We discuss efficient ways of implementing finite difference methods for solving 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 matrix-free or 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 index to linear indexing is build into the matrix. The matrix is still stored as a 1-D array in memory. The default linear indexing is column wise. For example, a 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 one single index to access element of the matrix, e.g., \( A(4) = 5 \).

In MATLAB, there are two matrix systems to represent a two dimensional grid: geometry consistent matrix and 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.

\[ \text{FIGURE 1. Two indexing systems} \]
The command \[x,y\] = meshgrid(0:0.5:1,2:-0.5:0) will produce a 5 × 3 matrices. Note that the flip of indexing in the \(y\)-coordinate makes the matrix is geometrically consistent with the domain. This index system is illustrated in Fig. 1(a).

We then figure out the mapping between the algebraic index \((i,j)\) and the geometric coordinate \((x_i, y_j)\) of a grid point. There is an inconsistency of the convention of notation of matrix and Cartesian coordinate. For a matrix, the 1st index \(i\) is the row and the 2nd \(j\) is the column while in Cartesian coordinate, \(i\) is associated to the \(x\)-coordinate and \(j\) to the \(y\)-coordinate. In the command \[x,y\] = meshgrid(xmin:hx:xmax,ymax:-hy:ymin), the coordinate of the \((i,j)\)-th grid is \((x_j, y_i) = (x_{\text{min}} + (j - 1)h_x, y_{\text{min}} + (n - i + 1)h_y)\), which violates the convention of associating index \(i\) to \(x_i\) and \(j\) to \(y_j\).

If one is more comfortable with the \((i,j)\) to \((x_i,y_j)\) mapping, one can use the command ndgrid. For example, \[x,y\] = ndgrid(0:0.5:1,0:0.5:2) will produce two 3 × 5 matrices. In the output of \[x,y\] = ndgrid(xmin:hx:xmax,ymin:hy:ymin), the coordinate of the \((i,j)\)-th grid is \((x_i, y_j) = (x_{\text{min}} + (i - 1)h_x, y_{\text{min}} + (j - 1)h_y)\). This index system is called coordinate consistent and illustrated in Fig. 1(b). In this system, one had better associate the index change to the corresponding change of 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 the matrix is not geometrically consistent with the domain.

```
>> [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

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

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

Which index system shall we chose? First chose the one you feel more comfortable. 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 line 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 if want to use horizontal lines, then `ndgrid` system.

We now discuss the transfer between multiple subscripts and 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. 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.

For a matrix function `u(i, j)`, `u(:)` will change it to a 1-d array using 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 linear indexing is to explicitly store an index matrix. For `meshgrid` system

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

```matlab
>> 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 × 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 of storing `idxmat`. Thus we use `uint32` to reduce the memory requirement.

For `ndgrid` system, to get a geometrically consistent index matrix, we can use the following command. But for such coordinate consistent system, it is better to use the subscripts directly.

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

```matlab
>> idxmat = flipud(reshape(uint32(1:15), 3, 5))
idxmat =
14     15
11     12
8      9
5      6
2      3
1      2
```

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

```matlab
>> [jj, ii] = meshgrid(1:3, 1:5)
jj =
1     2     3
4     1     2     3
5     1     2     3
6     1     2     3
7     1     2     3
8     1     2     3
9     1     1     1
10    2     2     2
```
For \texttt{ndgrid} system

\begin{verbatim}
>> [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
\end{verbatim}

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 \texttt{meshgrid} system, the index of each part of the boundary of the domain is

\texttt{meshgrid}: left - (1,:) right - (:,end) top - (1,:) bottom - (end,:) 
which is consistent with the boundary of the matrix. If using a \texttt{ndgrid} system, it becomes

\texttt{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 index of all boundary nodes can be found by the following codes

\begin{verbatim}
isbd = true(size(u));
isbd(2:end-1,2:end-1) = false;
bdidx = find(isbd(:));
\end{verbatim}

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

\section{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, e.g., Conjugate Gradient method, to compute \(A^{-1}f\) which only requires the computation of \(Au\). Ironically this is convenient because a matrix is used to store the function. For matrix-free implementation, the coordinate consistent system, i.e., \texttt{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 to modify the boundary stencil.

