PROGRAMMING OF FINITE DIFFERENCE METHODS IN MATLAB

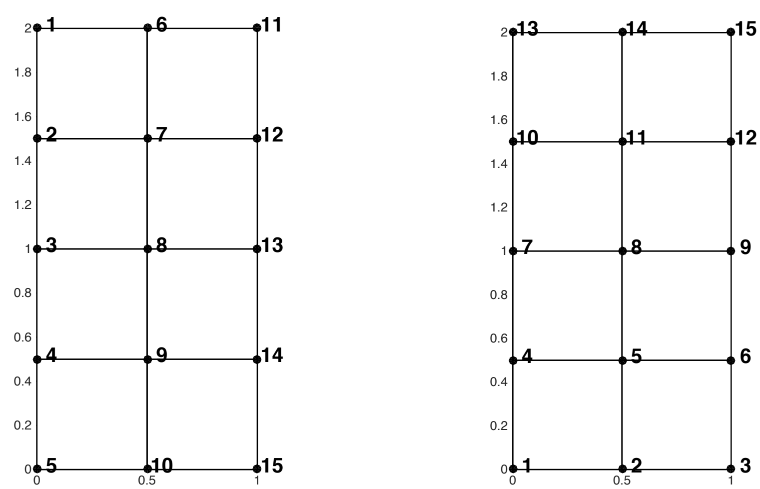
LONG CHEN

We discuss efficient ways of implementing finite difference methods for solving the Poisson equation on rectangular domains in two and three dimensions. The key is the matrix indexing instead of the traditional linear indexing. With such an indexing system, we will introduce a matrix-free and a tensor product matrix implementation of finite difference methods.

1. INDEXING USING MATRICES

Geometrically a 2-D grid is naturally linked to a matrix. When forming the matrix equation, we need to use a linear indexing to transfer this 2-D grid function to a 1-D vector function. We can skip this artificial linear indexing and treat our function $u(x, y)$ as a matrix function `u(i,j)`. The multiple subscript indexing to the linear indexing is build into the matrix. The matrix is still stored as a 1-D array in memory. The default linear indexing in MATLAB is column wise. For example, a matrix `A = [2 9 4; 3 5 11]` is stored in memory as the array `[2 3 9 5 4 11]'`. One can use a single index to access an element of the matrix, e.g., `A(4) = 5`.

In MATLAB, there are two matrix systems to represent a two dimensional grid: the geometry consistent matrix and the coordinate consistent matrix. To fix ideas, we use the following example. The domain $(0, 1) \times (0, 2)$ is decomposed into a uniform grid with mesh size $h = 0.5$. The linear indexing of these two systems are illustrate in the following figures.



(a) meshgrid (b) ndgrid

FIGURE 1. Two indexing systems

The command `[x,y] = meshgrid(0:0.5:1,2:-0.5:0)` will produce 5×3 matrices. Note that the flip of the ordering `[2:-0.5:0]` in the y-coordinate makes the matrix is geometrically consistent with the domain in the sense that the shape of the matrix matches the shape of the domain. This index system is illustrated in Fig. 1(a).

```
>> [x,y] = meshgrid(0:0.5:1,2:-0.5:0)
x =
         0    0.5000    1.0000
         0    0.5000    1.0000
         0    0.5000    1.0000
         0    0.5000    1.0000
         0    0.5000    1.0000
y =
    2.0000    2.0000    2.0000
    1.5000    1.5000    1.5000
    1.0000    1.0000    1.0000
    0.5000    0.5000    0.5000
         0         0         0
```

In this geometrically consistent system, however, there is an inconsistency of the convention of notation of matrix and Cartesian coordinate. Let us figure out the mapping between the algebraic index `(i,j)` and the geometric coordinate (x_i, y_j) of a grid point. In the command

`[x,y] = meshgrid(xmin:hx:xmax,ymax:-hy:ymin),`

the coordinate of the (i, j)-th grid point is

$$(x_j, y_i) = (x_{\min} + (j-1)h_x, y_{\min} + (n-i+1)h_y),$$

which violates the convention of associating index i to x_i and j to y_j. For a matrix entry `A(i,j)`, the 1st index i is the row and the 2nd j is the column while in Cartesian coordinate, i is usually associated to the x-coordinate and j to the y-coordinate.

