Sweeping Algorithms for Five-Point Stencils and Banded Matrices*

Technical Memorandum ANL/MCS-TM-165, June 1992

Man Kam Kwong

Mathematics and Computer Science Division

Argonne National Laboratory

Argonne, IL 60439-4844

E-mail: `kwong@mcs.anl.gov`

Abstract

We record MATLAB experiments implementing the sweeping algorithms we proposed recently to solve five-point stencils arising from the discretization of partial differential equations, notably the Ginzburg-Landau equations from the theory of superconductivity. Algorithms tested include two-direction, multistage, and partial sweeping.

*This work was supported by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38.

1 Introduction

In [3] we proposed a method of sweeping for solving five-point stencils. Our work is motivated by our numerical study of the Ginzburg-Landau equations that arise in the theory of superconductivity. Nevertheless, the sweeping method is applicable to many generic five-point stencils derived from discretizing other partial differential equations. Particularly promising is the fact that a substantial portion of each algorithm is parallelizable.

In this article, we describe our experience in implementing the proposed methods. Part of the work was done with the help of Shirin Bar-Sela, who was a participant in the spring 1992 Student Research Participation Program at Argonne National Laboratory and who is currently an undergraduate at the University of Houston, Downtown.

At the preliminary stage of investigation, we chose MATLAB as our environment for experimenting, because of its convenient language features rendering program development simpler than in Fortran or C. After being successfully tested, a program will be translated into Fortran to give a faster version. Full exploitation of parallelism will then be the next step.

In Section 2, we give the definition of a regular and a Dirichlet five-point stencil, together with a list of our test problems. In Section 3, we describe the MATLAB environment in which the experiments were carried out. We have been testing a beta-version of the MATLAB 4 package. In the same section we describe some basic subroutines on which the sweeping programs are built. Section 4 covers one-stage, two-direction sweeping. Section 5 is devoted to a two-stage algorithm. In Section 6, we apply partial sweeping to a periodic Helmholtz operator. Section 7 discusses how the method of continuation can be used to invert stencils that are close to a stencil whose inverse has already been computed. Finally, in Section 8, we briefly discuss banded matrix inversion. We leave this discussion until last because the task of inverting such a matrix is conceptually the same of inverting a five-point stencil, and indeed is even simpler.

Our numerical experiments have shown that the sweeping method is easily implementable and is efficient for a large class of problems. Further work will focus on the feasibility of using parallelism to handle large scale problems on the larger supercomputers.

2 Problem Definition for Five-Point Stencils

The objects of our study are matrices of dimension $ny \times nx$. Typically, such a matrix arises from discretizing a partial differential equation. A matrix z may represent the function values of the dependent variable at the grid points of a rectangular domain. Another matrix, p, may represent given potential values at the same grid points. Thus, we sometimes refer to an element (more precisely, the location of the element) of p or of z as a point. To avoid long expressions, we shall use, whenever there is no risk of confusion, the same symbol p to denote a generic element $p(i, j)$ of the matrix. Points in the first and last rows or columns are called boundary points, while the rest are called interior points.

Each interior point has four neighbors, $p^{\leftarrow}$, $p^{\rightarrow}$, $p^{\downarrow}$, and $p^{\uparrow}$, understood in the usual sense. For boundary points, the interpretation of some (two if it is a corner point and one otherwise) of the neighbors is dictated by the boundary conditions of the original partial differential equation. In most of our experiments, we define the left-hand neighbor of a point in the first column to be the last point in the same row, the right-hand neighbor of a point in the last columns to be the first point in the same row, and so on. In other words, the relationship of neighborhood is continued across a boundary by wrapping around to the opposite boundary. As explained in [3], the periodic Ginzburg-Landau model that Du, Gunzburger, and Peterson described in [2] leads to a different definition for the lower neighbor of a point on the first row (and the upper neighbor of a point on the last row).

A five-point stencil is a linear combination between each point and its four neighbors:

$$S[p] = Cp + Lp^{\leftarrow} + Rp^{\rightarrow} + Dp^{\downarrow} + Up^{\uparrow}, \tag{2.1}$$

where C, L, D, and U are given matrices of dimension $ny \times nx$. A nonhomogeneous stencil equation is

$$S[z] = b, \tag{2.2}$$

where b is a given $ny \times nx$ matrix. It is a linear system of $ny \times nx$ equations, each of which is referred to as a point equation. Our objective is to solve for z, effectively and accurately.

We make the following assumptions:

1. All elements in L (D) not in the first column (row) and all elements in R (U) not in the last column (row) are nonzero.
2. The linear operator representing the five-point stencil is nonsingular. Indeed, in most cases of application, the operator is Hermitian and strictly positive or negative definite.

If the first column of L and the last column of R are both zero, we call the stencil a Dirichlet stencil. In that case, boundary points on the left have no effect on boundary points on the right and vice versa. We can, therefore, consider that points on the left boundary have *no* left neighbors and points on the right boundary have *no* right neighbors. Alternatively, we can consider that points on the left boundary have *imaginary* right neighbors that assume zero values. It is according to this second interpretation that we use the term Dirichlet. Note, however, that other types of separated boundary conditions on the partial differential equation, such as the Neumann and Robin types, also lead to Dirichlet stencils.