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

```matlab
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 coordinate $x, y$ in matrix form. For example for $f(x, y) = 8\pi^2 \sin(2\pi x) \sin(2\pi y)$, the scaled right hand side can be computed as

```matlab
[x,y] = meshgrid(0:h:1,1:-h:0);
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 $O(n \times n)$. For boundary condition, we can reduce it to $O(n)$ by restricting to $bdidx$ only.

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

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

```matlab
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

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

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

```matlab
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.

```matlab
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. But modification is needed for non-homogenous Neumann boundary condition.
The symmetric version Gauss-Seidel will be the combination of forward and backwards and is also an SPD operator which can be used in \texttt{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 \texttt{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 \(\text{mod}(i+j, 2) == 0\). 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](image)

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

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 tensor product matrix implementation. For tensor product matrix implementation, it is better to use geometry consistent, i.e., \texttt{meshgrid}, indexing system.

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

1. \(e = \text{ones}(n,1);\)
2. \(T = \text{spdiags}([-e 2*e -e], -1:1, n, n);\)

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

$$Au = u \ast T + T \ast u;$$

For different mesh size or different stencil in $x$ and $y$-direction, one should generate specific $T_x$ and $T_y$ and use

1. $Au = u \ast T_x + T_y \ast u; \quad \% \text{ meshgrid system}$
2. $Au = T_x \ast u + u \ast T_y; \quad \% \text{ ndgrid system}$

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

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

The matlab command is $\text{kron}(A,B)$.

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

1. $(AX)(\cdot) = (I_n \otimes A) \cdot X(\cdot)$.
2. $(XB)(\cdot) = (B^T \otimes I_n) \cdot X(\cdot)$.

Here we borrow the notation $(\cdot)$ 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

1. $A = \text{kron}(\text{speye}(nx),Ty) + \text{kron}(Tx,\text{speye}(ny)); \quad \% \text{ meshgrid system}$
2. $A = \text{kron}(\text{speye}(ny),Tx) + \text{kron}(Ty,\text{speye}(nx)); \quad \% \text{ ndgrid system}$

**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

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

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

```matlab
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 \texttt{meshgrid} can be used to generate a 3-D tensor product grid and \texttt{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 \texttt{Multidimensional Arrays} in MATLAB first. In the following we discuss issues related to the implementation of 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 \texttt{squeeze} (\( \lambda(i, :, :) \)) to squeeze 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

\begin{verbatim}
1 [n1,n2,n3] = size(u);
2 i = 2:n1-1;
3 j = 2:n2-1;
4 k = 2:n3-1;
5 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);
\end{verbatim}

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

\begin{verbatim}
1 for k = 1:n3
2   Au(:,:,k) = u(:,:,k)*T2 + T1*u(:,:,k);
3 end
\end{verbatim}

The third one is different

\begin{verbatim}
1 for j = 1:n2
2   Au(:,j,:) = squeeze(u(:,j,:))*T3;
3 end
\end{verbatim}

To vectorize the above code, i.e., avoid \texttt{for} loop, one can use \texttt{reshape} which operates in a column-wise manner. First think about the original data as a long vector by stacking columns. Then \texttt{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.

\begin{verbatim}
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 u(:,:,2) =
7    16   19   22   25   28
8    17   20   23   26   29
\end{verbatim}
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 difference in the first direction can be realized by

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

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

```matlab
>> 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.

```matlab
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 tensor product matrix implementation comparing with the matrix-free one. The advantage of the former one is: one can easily build the boundary condition, the non-uniform grid size, and non-standard stencil into the one dimensional matrix.