

PROGRAMMING OF FINITE DIFFERENCE METHODS IN MATLAB

LONG CHEN

We discuss efficient implementations of finite difference methods for solving the Poisson equation on rectangular domains in two and three dimensions. The key idea is to use matrix indexing instead of the traditional linear indexing. With this indexing system, we introduce both a matrix-free formulation 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, a linear indexing is often used to map the 2-D grid function to a 1-D vector function. We can skip this artificial step and treat the function $u(x, y)$ as a matrix function $u(i, j)$.

The mapping from multiple subscripts to linear indexing is built into the matrix structure. The matrix is still stored as a 1-D array in memory. In MATLAB, the default linear indexing is column-wise.

1.1. Grid. In MATLAB, there are two matrix systems to represent a two-dimensional uniform grid:

- (1) the geometry-consistent matrix: `meshgrid`;
- (2) the coordinate-consistent matrix: `ndgrid`.

To fix ideas, consider the following example. The domain $\Omega = (0, 1) \times (0, 2)$ is decomposed into a uniform grid with mesh size $h = 0.5$. The linear indexing of these two systems is illustrated in Fig. 1.1.

Geometry-consistent matrix. The command

```
[x,y] = meshgrid(0:0.5:1,2:-0.5:0)
```

produces 5×3 matrices which is geometrically consistent with the domain, in the sense that the shape of the matrix matches the shape of the domain. Note that the reversed ordering $[2:-0.5:0]$ in the y -coordinate. This indexing system is illustrated in Fig. 1.1 (a).

```
1 >> [x,y] = meshgrid(0:0.5:1,2:-0.5:0)
2 x =
3     0    0.5000    1.0000
4     0    0.5000    1.0000
5     0    0.5000    1.0000
6     0    0.5000    1.0000
7     0    0.5000    1.0000
8 y =
9    2.0000    2.0000    2.0000
10    1.5000    1.5000    1.5000
11    1.0000    1.0000    1.0000
12    0.5000    0.5000    0.5000
13     0         0         0
```

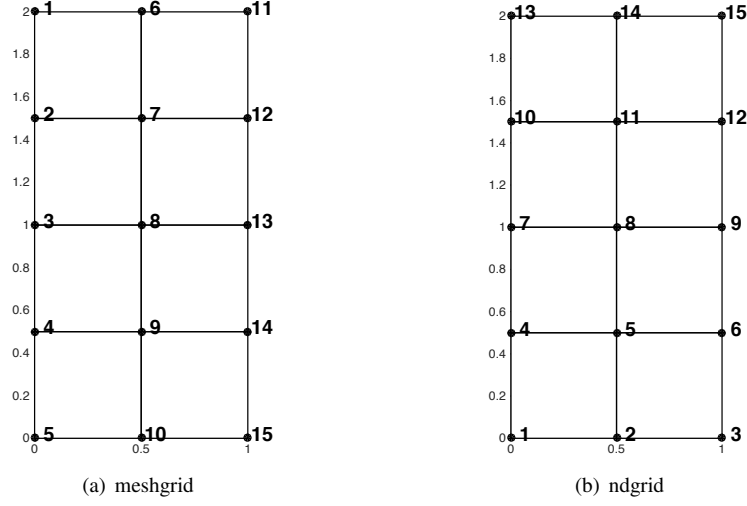


FIGURE 1. Two indexing systems

In this geometrically consistent system, however, there is an inconsistency between the notation of matrices and Cartesian coordinates. In the command

$$[x, y] = \text{meshgrid}(a:h_x:b, d:-h_y:c),$$

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

$$(x_j, y_i) = (a + (j - 1)h_x, d - (i - 1)h_y),$$

which violates the convention of associating index i with x_i and index j with y_j . For a matrix entry $A(i, j)$, the first index i refers to the row and the second index j refers to the column, whereas in Cartesian coordinates, i is usually associated with the x -coordinate and j with the y -coordinate.