Test Problems

Problem I. The classical Dirichlet problem for the two-dimensional Poisson equation

$$\Delta u = f \tag{2.3}$$

on a rectangular domain, upon discretization over a uniform grid, gives the usual five-point stencil with coefficients $\{-4, 1, 1, 1, 1\}$ except at the boundary, where one or two of the appropriate coefficients becomes zero. More specifically, we pick for our first test problem

$$C = -4, \quad D = 1, \quad U = 1, \tag{2.4}$$

$$L(:,1) = 0, \quad L(:,2:nx) = 1, \tag{2.5}$$

$$R(:,nx) = 0, \quad R(:,1:nx-1) = 1. \tag{2.6}$$

For all our test problems, we take

$$b = 1. \tag{2.7}$$

Problem II. The generalized Helmholtz equation

$$\nabla(\rho(x)\nabla u) - \lambda u = f, \quad \lambda > 0, \tag{2.8}$$

gives rise to a stencil with nonconstant coefficients, whenever $\rho(x)$ is not a constant. We simulate this class of problem by perturbing the matrix C in Problem I. For $nx = ny = 16$, we take

$$C(i, j) = -4 - \frac{j-1}{16}. \tag{2.9}$$

The other coefficients remain the same.

Problem III. To get a non-Dirichlet test problem, we take the stencil of Problem II and change the first column of L and the last column of R to ones.

Problem IV. For a discussion of the derivation of the discrete Ginzburg-Landau operator that has motivated our study, see [3]. Other numerical approaches include the recent work of Du, Gunzburger, and Peterson (finite element method) [?] and Garner et al. (optimization techniques) [4]. We choose for our sample test problem the following parameters:

1. The vortex number $nv = 2$.
2. Domain size is $L_x \times L_y = 3\sqrt{3}$ units $\times$ $3\sqrt{3}$ units.
3. Number of grid points are equal for both axes, $N = nx = ny$.
4. The first component of the vector potential A is a linear function with $A(1,:) = 0$ and $A(N,:) = g$, where $g = 2\pi/L_x = 2.418399$.
5. The second component of the vector potential $B = 0$.

The coefficients for the stencil are found by using the formulas given in [3]. For reference, we list below those corresponding to $N = 40$:

1. $C(i, j) = -4$
2. $L = R$. Each row is constant, and the elements in each columns are as follows:

```
 1.0000                 1.0000 + 0.0079 i      0.9999 + 0.0157 i
 0.9997 + 0.0236 i      0.9995 + 0.0314 i      0.9992 + 0.0393 i
 0.9989 + 0.0471 i      0.9985 + 0.0550 i      0.9980 + 0.0628 i
 0.9975 + 0.0706 i      0.9969 + 0.0785 i      0.9963 + 0.0863 i
 0.9956 + 0.0941 i      0.9948 + 0.1019 i      0.9940 + 0.1097 i
 0.9931 + 0.1175 i      0.9921 + 0.1253 i      0.9911 + 0.1331 i
 0.9900 + 0.1409 i      0.9889 + 0.1487 i      0.9877 + 0.1564 i
 0.9864 + 0.1642 i      0.9851 + 0.1719 i      0.9837 + 0.1797 i
 0.9823 + 0.1874 i      0.9808 + 0.1951 i      0.9792 + 0.2028 i
 0.9776 + 0.2105 i      0.9759 + 0.2181 i      0.9742 + 0.2258 i
 0.9724 + 0.2334 i      0.9705 + 0.2411 i      0.9686 + 0.2487 i
 0.9666 + 0.2563 i      0.9646 + 0.2639 i      0.9625 + 0.2714 i
 0.9603 + 0.2790 i      0.9581 + 0.2865 i      0.9558 + 0.2940 i
 0.9535 + 0.3015 i
```

This list gives $L(1,j)$, $L(2,j)$, and $L(3,j)$ in the first row; $L(4,j)$, $L(5,j)$, $L(6,j)$ in the second row; an so on.

3. $D(i,j) = 1$ for all $i > 1, j \geq 1$, and the first row is given by the following:

```
 1.0000                 0.9511 + 0.3090 i      0.8090 + 0.5878 i
 0.5878 + 0.8090 i      0.3090 + 0.9511 i      0.0000 + 1.0000 i
-0.3090 + 0.9511 i     -0.5878 + 0.8090 i     -0.8090 + 0.5878 i
-0.9511 + 0.3090 i     -1.0000 + 0.0000 i     -0.9511 - 0.3090 i
-0.8090 - 0.5878 i     -0.5878 - 0.8090 i     -0.3090 - 0.9511 i
-0.0000 - 1.0000 i      0.3090 - 0.9511 i      0.5878 - 0.8090 i
 0.8090 - 0.5878 i      0.9511 - 0.3090 i      1.0000 - 0.0000 i
 0.9511 + 0.3090 i      0.8090 + 0.5878 i      0.5878 + 0.8090 i
 0.3090 + 0.9511 i      0.0000 + 1.0000 i     -0.3090 + 0.9511 i
-0.5878 + 0.8090 i     -0.8090 + 0.5878 i     -0.9511 + 0.3090 i
-1.0000 + 0.0000 i     -0.9511 - 0.3090 i     -0.8090 - 0.5878 i
-0.5878 - 0.8090 i     -0.3090 - 0.9511 i     -0.0000 - 1.0000 i
 0.3090 - 0.9511 i      0.5878 - 0.8090 i      0.8090 - 0.5878 i
 0.9511 - 0.3090 i
```

4. $U(i,j) = 1$ for all $i < N, j \geq 1$, and the last row is the conjugate of the first row of D.

3 MATLAB Environment and Some Basic Functions

We used for our experiments the Beta 3 Version of MATLAB 4. This is chosen over the previous release MATLAB 3.5 because of its better performance and enhanced graphics capabilities. Although the new graphics features are not involved in the part of the project we are reporting here, they are used in our overall study of the Ginzburg-Landau equations.