The command `ndgrid` will produce a coordinate consistent matrix in the sense that the mapping is `(i,j)` to (x_i, y_j) and thus will be called coordinate consistent indexing. For example, `[x,y] = ndgrid(0:0.5:1,0:0.5:2)` will produce two 3×5 not 5×3 matrices; see Fig. 1(b).

```
>> [x,y] = ndgrid(0:0.5:1,0:0.5:2)
x =
         0         0         0         0         0
    0.5000    0.5000    0.5000    0.5000    0.5000
    1.0000    1.0000    1.0000    1.0000    1.0000
y =
         0    0.5000    1.0000    1.5000    2.0000
         0    0.5000    1.0000    1.5000    2.0000
         0    0.5000    1.0000    1.5000    2.0000
```

In the output of

`[x,y] = ndgrid(xmin:hx:xmax,ymin:hy:ymax),`

the coordinate of the (i, j)-th grid point is

$$(x_i, y_j) = (x_{\min} + (i-1)h_x, y_{\min} + (j-1)h_y).$$

In this system, one can link the index change to the conventional change of the coordinate. For example, the central difference $u(x_i + h, y_j) - u(x_i - h, y_j)$ is transferred to `u(i+1,j) - u(i-1,j)`. When display a grid function `u(i,j)`, however, one must be aware of that the shape of the matrix is not geometrically consistent with the domain.

Remark 1.1. No matter which indexing system in use, when plotting a grid function using `mesh` or `surf`, it results the same geometrically consistent figures.

Which index system shall we choose? First of all, choose the one you feel more comfortable and thus has less chance to produce bugs. A more subtle issue is related to the linear indexing of a matrix in MATLAB. Due to the column-wise linear indexing, it is much faster to access one column instead of one row at a time. Depending on which coordinate direction the subroutine will access more frequently, one chose the corresponding coordinate-index system. For example, if one wants to use vertical line smoothers, then it is better to use `meshgrid` system and `ndgrid` system for horizontal lines.

We now discuss the transfer between multiple subscripts and the linear indexing. The commands `sub2ind` and `ind2sub` is designed for such purpose. We include two examples below and refer to the documentation of MATLAB for more comprehensive explanation and examples. The command `k=sub2ind([3 5],2,4)` will give `k=11` and `[i,j]=ind2sub([3 5],11)` produces `i=2, j=4`. In the input `sub2ind(size, i,j)`, the `i,j` can be arrays of the same dimension. In the input `ind2sub(size, k)`, the `k` can be a vector and the output `[i,j]` will be two arrays of the same length of `k`. Namely these two commands support vectors arguments.

For a matrix function `u(i,j)`, `u(:)` will change it to a 1-D array using the column-wise linear indexing and `reshape(u,m,n)` will change a 1-D array to a 2-D matrix function.

A more intuitive way to transfer multiple subscripts into the linear index is to explicitly store an index matrix. For `meshgrid` system, use

```
idxmat = reshape(uint32(1:m*n), m, n);
```

```
>> idxmat = reshape(uint32(1:15),5,3)
idxmat =
           1           6          11
           2           7          12
           3           8          13
           4           9          14
           5          10          15
```

Then one can easily get the linear indexing of the j-th column of a $m \times n$ matrix by using `idxmat(:,j)` which is equivalent to `sub2ind([m n], 1:m, j*ones(1,m))` but much easier and intuitive. The price to pay is the extra memory for the full matrix `idxmat` which can be minimized using `uint32`.

For the `ndgrid` system, to get a geometrically consistent index matrix, we can use the following command.

```
idxmat = flipud(transpose(reshape(uint32(1:m*n), n, m)));
```

```
>> idxmat = flipud(reshape(uint32(1:15),3,5)')
idxmat =
          13          14          15
          10          11          12
           7           8           9
           4           5           6
           1           2           3
```

For such coordinate consistent system, however, it is recommended to use the subscript indexing directly.

Similarly we can generate matrices to store the subscripts. For the `meshgrid` system

```
>> [jj,ii] = meshgrid(1:3,1:5)
jj =
     1     2     3
     1     2     3
     1     2     3
     1     2     3
     1     2     3
ii =
     1     1     1
     2     2     2
     3     3     3
     4     4     4
     5     5     5
```

For the `ndgrid` system

```
>> [ii,jj] = ndgrid(1:3,1:5)
ii =
     1     1     1     1     1
     2     2     2     2     2
     3     3     3     3     3
jj =
     1     2     3     4     5
     1     2     3     4     5
     1     2     3     4     5
```

Then `ii(k),jj(k)` will give the subscript of the k-th node.