Coordinate consistent matrix. 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 the coordinate consistent indexing. In the output of

$$[x, y] = \text{ndgrid}(a:h_x:b, c:h_y:d),$$

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

$$(x_i, y_j) = (a + (i - 1)h_x, c + (j - 1)h_y).$$

In this system, one can link the change of indices to the conventional change of coordinates. For example, the central difference

$$u(x_i + h, y_j) - u(x_i - h, y_j)$$

is represented in discrete form as

$$\text{ndgrid} \quad u(i+1, j) - u(i-1, j).$$

While in the geometric consistent system, it is

$$\text{meshgrid} \quad u(i, j+1) - u(i, j-1).$$

When displaying a grid function $u(i, j)$, one must be aware that the shape of the matrix is not geometrically consistent with the domain. For example,

```
[x,y] = ndgrid(0:0.5:1,0:0.5:2)
```

produces two 3×5 matrices, not 5×3 ; see Fig. 1.1 (b).

```
1 >> [x,y] = ndgrid(0:0.5:1,0:0.5:2)
2 x =
3      0      0      0      0      0
4    0.5000  0.5000  0.5000  0.5000  0.5000
5    1.0000  1.0000  1.0000  1.0000  1.0000
6 y =
7      0    0.5000    1.0000    1.5000    2.0000
8      0    0.5000    1.0000    1.5000    2.0000
9      0    0.5000    1.0000    1.5000    2.0000
```

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.

1.2. Subscript indexing and linear indexing. Recall that the default linear indexing in MATLAB is column-wise. For example, the matrix $A = \begin{bmatrix} 2 & 9 & 4 \\ 3 & 5 & 11 \end{bmatrix}$ 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., $A(4) = 5$.

For a matrix function $u(1:m, 1:n)$, the command `u(:)` converts it into a 1-D array using column-wise linear indexing, and `reshape(u,m,n)` converts a 1-D array back into a 2-D matrix function.

The commands `sub2ind` and `ind2sub` are designed to transfer between subscript indexing and linear indexing. We include two examples and refer to the MATLAB documentation for more details. The command `k = sub2ind([3 5],2,4)` gives `k = 11`, and `[i,j] = ind2sub([3 5],11)` produces `i = 2, j = 4`. In `sub2ind(size,i,j)`, the arguments `i, j` can be arrays of the same dimension. Similarly, in `ind2sub(size,k)`, the argument `k` can be a vector, and the output `[i,j]` will be two arrays of the same length as `k`. Namely, these two commands support vector arguments.

A more intuitive approach is to explicitly store an index matrix. For the `meshgrid` system, use the following command

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

Then one can easily obtain the linear indexing of the j -th column by using `idxmat(:, j)`, which is equivalent to `sub2ind([m n], 1:m, j*ones(1,m))`, but is much simpler and more intuitive. The cost is the extra memory needed for the full matrix `idxmat`, which can be reduced by using `uint32`.

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

```

1 >> [jj,ii] = meshgrid(1:3,1:5)
2 jj =
3     1     2     3
4     1     2     3
5     1     2     3
6     1     2     3
7     1     2     3
8 ii =
9     1     1     1
10    2     2     2
11    3     3     3
12    4     4     4
13    5     5     5

```

Then `ii(k)`, `jj(k)` will give the subscript of the k -th node. Notice that in the output, the ordering of subscripts is `[jj,ii]`.

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

```

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

```

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

1.3. Boundary. As an example, we discuss accessing boundary points. Using subscripts in the `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 displayed matrix.

If we use the `ndgrid` system, the indices become

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

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

The linear indexes of all boundary nodes can be also found by using the logical matrix

```

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

```

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

2. MATRIX-FREE IMPLEMENTATION

The term ‘matrix-free’ means that the matrix-vector product Au can be implemented without forming the matrix A explicitly. Such a matrix-free implementation is useful when iterative methods are used to compute $A^{-1}f$, e.g., the Conjugate Gradient (CG) method, which only requires the computation of Au . Ironically, the matrix-free implementation is possible because the function is stored as a matrix rather than as a vector.