The coefficients C, L, R, D, and U for the various problems are stored in different data files to be loaded either via the `startup.m` file or via the `load` command. For those experiments described in Sections 4 and 5, we have only one set of coefficients to deal with each time. It is convenient to declare these coefficients as *global* variables, so that we do not have to pass them as arguments to function subroutines.

The declaration of global variables inside a function is a new feature of MATLAB 4. In previous versions, a global variable had to be declared at the highest interactive level. Thus, *all* function calls treated that variable as global. A lurking danger was that a user could unknowingly invoke some function that contained a variable with the same name, but was meant to be a local variable. The new rule in MATLAB 4 is that unless the variable is *also* declared to be global *inside* a function, the external declaration has no effect. We, however, prefer that the requirement for an external declaration be eliminated, since each function already has its own. Perhaps the purpose of having this requirement is to allow functions to share semi-global variables among themselves, but not with the workspace. We believe that this is a dangerous practice (a user may have forgotten that he has already used a global variable with the same name in a function he defined months ago) and should be discouraged. In principle, a function should share a global variable with another function through a common global variable in the workspace. Furthermore, functions having global variables should be special-purpose functions used only for a special project.

For the partial sweeping algorithm discussed in Section 6, the situation is different. It is no longer useful to declare the original coefficients as global variables, since we work with more than one sets of submatrices instead of the original matrices. All the functions described below will need to be rewritten to put the coefficients back into the input argument list.

3.1 `S.m` — stencil operator

```
%  y = S(p)    To compute the image of  p  under the
%                 stencil  S
%
%              C L R D U  should exist in workspace

function y=S(p)

global C L R D U

[ny nx]=size(p);

y = C.*p + L.*p(:,[nx,1:nx-1]) + R.*p(:,[2:nx,1]) + ...
    D.*p([ny,1:ny-1],:) + U.*p([2:ny,1],:);
```

The main use of this function is to determine how accurately an approximate solution za satisfies the given stencil equation. One can use the residue

`bb = b - S(za)`

to obtain a correction `zc` to the approximate solution, simply by solving the residue equation

`S(zc) = bb.`

The following is the modified version that includes the coefficients as input arguments (needed for the partial sweeping algorithm).

```
function y=S(C,L,R,D,U,p)

[ny nx]=size(p);

y = C.*p + L.*p(:,[nx,1:nx-1]) + R.*p(:,[2:nx,1]) + ...
    D.*p([ny,1:ny-1],:) + U.*p([2:ny,1],:);
```

Since modifications for the other functions in the rest of the section are similar and straightforward, they are omitted.

3.2 swp.m — basic column sweeping operator

```
%  y = swp(x,b,n1,n2)    Basic sweeping operation for
%                          Sx = b  from column  n1  to  n2
%                        If n2==0, sweep --> 1 column
%                              <          <--

function y=swp(x,b,n1,n2)
global C L R D U

[ny nx]=size(x);

y=x;
if n1<=n2
if n2==0, n2=n1; end
for ii=n1-1:n2-1
  if ii==1 yL=y(:,nx); else yL=y(:,ii-1); end
  y(:,ii+1) = (b(:,ii)-(C(:,ii).*y(:,ii)+L(:,ii).*yL+ ...
     D(:,ii).*y([ny,1:ny-1],ii)+ ...
     U(:,ii).*y([2:ny,1],ii)))./R(:,ii);
end
else
if n2<0, n2=n1; end
for ii=n1+1:-1:n2+1
  if ii==nx yR=y(:,1); else yR=y(:,ii+1); end
  y(:,ii-1) = (b(:,ii)-(C(:,ii).*y(:,ii)+R(:,ii).*yR+ ...
     D(:,ii).*y([ny,1:ny-1],ii)+ ...
     U(:,ii).*y([2:ny,1],ii))).lemma(:,ii);
end
end
```

We start with a matrix `x` that does not satisfy the stencil equation `S[z]=b`. The idea of the sweeping method is to alter the columns (rows) of `x` to satisfy as many of the point equations as possible. For column sweeping, there are two directions to choose from: towards the right or towards the left. In the former case, a column, say, column j, is altered so that all point equations pertaining to the column $j-1$ are satisfied. When sweeping to the left, equations pertaining to the column $j+1$ are used.

The function `swp` sweeps the matrix `x` by altering columns from `n1` to `n2`, inclusively, using `b` as the righthand side of the stencil. The direction of sweeping is determined by the relative sizes of the arguments `n1` and `n2`, except in the case when only one column is to be swept; then `n1` is equal to `n2`, and the default direction is to the right. For the case of sweeping only one column, we provide an alternative way to specify the direction: to the left if `n2 = 0` and to the right if `n2<0`.

Note that in the calculation of a new column in each sweeping step, the elements can be computed in parallel.

We point out a pitfall of numerical experimentation. Rounding errors, if not carefully controlled, can yield results contradictory to theoretical predictions. Sweeping is one of those procedures that has to be closely scrutinized. Suppose that one starts with a matrix `x` and sweeps it from column `n1` (towards the right) to column `n2` to get a new matrix `y`. If one now sweeps `y` from column `n2-2` back to `n1`, one should, theoretically, make no changes to the columns of `y`. However, when we experimented with the simple Laplacian stencil of size 40×40, and `x(i,j)=0` except `x(1,1)=1`, we found that the two matrices differ by 4.2×10^{58}. In fact, the magnitude of the elements in the last column of `x` is on the order of 10^{29}, an indication of the rate of growth by sweeping through 40 columns. Sweeping back from the right will magnify any error by an equally large factor.

