "`Operator`" act by left-multiplication, whereas types that end in "`InvOperator`" act by left-division.

3.2.1 Assembly of the Saddle-point Matrix

As in the MATLAB implementation, the Julia implementation has two types of structures for representing the two types of saddle-point matrices. Internally, the major differences are that the "-SC" matrix structs store *factorizations* of A and A^T (when possible) and that size information no longer needs to be stored separately as this is specified at instantiation for `LinearMaps`. In Julia, these types of structures are indeed *types* in the strict sense of the word, and are declared as follows:

```
abstract type SpmrMatrix end

struct SpmrScMatrix{T<:FloatInvOperator, U<:FloatInvOperator,
                    V<:FloatOperator, W<:FloatOperator} <: SpmrMatrix
    A::T
    A^T::U
    G_1^T::V
    G_2::W

    # [Inner constructor omitted]
end

struct SpmrNsMatrix{T<:FloatOperator,
                    U<:FloatOperator, V<:FloatOperator} <: SpmrMatrix
    A::T
    H_1::U
    H_2::V

    m::Int

    # [Inner constructor omitted]
end
```

Note that m must still be stored in `SpmrNsMatrix`, as A, H_1, and H_2 could all be $n \times n$ (see Section 3.1.1).

There are multiple (outer) constructors for each of these types, and their delegation relationships are best understood by considering the diagrams below (the names of some types have been replaced by their acronyms for brevity).

The inner constructors are responsible for enforcing type invariants, such as the compatibility of the block sizes. Observe that on line 12 of Code Excerpt 2, constructor #4 in Fig. 1 is invoked.

```
1  SSM(A::T, A^T::U, G_1^T::V, G_2::W) where {T<:FIO, U<:FIO, V<:FO, W<:FO}
2  SSM(A::T, G_1^T::V, G_2::W) where {T<:ILM{Float64}, V<:RO, W<:RO}
3  SSM(A::T, G_1^T::V, G_2::W) where {T<:AM{Float64}, V<:FO, W<:FO}
4  SSM(A::T, G_1^T::V, G_2::W) where {T<:AM{<:Real}, V<:RO, W<:RO}
5  SSM(K::AM{<:Real}, n::Int)
```

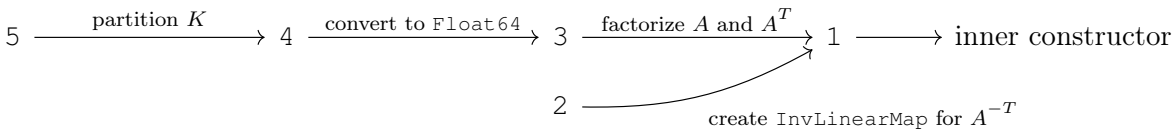

Figure 1: *Delegation diagram for* `SpmrScMatrix` *outer constructors.*

```
1  SNM(A::T, H_1::U, H_2::V, m::Int) where {T<:FO, U<:FO, V<:FO}
2  SNM(A::T, H_1::U, H_2::V, m::Int) where {T<:RO, U<:RO, V<:RO}
```

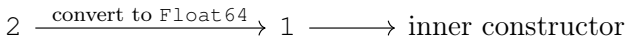

Figure 2: *Delegation diagram for* `SpmrNsMatrix` *outer constructors.*

At the factorization step in Fig. 1, a convenient factorization is computed according to the structure and properties of the matrix. For instance, if A is positive-definite, a Cholesky factorization is performed; if A is sparse and symmetric, an LDL^T factorization is performed; and so on. This is done by Julia's built-in `factorize` function, which is a black-box routine that returns an object of type `Factorization`. (In particular, a `Factorization{Float64}` is a `FloatInvOperator`, so the constructor call is subsequently dispatched to constructor #1 in Fig. 1.)

3.2.2 Iteration of the SPMR Method

Simultaneous bidiagonalization. In our implementation of the SPMR iterations, we make use of another sophisticated abstraction afforded by Julia: *iteration abstraction*. Specifically, the bidiagonalization algorithms (SIMBA/SIMBO) are encapsulated by *iterator* objects, and the generation of vectors by these algorithms is conceptually regarded as 'traversal' of a saddle-point matrix. Thus, SIMBA-SC and SIMBO-SC are two ways of 'traversing' an `SpmrScMatrix`, while SIMBA-NS and SIMBO-NS are two ways of 'traversing' an `SpmrNsMatrix`. Although Julia does not possess interfaces proper, it is possible to use its generic `iterate` function to implement a simplified variant of the well-known Iterator design pattern (in fact, this variant is precisely the 'minimal interface' variant proposed by Gamma et al.) [5].

