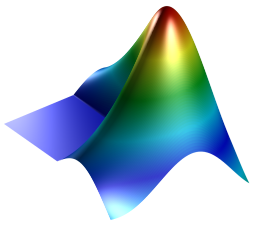
Matrices handling in PDEs resolution with MATLAB®

Riccardo Zamolo
riccardo.zamolo@phd.units.it

Dipartimento di Ingegneria Meccanica e Navale
Università degli Studi Trieste, 34127 TRIESTE



April 6, 2016

OUTLINE

Introduction

- In this part of CFD course we'll see in practical terms how to handle the entities (matrices, vectors, etc.) that arise from the discretization of the fluid dynamics equations;
- We'll focus on the practical implementation of simple 1D and 2D steady-state cases;
- The *Finite Volume Method* (FVM) will be used for the space discretization of the problems, using Cartesian structured grids only;
- MATLAB® will be used as reference language, but a totally similar logic in the manipulation of the matrices is used in other languages (Scilab or Python for example, both free and *Open source*).

Example

- 2D *Poisson* problem:

$$\begin{cases} \nabla^2\phi = s & \text{on } \Omega \subset \mathbb{R}^2 \\ B(\phi) = 0 & \text{on } \partial\Omega \end{cases}$$

When a grid and proper boundary conditions are chosen, the FV discretization leads to the following linear system:

$$\mathbf{A}\mathbf{\Phi} = \mathbf{S}$$

How do we proceed? In practical terms:

- How do we build the coefficient matrix **A**?
- How do we build the source vector **S**?
- How do we find the solution vector **Φ** ?

Source code files

MATLAB® `.m` source code files used in these slides can be found at:

`http://moodle2.units.it/`

entering the TERMOFLUIDODINAMICA COMPUTAZIONALE course:

- C1 `AdvDiff_1D_Sparse.m`
  1D advection-diffusion problem with direct solution (sparse matrix);
- C2 `AdvDiff_2D_Sparse.m`
  2D advection-diffusion problem with direct and pcg solution (sparse matrix);
- C3 `Poisson_2D_Iter.m`
  2D Poisson problem with iterative Jacobi and SOR methods (matrix-free).

1D advection-diffusion problem

- 1D steady-state advection-diffusion equation:

$$(\rho \mathrm{w} \phi)_x = (\Gamma \phi_x)_x + s \tag{1}$$

  with constant properties ρ, Γ and constant velocity w;
- Domain:

$$\Omega = [0, L]$$

- Boundary conditions (BC):

Type	*Dirichlet* BC	*Neumann* BC
Equation	$\phi = \phi_{BC}$	$\Gamma \phi_x = J''_{d,BC}$
Location	$x = 0$	$x = L$

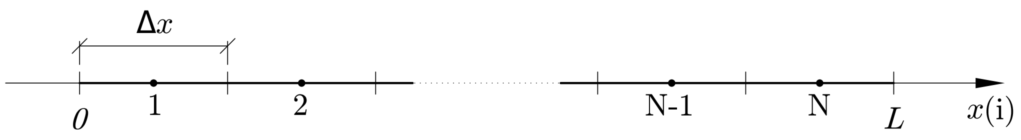
Variables

- Variables definition:

```
% Constant properties (rho, gamma) and advection velocity (w)
rho   = 1 ;
Gamma = 1 ;
w     = 1 ;

% Domain length
L = 2*pi ;

% BC values
phi_BC = 0 ;
Jd_BC  = 0 ;
```

Domain discretization

- Discretization of the domain Ω in N finite volumes (FV) with constant side length



$\Delta x = L/N$:

```
% FV number
N = 100 ;

% FV side length
dx = L / N ;

% FV centroids abscissae (column vector)
X = dx * ( (1:N) - 0.5 )' ;
```

Equation discretization

- FV discretization of eq. (1) with 2nd order Central Differencing Scheme (CDS) approximation for both diffusive and advective fluxes:

$$A_W\phi_W + A_P\phi_P + A_E\phi_E = S_P \tag{2}$$

```
% FV equation coefficients (constants for each FV)
A_W = -rho * w / 2 - Gamma / dx ;
A_E =  rho * w / 2 - Gamma / dx ;
A_P = -( A_W + A_E ) ;
```

- Eq. (2) holds for each of the *N* finite volumes except for the first and the last FV, where this equation must be corrected because one of the neighbour cells doesn't exist: BC will be imposed using a *ghost cell*.