Our next step is to build a more user-friendly version of the basic sweeping operation. The input format of a MATLAB function is rather flexible. The function can be called with fewer arguments than specified in its definition, and the data types of the arguments are not rigidly fixed by declarations in the subroutine, as in Fortran or C subroutines. This flexibility allows one to design functions that can assume default values whenever certain arguments are omitted from the input list. Unfortunately, this flexibility can also be a source of extremely subtle bugs, if used carelessly.

3.3 sw0 — a more flexible sweeping subroutine

```
% y = sw0(x,n1,n2)          sweep matrix  x
%     sw0(x,b,n1,n2)           from column n1 to n2 and
%     sw0(x,n1,n2,m1,m2)       from m1 to m2
%     sw0(x,b,n1,n2,m1,m2)
%
%           b               rhs, default 0 if omitted
%           n1 <= (>) n2    sweep to the right (left)
%           n2 == (<) 0     sweep 1 column right (left)

function y=sw0(x,b,n1,n2,m1,m2)
global C L R D U

if nargin==3      if max(size(b))==1
                     y = swp(x,0*x,b,n1);
                  else
                     y = swp(x,b,n1,0);
                  end
elseif nargin==4  y = swp(x,b,n1,n2);
elseif nargin==5  y = sw0(x,0*x,b,n1,n2,m1);
else              z = swp(x,b,n1,n2);
                  y = swp(z,b,m1,m2);
end
```

Many of the sweeping operations used in the algorithms involve simultaneous sweeping, mostly in two opposite directions on two separate portions of the same matrix. Also, in some cases, the sweepings are done on the homogeneous stencil, corresponding to taking the righthand side `b=0`. The function `sw0` is designed to cover these two situations in the most user-friendly way. If `b` is omitted in the argument list, the homogeneous stencil is assumed. If two column ranges are specified, then two sweepings will be done; otherwise, only one sweeping is needed. Obviously, `sw0` is based on `swp`.

In the MATLAB implementation, when two sweepings are requested, they are performed in serial. They should be performed simultaneously on a parallel computer.

3.4 err.m — stencil error for one or two columns

```
%  e = err(x,b,n1,n2)    compute error  Sx - b  for two
%      err(x,n1)         or one column; b = 0  (default)
%      err(x,n1,n2)

function e=err(x,b,n1,n2)
global C L R D U

if nargin==2
   n=b; b=0*x;
elseif nargin==3
   if max(size(b))==1, n=[b n1]; b=0*x; else n=n1; end
else
   n=[n1 n2];
end

[ny nx]=size(x);

e=[];
for ii=n
   if ii==1, xL=x(:,nx); else xL=x(:,ii-1); end
   if ii==nx, xR=x(:,1); else xR=x(:,ii+1); end
   e1=C(:,ii).*x(:,ii)+L(:,ii).*xL+R(:,ii).*xR+ ...
       D(:,ii).*x([ny,1:ny-1],ii)+ ...
       U(:,ii).*x([2:ny,1],ii)-b(:,ii);
   e = [e; e1];
end
```

After a sweeping operation on a matrix, some point equations, typically in one or two columns, still may not be satisfied. The function `err` computes the error `Sx-b`. The answer is given as a column vector of length `ny` or `2ny` according to whether one or two columns are involved. The function assumes a default homogeneous stencil if the input argument `b` is omitted.

4 Two-Direction Sweeping

We skip the simple one-direction sweeping, which works well in most cases for N up to about 12.

```
Program: Two-Direction Sweeping for Solving   S[z] = b

  Compute the rectifying matrix  RR :
       for  i  from  1  to  ny
         x(i,nx/2) = 1, x = 0  otherwise
           sweep  x  with the homogeneous stencil
         error vector from first and last column
           is added to  RR
       end
       repeat using  x(i,nx/2+1) = 1

  Construct a test solution  y :
       y = 0
       sweep  y  outward from the two columns in the
         center
       compute error vector  er  from first and last
         columns

  Construct the solution  z :
       − RR \ er  gives the two center columns of  z
       sweep outward to compute  z
```

We shall use the name *rectifying matrix* to refer to both the matrix RR and its inverse. The same holds for other rectifying matrices introduced later. The context will ensure that no confusion arises.

We found that sweeping outward was more suitable for general Dirichlet stencils. The main reason is that the solution of a Dirichlet stencil is more sensitive to adjustments made in the first and last columns than those made in the central columns. The rectifying matrix for computing the central columns is, therefore, better conditioned than that for computing the boundary columns. As will be explained in Section 6, a modification is called for if the sweeping is to be incorporated as a subroutine within a partial sweeping algorithm.

Our implementation splits the algorithm into two M-files, the first to find RR and the second to solve for z. The former is the more time-consuming part of the the algorithm. Once we have RR computed, only the second M-file is needed to solve different equations for the same stencil.

```
% arr.m
%
% Routine to compute the rectifying matrix RR
%
%      Input (from workspace) :-
%
%            C L R D U
%
%      Output :-
%
%            RR RRI x (last sweeping matrix)

RR=[];
for j=1:ny
  x=zeros(ny,nx); x(j,1)=1;
  x=sw0(x,2,nx/2);
  RR = [RR err(x,nx/2,nx/2+1)];
end

for j=1:ny
  x=zeros(ny,nx); x(j,nx)=1;
  x=sw0(x,nx-1,nx/2+1);
  RR = [RR err(x,nx/2,nx/2+1)];
end

RRI = inv(RR);
```

This subroutine consists simply of two loops to find the 2ny columns of RR. Notice that each sweeping cycle of x is performed completely independently of the others. Parallel computation is, therefore, the correct route to go.