2.1. Stencil and data evaluation. The function is stored as a matrix $u(1:m, 1:n)$. The following double loops compute Au for all interior nodes. The h^2 scaling is moved to the right-hand side. For Neumann boundary conditions, additional loops for boundary nodes are needed since the boundary stencils are different; see [Introduction to Finite Difference Methods](#).

```

1 function Au = Poisson(u)
2 for i = 2:m-1
3     for j = 2:n-1
4         Au(i,j) = 4*u(i,j) - u(i-1,j) - u(i+1,j) - u(i,j-1) - u(i,j+1);
5     end
6 end

```

Since MATLAB is an interpreted language, every line is compiled 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 vector indexing:

```

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

```

One can use the double `for` loops for debugging, and then simply remove the `for` loops to speed up.

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

```

1 [x,y] = ndgrid(0:h:1,0:h:1);
2 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. 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 `bddidx` only.

```

1 u(bddidx) = sin(2*pi*x(bddidx)).*cos(2*pi*y(bddidx));

```

Note that for the matrix-free implementation, there is no need to modify the right-hand side for Dirichlet boundary conditions. The boundary values of u are assigned before the iteration and remain fixed, since only the interior nodal values are updated. For Neumann boundary conditions, however, an additional update of the boundary nodes is required.

2.2. Iterative methods. We consider a matrix-free implementation of iterative methods for solving $Au = f$ using the finite difference method. A generic form is:

```

1 while k < maxIt & max(abs(residual(:))) > tol
2     u = OneStep(fh2, u);
3     k = k + 1;
4     residual = fh2 - Au;
5 end

```

We set a maximum iteration step `maxIt` to avoid an endless loop. The tolerance `tol` should be normalized, or equivalently expressed in terms of the relative residual. For example, with a zero initial guess, one may set `tol = 1e-6*fh2`.

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

```

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

```

which simply solves the finite difference equation at $[i, j]$.

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

```

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

```

Gauss-Seidel iteration. A more efficient iterative method, Gauss–Seidel (G-S) iteration, updates the coordinates sequentially one at a time.

```

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

```

The ordering matters in the Gauss–Seidel iteration. The backward G-S method can be implemented by reversing the order of the i, j loops.

```

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

```

The symmetric Gauss–Seidel is the combination of one forward and one backward G-S iteration. It is an SPD operator and can be used in `pcg` to accelerate the computation.

Vectorization of the Gauss–Seidel iteration is subtle. If we simply remove the `for` loops, the result is the Jacobi iteration, since the values of u on the right-hand side are the input one, not updating when index is changing. To vectorize G-S, we first classify the nodes into two categories: red nodes and black nodes; see Fig. 2. Black nodes can be identified by the condition $\text{mod}(i+j, 2) == 0$. A crucial observation is that updating red nodes requires only the values of black nodes, and updating black nodes requires only the values of red nodes. Thus, Gauss–Seidel iteration applied to this red-black ordering can be implemented as Jacobi iterations.

```

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

```

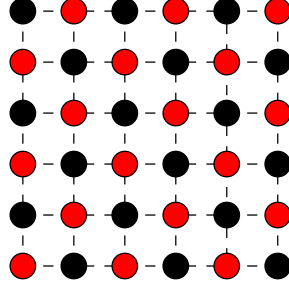


FIGURE 2. Red-Black Ordering of vertices

```

4  u(i,j) = (fh2(i,j) + u(i-1,j) + u(i+1,j) + u(i,j-1) + u(i,j+1))/4;
5  i = 3:2:m-1; j = 3:2:n-1;
6  u(i,j) = (fh2(i,j) + u(i-1,j) + u(i+1,j) + u(i,j-1) + u(i,j+1))/4;
7  % case 2 (black points): mod(i+j,2) == 1
8  i = 2:2:m-1; j = 3:2:n-1;
9  u(i,j) = (fh2(i,j) + u(i-1,j) + u(i+1,j) + u(i,j-1) + u(i,j+1))/4;
10 i = 3:2:m-1; j = 2:2:n-1;
11 u(i,j) = (fh2(i,j) + u(i-1,j) + u(i+1,j) + u(i,j-1) + u(i,j+1))/4;

```

We can find out all black nodes by checking if $\text{mod}(i+j, 2) == 0$.