The iterator types for "-SC" matrices are termed `SimbaScIterator` and `SimboScIterator`, and similarly for "-NS" matrices. The objects returned by the iterators, which we call "iterates", are of type `SpmrScIterate` or `SpmrNsIterate` according as the iterator is an "-SC" iterator or an "-NS" iterator. The iterates themselves are structs containing both scalars and vectors generated by the biorthogonalization algorithms. It is important to note that the iterates do not contain any state information, as this is entirely encapsulated by the iterator. Internally, the iterator maintains a reference to the saddle-point matrix and to the preconditioner being used (if there is one), and stores the current iterate (from which the next is computed) as well as the initial iterate so that the iteration may be restarted.

Concretely speaking, if `K` is an `SpmrScMatrix` as in line 12 of Code Excerpt 2, then the corresponding `SpmrScIterator` is constructed by a call of the form

```
simba_sc(K, b, c, precond=P)
```

within the function `spmr_sc`, which would then iterate over the iterator `SSI` as

```
for SI in SSI
    # [Do something with SI.u, SI.v, SI.w, SI.z, etc.]
end
```

Yet another abstraction presents itself, given the structural similarities between SIMBA-SC and SIMBO-SC (and between SIMBA-NS and SIMBO-NS): the metaprogrammatic generation of the code for the iterators themselves! For instance, the code that generates the iterator types reads

```
mutable struct $iterator_type
    K::$matrix_type
    SI::$iterate_type

    SI₀::$iterate_type  # Remember SI₀ so that we can reiterate

    M::FloatInvOperator  # Preconditioner
end
```

We also use macros to abstract the operations common to both "-SC" and "-NS" methods. As an example, consider the A-biconjugacy between U and W described in Section 2.2, which is present in both SIMBA and SIMBO. This property is imposed by the following sequence of operations:

$$\begin{aligned} &\xi_{k+1} \leftarrow \hat{u}_{k+1} \cdot w_{k+1} \\ &\alpha_{k+1} \leftarrow |\xi_{k+1}|^{1/2} && \gamma_{k+1} \leftarrow \alpha_{k+1} \\ &u_{k+1} \leftarrow \operatorname{sign}(\xi_{k+1}) u_{k+1}/\alpha_{k+1} && w_{k+1} \leftarrow \operatorname{sign}(\xi_{k+1}) w_{k+1}/\gamma_{k+1} \end{aligned}$$

(the meanings of α, γ, ξ, and $\hat{u}$ are irrelevant to the present discussion). The macro for these operations is

```
macro conjugate!(u, w, α, γ, ξ, û, n)
    return quote
        $ξ = $û · $w
        $α = $γ = sqrt(abs($ξ))

        @scal_signinv!($u, $α, $ξ, $n)
        @scal_signinv!($w, $γ, $ξ, $n)
    end |> esc
end
```

which in turn depends on the macro `scal_signinv!`, defined by

```
macro scal_signinv!(v, α, ξ, n)
    return quote
        BLAS.scal!($n, flipsign(inv($α), $ξ), $v, 1)
    end |> esc
end
```

This second macro performs the operation $v \leftarrow \operatorname{sign}(\xi) v/\alpha$ for $v \in \mathbb{R}^n$.

Particularities of the bidiagonalization algorithms (e.g., initialization steps) are dealt with by quoted expression interpolation. These quoted expressions are stored in a global constant dictionary.

Returning to the point raised in Section 3.1.2 about type-based conditionals – which are generally viewed as poor design but are inevitable in MATLAB – we see that such a construct is unneeded in the Julia implementation owing to multiple dispatch. To wit, the Julia equivalent of the call `mul(K.G1, x)` dispatches on the type of `K.G1`, which is by definition a subtype of `FloatOperator`, i.e., one of `AbstractMatrix{Float64}` or `LinearMap{Float64}`.

The main iteration. The implementation of the rest of the SPMR iteration likewise exploits the homology of the SPMR family. In particular, we define macros for both the initialization and the updating of x (p), y, and the (implicit) QR decomposition used to solve (8). As an illustration, Code Excerpt 3 alone generates both the SPMR-SC and SPQMR-SC methods.

```
function $func(K::SpmrScMatrix, g::AbstractVector{<:Real};
               tol::Float64=1e-6, maxit::Int=10, precond::RealInvOperator=I)
    n, m = block_sizes(K)

    SSI, SI₀ = $bidiag_func(K, g, g, precond=precond)

    @init_qr(SI₀)
    @init_x!(x, SI₀, n)
    @init_y!(y, SI₀, m)

    if abs(SI₀.ξ) < eps()
        return SpmrScResult(x, y, OTHER, 0, Float64[], precond)
    end

    resvec = Vector{Float64}(undef, min(m, maxit))

    $(iteration_quotes[func][:init])

    for (k, SI) in enumerate(SSI)
        if k > maxit
            return SpmrScResult(x, y, MAXIT_EXCEEDED, maxit, resvec[1:maxit],
                                precond)
        elseif abs(SI.ξ) < eps()
            return SpmrScResult(x, y, OTHER, k-1, resvec[1:k-1], precond)
        end

        @update_qr!(Ω, SI)
        @update_x!(x, SI, n)
        @update_y!(y, SI, m)

        $(iteration_quotes[func][:compute_res])
    end

    return SpmrScResult(x, y, MAXIT_EXCEEDED, m, resvec, precond)
end
```