The inverse of RR is computed once and for all if it is known beforehand that there is more than one equation of the same stencil to be solved. Otherwise, the command

```
rr = - RR \ er;
```

can be used in the second M-file. This command computes `rr` by using Gaussian elimination.

```
% asw.m
%
% Simple two-sided sweeping for solving  Sz = b
%
% Using module subroutines  sw0  and  err
%       and previously computed  RR
%
%   Input (defined in workspace):-
%
%       b C L R D U
%                RR  computed using arr.m
%
%   Output:
%                y   trial solution with y(:,1)=y(:,N)=0
%                er  errors in the 2 central columns in  y
%                rr  the 2 central columns in  z
%                z   solution

% Initial sweeping to get trial solution  y  and error  er

y=zeros(ny,nx);
y=sw0(y,b,nm-1,1,nm+2,nx);
er=err(y,b,1,nx);

% Back sweeping to compute solution z

rr=-RRI*er;         % OR  rr=-RR\er;
z=zeros(ny,nx);
z(:,nm)=rr(1:ny); z(:,nm+1)=rr(ny+1:2*ny);
z=sw0(z,b,nm-1,1,nm+2,nx);
```

Theoretically, these two subroutines are all that is needed to compute the solution of the stencil equation. A solution-refining routine to be described below can be used if the accuracy of the computed answer is not satisfactory.

Here are the results from some typical tests:

Problem	Grid Size	$\max z$	$\max\{\lvert b - Sz\rvert\}$	Flops 1	Flops 2
I	16×16	36.0	1.4922×10^{-8}	92817	32783
II	16×16	3.9540	1.5320×10^{-9}	92817	32603
III	16×16	4.7990	1.9568×10^{-9}	92817	32575
IV	16×16	3.7697	5.6694×10^{-10}	360785	139211
I	40×40	213.35	142.9461	1424361	408453
IV	40×40	69.37	37.7422	5737721	1689627

If one examines the full residue matrix rather than just its maximum, one should be more impressed because the errors are concentrated only on the two boundary columns; the residues for the interior columns are practically nil, a logical consequence of the sweeping operation. The average residue, or residue in the mean, is therefore much smaller than mres = $\max\{|b - Sz|\}$.

The last two columns in the table give the number of floating-point operations for the two subroutines, respectively. The significantly greater number of flops needed for Problem IV is due to the presence of complex number operations. The number of flops executed by a subroutine, however, is not proportional to the elapse time, which can be determined by using the MATLAB commands `tic` and `toc`. Even thought `arr` requires only about three times as many flops as `asw`, the execution time for `arr` is between 15 to 40 times as great as that of `asw`, depending on the current load of the machine. The subroutine `asw` ran extremely fast for small `nx` and `ny`.

Accuracy falls off as the size of the grid increases, as evidenced by the last two rows of the table. Errors accrue in the inversion of the matrix `RR`, which has very large entries, and in the unstable sweeping operation. One may be tempted to conclude that the solutions for the last two problems are even worse than the zero matrix, which gives a maximum residue of 1. Nevertheless, one must bear in mind that, for the zero matrix, there is a residue of 1 at every grid point, whereas the seemingly appalling residue of `z` appears only at the boundary points.

The solution `z` can be refined by using another subroutine that solves the residue equation. In Section 7, we shall see that the same technique can be used to obtain the inversion of nearby stencils.

```
% Subroutine to refine a computed solution to  Sz = b
%
% * If this is the first time the subroutine is called
% * after  z  is computed by using  asw.m, copy  b  and
% * z  to the variables  bb  and  zz  first  by using
% *
% *      bb = b;  zz = a;
% *
% * If more than one iteration is needed, run the subroutine
% * in a  "for"  statement
%
%
%   Input (defined in workspace):-
%
%       bb zz C L R D U RR (computed using arr.m)
%
%   Output:
%                 b   residue
%                 z   solution of residue equation
%                 zz  improved solution

b=bb-S(zz);
asw
zz=zz+z;
```

Applying this refinement subroutine just once to the problem in the first row of the previous table, we obtained an improved solution with mres $= 2.8422 \times 10^{-14}$. Examining the solution `z` shows that accuracy is really up to the last digit allowable by machine accuracy. Improving the solution to the problem in the second row gave mres $= 3.9968 \times 10^{-15}$. In both cases, more applications of the refinement subroutine do not lead to further improvement, as the limit of the machine accuracy is already reached.

For the Ginzburg-Landau operator with `N = 40`, one application of the refinement subroutine gave an improved solution with mres $= 1.54220 \times 10^{-8}$. A second iteration gave mres $= 1.24130 \times 10^{-12}$.

It is easy to incorporate the refinement subroutine into the second subroutine in the usual way so that the solution will be automatically refined either until a specified accuracy is attained or until no further improvements are possible.

If different equations are to be solved for the same stencil, it is worthwhile to perform the refinement on the rectifying matrix `RRI`, especially in the case when many refinement steps are needed, because the process can be speeded up if we use the newly improved `RRI` in the next iterative step of refinement. In theory, each column in `RRI` is the two center columns of the solution to the stencil equation in which the righthand side is zero at all but one appropriate point in the two boundary columns, at which the value is 1. By refining the solution to this system, we can obtain a refined column for `RRI`. The columns can be refined in parallel.