Last we discuss the access of boundary points which is important when imposing boundary conditions. Using subscripts of `meshgrid` system, the index of each part of the boundary of the domain is

`meshgrid`: left - `(:,1)` right - `(:,end)` top - `(1,:)` bottom - `(end,:)`

which is consistent with the boundary of the matrix. If using a `ndgrid` system, it becomes

`ndgrid`: left - `(1,:)` right - `(end,:)` top - `(:,end)` bottom - `(:,1)`.

Remember the coordinate consistency: i to x and j to y. Thus the left boundary will be $i = 1$ corresponding to $x = x_1$.

The linear indexes of all boundary nodes can be found by the following codes

```
isbd = true(size(u));
isbd(2:end-1,2:end-1) = false;
bdidx = find(isbd(:));
```

In the first line, we use `size(u)` such that it works for both `meshgrid` and `ndgrid` system.

2. Matrix free implementation

Here the 'matrix free' means that the matrix-vector product Au can be implemented without forming the matrix A explicitly. Such matrix free implementation will be useful if we use iterative methods to compute $A^{-1}f$, e.g., the Conjugate Gradient methods which only requires the computation of Au. Ironically this is convenient because a matrix is used to store the function. For the matrix-free implementation, the coordinate consistent system, i.e., `ndgrid`, is more intuitive since the stencil is realized by subscripts.

Let us use a matrix `u(1:m,1:n)` to store the function. The following double loops will compute Au for all interior nodes. The h^2 scaling will be moved to the right hand side. For Neumann boundary conditions, additional loops for boundary nodes are needed since the boundary stencils are different; see .

```
for i = 2:m-1
    for j = 2:n-1
        Au(i,j) = 4*u(i,j) - u(i-1,j) - u(i+1,j) - u(i,j-1) - u(i,j+1);
    end
end
```

Since MATLAB is an interpret language, every line will be complied when it is executed. A general guideline for efficient programming in MATLAB is: *avoid large `for` loops*. A simple modification of the above double loops is to use the vector indexing.

```
i = 2:m-1;
j = 2:n-1;
Au(i,j) = 4*u(i,j) - u(i-1,j) - u(i+1,j) - u(i,j-1) - u(i,j+1);
```

To evaluate the right hand side, we can use coordinates `(x,y)` in the matrix form. For example, for $f(x,y) = 8\pi^2 \sin(2\pi x)\cos(2\pi y)$, the h^2 scaled right hand side can be computed as

```
[x,y] = ndgrid(0:h:1,0:h:1);
fh2 = h^2*8*pi^2*sin(2*pi*x).*cos(2*pi*y);
```

Note that `.*` is used to compute the component-wise product for two matrices. For non-homogenous boundary conditions, one needs to evaluate boundary values and add to the right hand side. The evaluation of a function on the whole grid is of complexity $\mathcal{O}(m \times n)$. For boundary condition, we can reduce to $\mathcal{O}(m+n)$ by restricting to `bdidx` only.

```
u(bdidx) = sin(2*pi*x(bdidx)).*cos(2*pi*y(bdidx));
```

One Jacobi iteration for solving the matrix equation $Au = f$ can be implemented as

```
j = 2:n-1;
i = 2:m-1;
u(i,j) = (fh2(i,j) + u(i-1,j) + u(i+1,j) + u(i,j-1) + u(i,j+1))/4;
```

The weighted Jacobi iteration can be obtained as a combination of current approximation of Jacobi iteration. Let `uJ` be the updated using Jacobi iteration and $\omega \in (0,1)$ be a weight. Then the weighted Jacobi iteration is

```
u = omega*u + (1-omega)*uJ;
```

A more efficient iterative methods, Gauss-Seidel (G-S) iteration updates the coordinates sequentially one at a time. Here is the implementation using `for` loops.

```
for j = 2:n-1
    for i = 2:m-1
        u(i,j) = (fh2(i,j) + u(i-1,j) + u(i+1,j) + u(i,j-1) + u(i,j+1))/4;
    end
end
```

The ordering does matter in the Gauss-Seidel iteration. The backwards G-S can be implemented by inverse the ordering of `i,j` indexing.

```
for j = n-1:-1:2
    for i = m-1:-1:2
        u(i,j) = (fh2(i,j) + u(i-1,j) + u(i+1,j) + u(i,j-1) + u(i,j+1))/4;
    end
end
```

Note that for the matrix-free implementation, there is no need to modify the right hand side for the Dirichlet boundary condition. The boundary values of `u` is assigned before the iteration and remains the same since only the interior nodal values are updated during the iteration. For Neumann boundary conditions, an additional update on boundary nodes is needed.

The symmetric version Gauss-Seidel will be the combination of forward and backwards and is an SPD operator which can be used in `pcg` to accelerate the computation of an approximated solution to the linear system $Au = f$.