```

1  [jj,ii] = meshgrid(1:n,1:m);
2  blackNodes = (mod(ii+jj,2) == 0);
3  redNodes = ~blackNodes;

```

Then the red-black Gauss-Seidel iteration is

```

1  i = ii(blackNodes); j = jj(blackNodes);
2  u(i,j) = (fh2(i,j) + u(i-1,j) + u(i+1,j) + u(i,j-1) + u(i,j+1))/4;
3  i = ii(redNodes); j = jj(redNodes);
4  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 of tensor-product type, the matrix-vector multiplication Au can also be implemented using the tensor product of one-dimensional matrices. This will be called the tensor-product matrix implementation.

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

```

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

```

Boundary conditions can be incorporated into T by modifying the entries near the boundary. Here T corresponds to homogeneous Dirichlet boundary conditions. The matrix T is stored as a sparse matrix (only non-zero stored) to save memory and operations.

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

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

which can be realized by left and right multiplication with the one-dimensional matrix:

$$\mathbf{A}\mathbf{u} = \mathbf{u}*\mathbf{T} + \mathbf{T}*\mathbf{u};$$

For different mesh sizes or different stencils in the x - and y -directions, one should generate specific matrices $\mathbf{T}_x$ and $\mathbf{T}_y$ and use

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

For Neumann boundary condition, modify $\mathbf{T}_x$ and $\mathbf{T}_y$.

We can write the matrix as the tensor product of one-dimensional matrices. Let $A_{m \times n}$ and $B_{p \times q}$ be two matrices. The Kronecker (tensor) product of A and B is defined as

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

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

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

- (1) $(AX)(:) = (I_n \otimes A)X(:)$,
- (2) $(XB)(:) = (B^T \otimes I_m)X(:)$.
- (3) $(A \otimes B)X(:) = ((B * X) * A^T)(:)$.

Here we use the notation $(:)$ to denote reshaping a matrix into a vector by stacking its columns from left to right.

Therefore, the matrix A for the five-point stencil is

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

and the corresponding MATLAB code is

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

Again in the computation, it is not needed to form A . Instead use the left and right matrix-product to compute $\mathbf{A}\mathbf{u}$ if only the matrix-vector product is of interest.

4. THREE AND HIGHER DIMENSIONS

The `meshgrid` command 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 tensors). Please refer to the MATLAB documentation on `Multidimensional Arrays`. We restrict our discussion on the implementation of the Poisson equation in 3-D.

The matrix-free computation of $\mathbf{A}\mathbf{u}$ is straightforward

```
1 [n1,n2,n3] = size(u);
2 i = 2:n1-1; j = 2:n2-1; k = 2:n3-1;
3 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) ...
4             u(i,j,k-1) - u(i,j,k+1);
```

The tensor-product representation of the matrix remains valid in higher 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 tensor-product matrix implementation is less direct, since the basic data structure in MATLAB is a matrix rather than a tensor. Slices in each direction are of different types. Only $A(:, :, i)$ is a matrix stored consecutively, called the i -th page of A . By contrast, $A(i, :, :)$ consists of elements across pages and is therefore not a matrix. One can use `squeeze(A(i, :, :))` to reshape it into a matrix. Again, storage is column-wise, and which coordinate (x , y , or z) corresponds to the column depends on the chosen indexing system.

Denote the stencil matrix in each direction by T_i , $i = 1, 2, 3$. The first two directions can be computed as

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

The third direction is different:

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

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

We explain the index change with the following example.

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

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

Reshaping in this way is like placing pages consecutively on the plane. This is efficient since it is merely a rearrangement of columns.

The stencil in the first direction can be implemented as

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

However, a similar trick cannot be used to implement the stencil in the other two directions. For example,

```
1 >> reshape(u, n1*n3, n2)
2 ans =
3     1     7    13    19    25
```

4	2	8	14	20	26
5	3	9	15	21	27
6	4	10	16	22	28
7	5	11	17	23	29
8	6	12	18	24	30

Multiplication of T_2 to the right of this reshaped matrix will not get the desired result.

A simple fix is to use `permute` to bring the desired direction to the first dimension, and then use `ipermute` to switch back afterwards.

```

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

```

Repeating this procedure for each direction and summing them together gives Au . This approach may seem cumbersome compared with the matrix-free implementation. The advantage is that one can easily incorporate boundary conditions, non-uniform grid sizes, and non-standard stencils into the one-dimensional matrices, without looping over boundary indices to modify the stencil.