5 Two-Stage Sweeping for Dirichlet Stencils

The refinement procedure described in Section 4 cannot take care of rounding errors arising from the rather unstable sweeping process; an error can grow exponentially as the number of columns swept increases. A multistage algorithm breaks up the sweeping into stages, each over a small range of columns. The method is still a direct one, as opposed to an iterative scheme like the partial sweeping of the next section. There is more than one way to implement a multistage algorithm. We describe only one of the approaches we have tested.

We divide the `nx` columns into four groups: (1) columns 1 to `n1-1`, (2) `n1` to `nm`, (3) `nm+1` to `n2`, and (4) `n2+1` to `nx`. In general, the four groups need not be equal in size. When `nx` is a multiple of 4, division into four equal groups corresponds to choosing `n1 = nx/4 + 1`, `nm = nx/2`, and `n2 = 3nx/4`. For example, when `nx = 40`, we use `n1 =11`, `nm = 20`, and `n2 = 30`.

We shall make use of columns `n1` and `n2` as initial columns in the final step of backward sweeping. The rectifying matrix `BRR` is, therefore, defined to be that relating the values of the solution `z` in these two initial columns to the errors in the two center columns.

Since the two initial columns are not adjacent to each other, we cannot really start backward sweeping right away; we must have a means to determine columns `n1-1` and `n2+1`. The matrices `BI1` and `BI2` computed in the first part of the algorithm are used for that purpose. They are found by matching the errors obtained by sweeping the first (last) quarter of the grid with `n1-1` and `n1` (`n2+1` and `n2`) as initial columns.

The rectifying matrix `BRR` is then computed by sweeping the second and third quarters of the grid, inwards from columns `n1` and `n2`. Note that in this implementation of the algorithm, the second stage of sweeping depends on the outcome, namely, the matrices `BI1` and `BI2`, from the first stage, and hence the two stages must be performed in serial. A modification can be made to carry out the second stage in parallel without prior knowledge of `BI1` and `BI2`. The tradeoff is the inversion of a larger matrix of dimension $2\mathtt{nx} \times 2\mathtt{nx}$ in the second stage. If there are more stages and if parallel processing is available, substantial speedup can be achieved. The implementation we give here is analogous to the method of marching, in the language of shooting methods, and the modification we alluded to is analogous to multiple shooting. For references to these shooting techniques, see [1].

```
Program: Two-Stage Sweeping for Solving   S[z] = b

  Compute the first-stage rectifying matrices  BRR1  and
     BRR2  and the continuation matrices  BI1  BI2 :
       BI1  and  BI2 give columns  n1−1  and  n2+1
         when columns  n1  and  n2  are known

  Compute the second-stage rectifying matrix  BRR :
       this matrix relates the errors in the two
         center columns to the choice of initial
         sweeping values in columns  n1  and  n2

  Construct a first-stage test solution  y  that is error free
     in the first and last quarters :
       y = 0
       sweep outward from columns  n1  and  n2
       compute errors from first and last columns
       use  er, BRR1, and  BRR2  to determine columns
         n1  and  n2  of  y

  Construct the solution  z :
       sweep second and third quarters
       compute errors in the two center columns
       use  BRR  to determine columns  n1  and  n2
       use  BI1  and  BI2  to determine columns
         n1−1  and  n2+1
       sweep to compute  z
```

The construction of the solution `z` is also carried out in two stages. In the first stage, we use the matrices `BRR1` and `BRR2` to smooth out as much as possible the errors in the first and last quarters, rather than sweeping them and thus magnifying them all the way to the center columns.

As before, we split up the program into two subroutines; the first yields the rectifying matrices and the second the solution. If required, a third subroutine for refining the solution can be added just as in the previous section.

```
% brr.m
%
% Routine to compute the rectifying matrices  BRR  BI1  BI2
%   for the Two-stage Two-way Sweeping
%
%      Input (from workspace) :-
%
%            C L R D U nx ny n1 n2 nm
%
%      Output :-
%
%            BRRI BRR BRR1 BRR2 BI1 BI2
%            bi1 bi2 x (last sweeping matrix)

BRR1=[]; BRR2=[]; bi1=[]; bi2=[];
for j=1:ny
  x=zeros(ny,nx); x(j,n1-1)=1; x(j,n2+1)=1;
  x=sw0(x,n1-2,1,n2+2,nx);
  BRR1=[BRR1 err(x,1)];
  BRR2=[BRR2 err(x,nx)];
  x=zeros(ny,nx); x(j,n1)=1; x(j,n2)=1;
  x=sw0(x,n1-2,1,n2+2,nx);
  bi1=[bi1 err(x,1)];
  bi2=[bi2 err(x,nx)];
end
  BI1 = -BRR1\bi1;
  BI2 = -BRR2\bi2;

BRR=[];
for j=1:ny
  x=zeros(ny,nx); x(j,n1)=1; x(:,n1-1)=BI1(:,j);
  x=sw0(x,n1+1,nm);
  BRR = [BRR err(x,nm,nm+1)];
end
for j=1:ny
  x=zeros(ny,nx); x(j,n2)=1; x(:,n2+1)=BI2(:,j);
  x=sw0(x,n2-1,nm+1);
  BRR = [BRR err(x,nm,nm+1)];
  BRRI = inv(BRR);
end
```