Equation discretization

- Writing eq. (2) for each of the *N* finite volumes, in a matrix notation we have:

$$\begin{bmatrix} A_{P1} & A_{E1} & & 0 \\ A_{W2} & \ddots & \ddots & \\ & \ddots & \ddots & A_{E(\text{N-1})} \\ 0 & & A_{W\text{N}} & A_{P\text{N}} \end{bmatrix} \begin{Bmatrix} \phi_1 \\ \phi_2 \\ \vdots \\ \phi_\text{N} \end{Bmatrix} = \begin{Bmatrix} S_1 \\ S_2 \\ \vdots \\ S_\text{N} \end{Bmatrix} \tag{3}$$

where the numerical subscript of each entry is referred to the number of the respective FV (row number).
In compact form:

$$\mathbf{A\Phi} = \mathbf{S} \tag{4}$$

- We note from eq. (3) that the coefficient matrix **A** is *tridiagonal*.

Data organization

- Considering the structure of eq. (3), it is appropriate to store the coefficients A_{Wi}, A_{Pi}, A_{Ei} and source terms S_i ($i = 1, \ldots, N$) in column vectors D_W, D_P, D_E and S ($N \times 1$ matrices):

```
% Preparation of the 3 diagonals of A, stored as (column) vectors
D_W = A_W * ones( N , 1 ) ;
D_P = A_P * ones( N , 1 ) ;
D_E = A_E * ones( N , 1 ) ;

% RHS vector preparation (midpoint second order integration)
S = s( X ) * dx ;
```

  where the user defined function s(·) computes the source term $s(\cdot)$; therefore s(X) is the column vector of source term s evaluated at the FV centroids abscissæ X.

Boundary conditions

To impose BC at the boundary locations, we must correct the coefficients and source term for the first and last FV using a *ghost cell*:

- $x = 0$ (FV #1), *Dirichlet* BC:

$$\phi(0) = \phi_{BC} \quad \Rightarrow \quad \frac{\phi_W + \phi_P}{2} = \phi_{BC} \quad \Rightarrow \quad \phi_W = 2\phi_{BC} - \phi_P$$

  Replacing ϕ_W in eq. (2) gives:

$$A_W(2\phi_{BC} - \phi_P) + A_P\phi_P + A_E\phi_E = S_P \qquad (5)$$

Boundary conditions

Rearranging eq. (5) gives:

$$\underbrace{(A_P - A_W)}_{A_{P1}} \phi_P + A_E \phi_E = \underbrace{S_P - 2A_W \phi_{BC}}_{S_1} \tag{6}$$

```
% Diagonals correction in order to impose BC
%   Left  end ( x=0, i=1 ) (Dirichlet)
D_P( 1 ) = D_P( 1 ) - D_W( 1 ) ;
  S( 1 ) =   S( 1 ) - 2 * D_W( 1 ) * phi_BC ;
D_W( 1 ) = 0 ;
```

The last code line assigns a null value to the A_{W1} coefficient: this is not strictly needed, but it's formally correct because the BC has been imposed and A_{W1} value is no longer needed.