Code Excerpt 3: *Generation of the SPMR-SC and SPQMR-SC methods in Julia.*

3.2.3 Recovery of the Results

From Code Excerpt 3, we also see how the result structs are constructed. In this respect, there is no significant difference between the MATLAB and Julia implementations, save for the fact that the `SpmrScResult` constructor performs the final $\mathcal{M}$-solve required when a preconditioner is being used (see Section 3.1.2).

4 Extensions and Concluding Remarks

Our implementations of the SPMR methods are efficient and extensible and adhere to core principles of software design. Several numerical experiments conducted by Estrin and Greif were successfully replicated and appear as examples accompanying the MATLAB and Julia code (in fact, Code Excerpts

1 and 2 perform one of these experiments) [3, pp. 23–24, 29]. Typically, both the MATLAB and Julia code ran 60–90 times faster than the prototype code, although the improvement was slightly less significant for Julia when `LinearMaps` were used. Cursory profiling of the code in this case revealed that a substantial amount of time was expended on type inference, suggesting that a native reimplementation of the functionality of `LinearMap` in SPMR may remediate the latter's performance.

Comprehensive user-friendly documentation was also produced for both implementations, which allows the packages to be easily extended. As an example, we implemented an inner-outer iterative method based on SPMR-SC and GMRES, which we called SPGMR, using the SPMR-MATLAB package. The essence of SPGMR (Saddle-Point Generalized Minimum Residual) is the application of GMRES to (1) with a preconditioner of the form

$$\mathcal{P} = \begin{bmatrix} \mathcal{G} & G_1^T \\ G_2 & 0 \end{bmatrix},$$

where $\mathcal{G}$ is an approximation to A and the $\mathcal{P}$-solves are performed using SPMR-SC (i.e., *inside* of the GMRES iteration). We may thereby apply SPMR-SC to a saddle-point matrix wherein the leading block is more efficiently invertible than A itself. For instance, $\mathcal{G}$ could be a lower triangular part or an incomplete factorization of A.

The code for SPGMR (in MATLAB) is simply

```
P = spmr_sc_matrix(G_func, G1, G2, n, m);
M_func = @(x) spgmr_inner(P, x, maxit);

[x, conv_flag, relres, iter, resvec] = gmres(K, [f; g], [], 1e-10, n+m, M_func);
```

where `G_func` is a function handle for $\mathcal{G}^{-1}$ and `spgmr_inner` is a function applying SPMR-SC with an arbitrary right-hand side `x` using the reduction described in Section 2.1. Specifically, `spgmr_inner` is

```
function sol = spgmr_inner(K, rhs, maxit)
    x_hat   = ldiv(K.A, rhs(1:K.n));
    g       = rhs(K.n + (1:K.m)) - K.G2 * x_hat;

    % The tolerance must be scaled as it is *relative*
    tol     = norm(rhs) / norm(g) * 1e-10;

    result  = spmr_sc(K, g, 'tol', tol, 'maxit', maxit);
    sol     = [result.x + x_hat; result.y];
end
```

Further investigation is required to determine the behaviour of SPGMR, which we may perform in the future.

The MATLAB and Julia packages for the SPMR methods may be found at `https://github.com/nick-hu/SPMR-MATLAB` and `https://github.com/nick-hu/SPMR`, respectively.

References

[1] M. Benzi, G. H. Golub, and J. Liesen. "Numerical solution of saddle point problems". In: *Acta Numerica* 14 (2005), pp. 1–137.

[2] J. Bezanson et al. "Julia: A Fresh Approach to Numerical Computing". In: *SIAM Review* 59.1 (2017), pp. 65–98.

[3] R. Estrin and C. Greif. "SPMR: A Family of Saddle-Point Minimum Residual Solvers". In: *SIAM Journal on Scientific Computing* 40.3 (2018), A1884–A1914.

[4] R. W. Freund and N. M. Nachtigal. "QMR: a quasi-minimal residual method for non-Hermitian linear systems". In: *Numerische Mathematik* 60.1 (1991), pp. 315–339.

[5] E. Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[6] Y. Saad. *Iterative Methods for Sparse Linear Systems*. 2nd ed. Society for Industrial and Applied Mathematics, 2003.

[7] M. Saunders, H. Simon, and E. Yip. "Two Conjugate-Gradient-Type Methods for Unsymmetric Linear Equations". In: *SIAM Journal on Numerical Analysis* 25.4 (1988), pp. 927–940.