The vectorization of Gauss-Seidel iteration is subtle. If we simply remove the `for` loops, it is the Jacobi iteration since the values of `u` on the right hand side is the old one. To vectorize G-S, let us first classify the nodes into two category: red nodes and black nodes; see Fig 2. Black nodes can be identified as `mod(i+j,2) == 0`. A crucial observation is that to update red nodes only values of black nodes are needed and vice verse. Then Gauss-Seidel iteration applied to this red-black ordering can be implemented as Jacobi iterations.

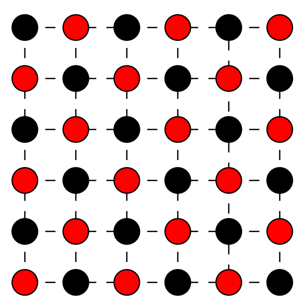


FIGURE 2. Red-Black Ordering of vertices

```
[m,n] = size(u);
% case 1 (red points): mod(i+j,2) == 0
i = 2:2:m-1; j = 2:2:n-1;
u(i,j) = (fh2(i,j) + u(i-1,j) + u(i+1,j) + u(i,j-1) + u(i,j+1))/4;
i = 3:2:m-1; j = 3:2:n-1;
u(i,j) = (fh2(i,j) + u(i-1,j) + u(i+1,j) + u(i,j-1) + u(i,j+1))/4;
% case 2 (black points): mod(i+j,2) == 1
i = 2:2:m-1; j = 3:2:n-1;
u(i,j) = (fh2(i,j) + u(i-1,j) + u(i+1,j) + u(i,j-1) + u(i,j+1))/4;
i = 3:2:m-1; j = 2:2:n-1;
u(i,j) = (fh2(i,j) + u(i-1,j) + u(i+1,j) + u(i,j-1) + u(i,j+1))/4;
```

3. TENSOR PRODUCT MATRIX IMPLEMENTATION

When the grid is in the tensor product type, the matrix-vector multiplication Au can be implemented using the tensor product of 1-D matrix which will be called the tensor-product matrix implementation, for which it is better to use the geometry consistent, i.e., `meshgrid`, indexing system.

For a uniform grid in one dimension, the matrix of the central difference discretization of the Poisson equation is tri-diagonal and can be generated by

```
e = ones(n,1);
T = spdiags([-e 2*e -e], -1:1, n, n);
```

The boundary condition can be build into `T` by changing the entries near the boundary. Here `T` corresponds to the homogenous Dirichlet boundary condition.

For a two dimensional $n \times n$ uniform grid, the five point stencil can be decomposed into

$$(2u_{i,j} - u_{i-1,j} - u_{i+1,j}) + (2u_{i,j} - u_{i,j-1} - u_{i,j+1})$$

which can be realized by the left product and right product with the 1-D matrix

```
Au = u*T + T*u;
```

For different mesh size or different stencil in x and y-direction, one should generate specific `Tx` and `Ty` and use

```
Au = u*Tx + Ty*u; % meshgrid system
Au = Tx*u + u*Ty; % ndgrid system
```

We then write the matrix as the tensor product of 1-D matrices.

Definition 3.1. *Let $A_{m\times n}$ and $B_{p\times q}$ be two matrices. Then the Kronecker (tensor) product of A and B is*

$$A \otimes B = \begin{pmatrix} a_{11}B & \dots & a_{1n}B \\ \dots & \dots & \dots \\ a_{m1}B & \dots & a_{mn}B \end{pmatrix}$$

The matlab command is `kron(A,B)`.

Let $A_{m\times m}$, $B_{n\times n}$ and $X_{m\times n}$ be there matrices. Then it is straightforward to verify the identities

(1) $(AX)(:) = (I_n \otimes A) \cdot X(:)$.
(2) $(XB)(:) = (B^{\intercal} \otimes I_n) \cdot X(:)$.

Here we borrow the notation $(:)$ to change a matrix to a vector by stacking columns from left to right. Therefore the matrix A for the five point stencil is

$$A = I_n \otimes T + T \otimes I_n,$$

and the corresponding matlab code is

```
A = kron(speye(nx),Ty) + kron(Tx,speye(ny));  % meshgrid system
A = kron(speye(ny),Tx) + kron(Ty,speye(nx));  % ndgrid system
```

Exercise 3.2. Write out a similar formulae for Neumann boundary condition.
Hint: Change both `T` and I at boundary indices.

Note that in the computation, it is not needed to form the tensor product of matrices. Instead use the left and right product to compute `Au` if only the matrix-vector product is of interest.