Boundary conditions

- $x = L$ (FV $\#N$), *Neumann* BC:

$$\Gamma \phi_x(L) = J''_{d,BC} \quad \Rightarrow \quad \Gamma \frac{\phi_E - \phi_P}{\Delta x} = J''_{d,BC} \quad \Rightarrow$$

$$\Rightarrow \quad \phi_E = \phi_P + \frac{\Delta x J''_{d,BC}}{\Gamma}$$

Replacing ϕ_E in eq. (2) gives:

$$A_W \phi_W + A_P \phi_P + A_E (\phi_P + \frac{\Delta x J''_{d,BC}}{\Gamma}) = S_P \tag{7}$$

Boundary conditions

Rearranging eq. (7) gives:

$$A_W \phi_W + \underbrace{(A_P + A_E)}_{A_{P\mathrm{N}}} \phi_P = \underbrace{S_P - A_E \frac{\Delta x J''_{d,BC}}{\Gamma}}_{S_\mathrm{N}} \tag{8}$$

```
%   Right end ( x=L, i=N ) (Neumann)
D_P( N ) = D_P( N ) + D_E( N ) ;
  S( N ) =   S( N ) - D_E( N ) * dx * Jd_BC / Gamma ;
D_E( N ) = 0 ;
```

Again, in the last code line we assign a null value to $A_{E\mathrm{N}}$ because the BC has been imposed and $A_{E\mathrm{N}}$ is no longer needed.

Sparsity

- At this point we have all the variables needed to build the final matrix system, eq. (3);
- As noted before, the coefficient matrix **A** is *tridiagonal*, i.e., only the main diagonal and the first lower and upper diagonals have non null entries;
- This particular *sparse* structure implies that the number of non null entries is proportional to N (in this case exactly $3N-2$), while the total number of formal entries of **A** is N^2;
- The solution of a linear system with N unknowns requires, in general, a number of operations (and thus a time consumption) proportional to N^3, while specific sparse solvers can reach an $\mathcal{O}(N)$ computational cost.

Sparse matrices

- For example, if we choose $N = 20000$, the memory size required to naively store $\mathbf{A}$ would be $8N^2$ bytes $= 3.2$ GB with double precision format, while the size of non null entries is only $8 \cdot 3N$ bytes $= 480$ kB;
- A naive resolution of this system would take approximately 1 minute on a modern PC, while a generic sparse solver requires less than 1 ms;
- It is clear that we must take advantage of the sparsity pattern of $\mathbf{A}$ to get acceptable computational resources consumption;
- MATLAB® can efficiently and easily handle sparse matrices, offering a wide set of elementary sparse operations and, overall, sparse solvers;
- Let's see how to handle these particular matrices for our specific problem.

Building sparse banded matrices with `spdiags()`

- We recognized the *tridiagonal* and *sparse* nature of coefficient matrix $\mathbf{A}$, eq. (3);
- Since these properties are particularly important and frequent in numerical problems, MATLAB® offers the `spdiags()` function that allows to build sparse banded matrices from the diagonals:

```
A = spdiags( D , d , m , n )
```

`A:`	sparse `m`×`n` banded matrix;
`D:`	matrix whose columns are the diagonals of `A`;
`d:`	vector of the shifts of each diagonal from the main diagonal.

Building sparse banded matrices with `spdiags()`

- Let's see how `spdiags()` works in practice with the `D_W`, `D_P` and `D_E` coefficients vectors (diagonals) of our 1D problem; since we want a square `N`×`N` matrix `A`, `m=n=N`:

```
% Concatenation of the (column) vectors of the diagonals of A
D = [ D_W D_P D_E ] ;

% Diagonals shifts from main diagonal
%   D_W is shifted -1 because it's the first lower diagonal
%   D_P is shifted  0 because it's the main diagonal
%   D_E is shifted  1 because it's the first upper diagonal
d = [ -1   0   1  ] ;

% spdiags() call
A_bad = spdiags( D , d , N , N ) ;
```

Building sparse banded matrices with `spdiags()`

- The previous call to `spdiags()` will produce the following (sparse) matrix `A_bad`:

$$\texttt{A\_bad} = \begin{bmatrix} A_{P1} & A_{E2} & & 0 \\ A_{W1} & \ddots & \ddots & \\ & \ddots & \ddots & A_{E\text{N}} \\ 0 & & A_{W(\text{N-1})} & A_{P\text{N}} \end{bmatrix} \qquad (9)$$

- `A_bad` looks like the coefficient matrix **A** of eq. (3), except for the ordering of the lower and upper diagonals: the upper diagonal starts with the 2nd entry (A_{E2}) while it should start with the 1st (A_{E1}); conversely, the lower diagonal starts with the 1st entry (A_{W1}) while it should start with the 2nd (A_{W2}).