The second subroutine solves the stencil for a given righthand side `b`:

```
% Two-stage Two-sided sweeping for solving  Sz = b
%
% Using module subroutines  sw0  and   err
%       and previously computed  BRRI  BRR  BI1  BI2
%
%   Input (defined in workspace):-
%
%       b C L R D U
%       BRR BRR1 BRR2 BI1 BI2  computed using brr.m
%
%   Output:
%                y  y2         trial solutions
%                er1 er2 er    sweeping errors
%                b2            residue
%                rr            correction for solution
%                z             solution

% First-stage sweeping to compute trial solution  y
%   in  1st  and  4th  quarters

y=zeros(ny,nx);
y=sw0(y,b,n1-2,1,n2+2,nx);
er1=err(y,b,1);  er2=err(y,b,nx);
y(:,n1-1)= -BRR1\er1;
y(:,n2+1)= -BRR2\er2;
y=sw0(y,b,n1-2,1,n2+2,nx);

% Compute residue

b2=b-S(y);

% Second-stage sweeping to get trial solution  y2
%   in  2nd  and  3nd  quarters

y2=zeros(ny,nx);
y2=sw0(y2,b2,n1+1,nm,n2-1,nm+1);
er=err(y2,b2,nx/2,nx/2+1);
                                        % to be continued ...
```

```
                                                    % ... continue
% Back sweeping to compute solution z

rr=-BRR\er;
z=zeros(ny,nx);
z(:,n1)=rr(1:ny); z(:,n2)=rr(ny+1:2*ny);
z(:,n1-1)=BI1*rr(1:ny); z(:,n2+1)=BI2*rr(ny+1:2*ny);
z=sw0(z,b2,n1-2,1,n2+2,nx);

% Combine with first trial solution  y

z = y + sw0(z,b2,n1+1,nm,n2-1,nm+1);
```

Test results are recorded in the following table.

Problem	Grid Size	$\max z$	$\max\{\lvert b - Sz\rvert\}$	Flops 1	Flops 2
I	16×16	36.0	7.3825×10^{-12}	127920	45591
II	16×16	3.9540	1.0534×10^{-12}	127920	45489
IV*	16×16	3.5734	4.1064×10^{-13}	546016	190469
I	40×40	210.00	2.7022×10^{-6}	1905064	543383
IV*	40×40	69.43	6.6991×10^{-7}	8363336	2256225
II	60×60	8.9116	7.4750×10^{-4}	6361240	1702789

* Since our two-stage algorithm applies only to Dirichlet stencils, Problem IV used in this testing has been modified accordingly.

As will be explained in Section 6, if this two-stage subroutine is used in a partial sweeping algorithm that employs two overlapping domain decompositions, the first-stage sweeping to find `y` can be omitted.

6 Partial Sweeping

Partial sweeping is an alternative to multistage sweeping for very large grid sizes. It is an iterative algorithm that is highly parallelizable. It is, however, restricted to positive or negative definite (or, more generally, accretive) stencils that have definite substencils. Fortunately, most stencils arising in practice are of this type.

We use a Problem II stencil as an example:

$$nx = 10, \quad ny = 80, \tag{6.1}$$

$$C(i,j) = -4 - \frac{j-1}{20}, \tag{6.2}$$

$$L = R = D = U = 1. \tag{6.3}$$

We use two different domain decompositions, each having four subdomains:

(1) columns 1 – 20, 21 – 40, 41 – 60, and 61 – 80.

(2) columns 11 – 30, 31 – 50, 51 – 70, and 71 – 80 plus 1 – 10.

We are no longer dealing with just one set of coefficient matrices, but rather with various sets of submatrices of the coefficient matrices. For this reason, the convenience provided by declaring the coefficient matrices as global variables is not available anymore. Thus, all functions described in Section 2 must be redefined, including the coefficient matrices as input arguments. Furthermore, it is more appropriate to rewrite the sweeping subroutine M-files (either those given in Section 3 or in Section 4) as functions, to enable them to be applied to different subdomains. The task involves nothing more than a careful record keeping of which variables are needed as input and which variables will be generated as output.

In the following pseudo-code, we add a "subscript" `k` (in principle, `k` runs from 1 to 8) to various variables to indicate that they are associated with the `k`th subdomain. For instance, the coefficient matrices of the subdomains are denoted by `Ck`, `Lk`, etc. Incidentally, MATLAB does not have a convenient way to index matrices, since arrays of dimensions more than two are not supported. One can still simulate (in a slightly awkward way) an array of matrices by concatenating the index (first turned into a string by using the command `num2str`) to the matrix name and then `eval` the entire string.

```
Program: Partial Sweeping for Solving   S[z] = b

   Generate the 8 sets of partial coefficient matrices :
         extract the appropriate submatrices
         equate the first column of Lk and the
            last column of Rk to zero

   Compute the rectifying matrices  RRIk :
         for j from 1 to 8
            RRIk = arr(Ck,Lk,Rk,Dk,Uk)
         end

   Repeat until bb = max(S[z] - b) < desired accuracy :
         sweep subdomains 1, 2, 3, and 4  (odd cycles)
         sweep subdomains 5, 6, 7, and 8  (even cycles)
            by using  z = asw(bb,Ck,Lk,Rk,Dk,Uk,RRIk)
            on the reduced residue equation
   End repeat
```

Results for the test problem are recorded below. The correct algorithm sweeps the stencil using two domain decompositions alternatively, in successive cycles. The maximum residue after each cycle is given in the first column. If only one decomposition is used throughout all the cycles, we still got convergence, but the rate was slow. These results are listed in the other two columns for the sake of comparison.

cycles	mres		
	alternate	only (1)	only (2)
1	2.3786	2.3786	1.5972
2	0.0029	0.4726	0.9638
3	3.6301×10^{-5}	0.3453	0.6072
4	2.9451×10^{-8}	0.1064	0.3743
5	3.2348×10^{-10}	0.0629	0.2327
6	2.6557×10^{-13}	0.0284	0.1440

In addition, we obtained

$$\max(z) = 4.7525, \quad \text{flops1} = 365528, \quad \text{flops2} = 54388.$$

The first flop count, flops1, is for the set-up part of the program — to find the submatrices and the corresponding rectifying matrices. The second flop count is for each cycle of partial sweeping.