In general let $A_{m\times n}$, $B_{p\times q}$, and $x_{q\times n}$ be three matrices. Then

$$(A \otimes B) \cdot x(:) = ((B * x) * A^T)(:).$$

A MATLAB function to realize this is attached below. (Thanks Lin!)

```
function y = kronecker_product(A,B,x)
% Lin Zhong (lzhong1@uci.edu)  June, 2013.
[r1,c1] = size(A);
[r2,c2] = size(B);
rx = reshape(x,c2,c1); % change the vector x into a matrix
ry = B*rx*A';
y = ry(:);
```

4. Three and Higher Dimensions

The `meshgrid` can be used to generate a 3-D tensor product grid and `ndgrid` can generate an n-D grid for any positive integer n. The two dimensional matrix can be generalized to multi-dimensional arrays with more than two subscripts (also called tensor). Please read the help doc on `Multidimensional Arrays` in MATLAB first. In the following we discuss issues related to the implementation of the Poisson equation in 3-D.

Slices in each direction are in different type. Only `A(:,:,i)` is a matrix stored consecutively, which is called the i-th page of `A`. But `A(i,:,:)` will be formed by elements across pages and thus not a matrix. One can use `squeeze(A(i,:,:))` to squeeze it into a matrix. Again it is stored column wise and which coordinate (x, y or z) corresponds to the column will depend on the index system.

The tensor product representation of the matrix is still valid in high dimensions. For example, in 3-D the 7-point stencil Laplace matrix is

$$A = I_n \otimes I_n \otimes T + I_n \otimes T \otimes I_n + T \otimes I_n \otimes I_n.$$

The matrix-free computation of `Au` is straightforward

```
[n1,n2,n3] = size(u);
i = 2:n1-1;
j = 2:n2-1;
k = 2:n3-1;
Au(i,j,k) = 6*u(i,j,k) - u(i-1,j,k) - u(i+1,j,k) - u(i,j-1,k) - u(i,j+1,k) ...
            u(i,j,k-1) - u(i,j,k+1);
```

The tensor product matrix implementation is less obvious since the basic data structure in MATLAB is matrix not tensor. Denote the stencil matrix in each direction by $T_i, i = 1, 2, 3$. The first two dimensions can be computed as

```
for k = 1:n3
    Au(:,:,k) = u(:,:,k)*T2 + T1*u(:,:,k);
end
```

The third one is different

```
for j = 1:n2
    Au(:,j,:) = squeeze(u(:,j,:))*T3;
end
```

To vectorize the above code, i.e., avoid `for` loop, one can use `reshape` which operates in a column-wise manner. First think about the original data as a long vector by stacking columns. Then `reshape` will create the reshaped matrix by transforming consecutive elements of this long vector into different shape.

We explain the index change by the following example.

```
>> u = reshape(1:3*5*2,3,5,2)
u(:,:,1) =
     1     4     7    10    13
     2     5     8    11    14
     3     6     9    12    15

u(:,:,2) =
    16    19    22    25    28
    17    20    23    26    29
    18    21    24    27    30
```

```
>> u1 = reshape(u, n1, n2*n3)
u1 =
     1     4     7    10    13    16    19    22    25    28
     2     5     8    11    14    17    20    23    26    29
     3     6     9    12    15    18    21    24    27    30
```

Reshape in this way is like to put pages consecutively on the plane. This is efficient since it is just a rearrangement of columns.

The stencil in the first direction can be realized by

```
Au1 = reshape(T1*reshape(u, n1, n2*n3), n1, n2, n3);
```

However, we cannot use similar trick to implement difference in the other directions. For example,

```
>> reshape(u, n1*n3, n2)
ans =
     1     7    13    19    25
     2     8    14    20    26
     3     9    15    21    27
     4    10    16    22    28
     5    11    17    23    29
     6    12    18    24    30
```

Multiplication of T_2 to the right will not get the desired result. A simple fix is to use `permute` to permute the desired direction to the first one and afterwards using the operation `ipermute` to switch back.

```
up = permute(u, [2 1 3]);
Au2 = reshape(T2*reshape(up, n2, n1*n3), n2, n1, n3);
Au2 = ipermute(Au2, [2 1 3]);
```

Repeat this procedure for each direction and add them together to get `Au`. It seems cumbersome to using the tensor product matrix implementation comparing with the matrix-free one. The advantage of the tensor-produce structure is: one can easily build the boundary condition, the non-uniform grid size, and non-standard stencil into the one dimensional matrix.