Building sparse banded matrices with `spdiags()`

- To fix this incorrect ordering of the diagonals, we can shift the elements in `D_W` and `D_E` vectors before the `spdiags()` call, using for example the `circshift()` command which circularly shifts the elements of a matrix:

  ```
  S_M = circshift( M , s , dim )
  ```

`S_M:`	shifted matrix;
`M:`	matrix to be shifted;
`s:`	integer shift;
`dim:`	dimension along which the shift is done.

  If the matrix `M` to be shifted is a column vector (that's our case) we can omit the third argument `dim` $(= 1)$.

Building sparse banded matrices with `spdiags()`

- We use `circshift()` with shift `s=-1` for `D_W` because a negative shift is needed for the lower diagonal, while a positive shift `s=1` is needed for the upper diagonal `D_E`:

```
% Lower diagonal (West) and upper diagonal (East) shift.
D_W = circshift( D_W ,  -1 ) ;
D_E = circshift( D_E ,   1 ) ;
```

- Final (correct) `spdiags()` call:

```
% Concatenation of the (column) vectors of the diagonals of A
D = [ D_W D_P D_E ] ;

% Diagonals shifts from main diagonal
d = [ -1   0   1  ] ;

% spdiags() call to build sparse tridiagonal A
A = spdiags( D , d , N , N ) ;
```

Remarks

- In our specific case where the A_W, A_P and A_E coefficients are constants for each FV, the shift correction with `circshift()` could be avoided (`D_W` and `D_E` have constant entries), but it's formally correct;
- The circular property of `circshift()` is not strictly needed because the element being circularly shifted is anyway unused by `spdiags()`;
- The shift correction with `circshift()` could be anyway avoided inverting the diagonals order and taking the transpose (`'`):

```
% Concatenation of the (column vectors) diagonals of A, inverse order
D = [ D_E D_P D_W ] ;

% Diagonals shifts from main diagonal
d = [ -1   0   1  ] ;

% spdiags() call with transpose (')
A = spdiags( D , d , N , N )' ;
```

Solution phase

- At this point we have the (sparse) coefficient matrix **A** and the source vector **S**, therefore we can calculate the solution vector $\mathbf{\Phi}$ with the backslash \ operator:

```
% Direct solution (backslash \): A * phi = S => phi = A\S
phi = A\S ;
```

- The \ operator is a powerful MATLAB® function that allows the direct solution of linear systems. It automatically checks for the input matrix nature (square/not square, dense/sparse, diagonal/tridiagonal/banded, Hermitian/non-Hermitian, real/complex, etc.) to choose the appropriate direct solver;
- We'll focus on the use of iterative solvers in the last section.

Solution plot

- To graphically display the solution vector `phi` versus the `X` abscissæ vector we can use the `plot()` function:

```
% Solution plot
plot( X , phi ) ;
```

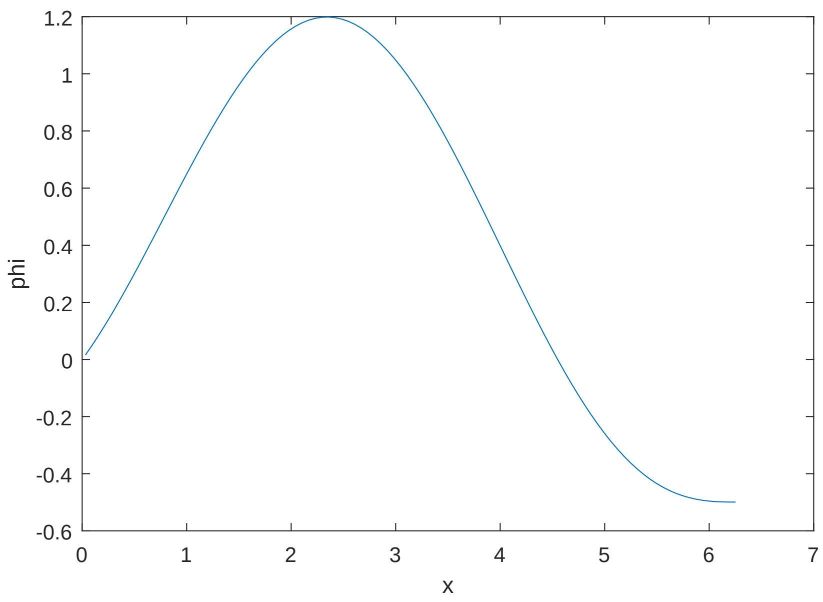
- We can add labels to the axes:

```
% Naming axes
xlabel( 'x' ) ;
ylabel( 'phi' ) ;
```