A careful scrutiny of the algorithm reveals that the two-direction sweeping implementation given by `arr` and `asw` is not best suited for partial sweeping, even though it is good enough for its original purpose. In all the cycles beyond the first one, the nonhomogeneous part of each residue equation has "support" only on the two center columns of each subdomain. The subroutine `asw` sweeps these errors towards the boundary before determining the initial columns for the correct solution. A better way is to find a different rectifying matrix that gives the initial columns directly from the errors on the two center columns — in a way analogous to the Green's function used in solving elliptic boundary value problems. Not only is the work for sweeping the errors saved, but any rounding error that may occur during the sweeping can be avoided. We omit the simple modifications needed.

Similarly, if a two-stage sweeping subroutine is used for the partial sweeping algorithm, except in the first cycle, the first-stage sweeping used to find `y` is not needed.

7 Perturbed Stencil and the Method of Continuation

In the numerical solution of the Ginzburg-Landau equations arising from superconductivity, Newton's method is applied to the set of nonlinear equations. The stencil equation given by the discrete Ginzburg-Landau operator that constitutes our test Problem IV is only one portion of the equations to be solved in one of the iterative Newton steps. As the computational procedure progresses, the Ginzburg-Landau stencil changes. Variable stencils also arise in time-dependent systems.

If changes in the stencil coefficients are gradual, we can find the inverse of a new stencil from the known inverse of the previous stencil quickly by using the refinement subroutine described in Section 4. Let S be a stencil for which we have determined an accurate rectifying matrix RRI. We are given a new stencil T such that, under some appropriate operator norm, $\|S^{-1}\Delta\| = \|S^{-1}(S - T)\|$ is small. The new stencil equation to be solved is

$$T[z] = (S + \Delta)[z] = b. \tag{7.1}$$

The refinement process is related to the fixed-point iterative scheme

$$z_n = S^{-1}(b - \Delta[z_{n-1}]). \tag{7.2}$$

Alternatively, we can regard RRI as an approximate rectifying matrix for the new stencil, and use the technique discussed after the refining subroutine in Section 4 to improve RRI to give an accurate rectifying matrix for T.

Even if the new stencil is not sufficiently close to S, the technique of continuation can sometimes be used. One simply connects S and T by a homotopy $T(t) = S + t(T - S)$, $0 < t < 1$, and computes the inverse of $T(t_i)$ for a suitably chosen sequence of numbers $t_i \in (0, 1)$ with the final choice $t_m = 1$.

8 Banded Matrices

One attractive feature of the simple one-direction sweeping algorithm for the Dirichlet stencil is that the bulk of the work is in computing the inverse of one single matrix RR of dimension $ny \times ny$, instead of the inverse of a sparse matrix of dimension $(nx)(ny) \times (nx)(ny)$, The more sophisticated two-direction sweeping for a general stencil requires the inversion of a $2ny \times 2ny$ matrix — the amount of work is about eight times greater but still manageable. Using stages introduces substantially more work. Finally, partial sweeping that uses m subdomains in the form of vertical strips means the inversion of m matrices each of dimension $ny \times ny$. In view of the increase in work as sophistication and grid size mount, an obvious question is whether there is still advantage over conventional methods. A satisfactory answer is possible only after further investigation.

Fortunately, if the width of a subdomain is small relative to its length, the corresponding rectifying matrix to be inverted turns out to be banded. It is well known that banded matrices need substantially less effort to invert than full matrices. For a recent reference, see [5] by S. Wright.

Many banded matrices can be inverted by the method of sweeping — in particular, those for which none of the elements in the band vanishes. A discussion can be found in [3]. When the banded matrix is large, instability of the sweeping will be a problem, and the technique of multi-stage or partial sweeping can be applied as in the case of solving stencil equations. We omit the details.

Acknowledgments I thank my colleagues Paul Plassmann and Steve Wright for many useful discussions and for sharing their recent work [4] and in making available their numerical codes.

References

[1] Ascher, U. M., Mattheij, R. M. M., and Russell, R. D., **Numerical Solutions of Boundary Value Problems for Ordinary Differential Equations**, Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[2] Du, Q., Gunzburger, M., and Peterson, J., *Modeling and analysis of a periodic Ginzburg-Landau model for type-II superconductors*, preprint, 1992.

[3] Kwong, Man Kam, *Sweeping algorithms for inverting the discrete Ginzburg-Landau operator*, Mathematics and Computer Science Division Preprint MCS-P307-0492, Argonne National Laboratory, Argonne, Ill., December 1991.

[4] Garner, J., Spanbauer, M., Benedek, R., Strandburg, K., Wright, S., and Plassmann, P., *Critical fields of Josephson-coupled superconducting multilayers*, Mathematics and Computer Science Division Preprint MCS-P281-1291, Argonne National Laboratory, Argonne, Ill., December 1991.

[5] Wright, S., *A parallel algorithm for banded linear systems*, SIAM J. Scientific Statistical Computing 12 (1991), 824–842.