- There's a large set of tunable properties for the graphics objects (Figures, Axes, Lines, etc.) to get aesthetically pleasant graphic plots.

Solution plot

- For example, our solution with source term $s = \sin(x)$ looks like this:



2D advection-diffusion problem

- 2D steady-state advection-diffusion equation:

$$\nabla \cdot (\rho \mathbf{w} \phi) = \nabla \cdot (\Gamma \nabla \phi) + s \tag{10}$$

  with constant properties ρ, Γ and constant velocity $\mathbf{w}$;
- Domain:

$$\Omega = [0, L] \times [0, L]$$

- Boundary conditions (BC):

Type	*Dirichlet* BC	*Neumann* BC
Equation	$\phi = \phi_{BC}$	$\Gamma d\phi/dn = J''_{d,BC}$
Location	$y = 0$ (south)	$y = L$ (north)
(side)		$x = 0$ (west)
		$x = L$ (east)

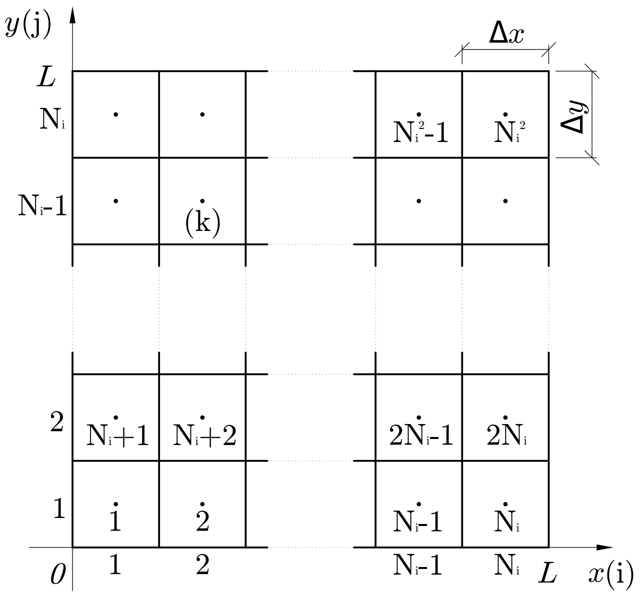
Variables

- Variables definition:

```
% Constant properties (rho, gamma) and advection velocity (w)
rho   = 1 ;
Gamma = 1 ;
w_x   = 0 ;
w_y   = 1 ;

% Domain square side length
L = 2*pi ;

% BC values
phi_BC = 0 ;
Jd_BC  = 0 ;
```

Domain discretization

- Discretization of the domain Ω in $N = N_i^2$ finite volumes (N_i for each dimension) with constant side length $\Delta x = \Delta y = L/N_i$:



Indexing

In 2D cases it's useful to employ two different indexing types for the discrete variables:

- *Spatial indexing* $(i, j = 1, \dots, N_i)$
  Two indexes are employed, one for each dimension: index *i* along the *x* dimension and index *j* along the *y* dimension.
  This indexing type is useful for the calculation of equation coefficients taking account of boundary conditions;
- *Linear indexing* $(k = 1, \dots, N)$
  One single index *k* spans the whole set of FV.
  This indexing type is needed in the building phase of the final equation system.

Indexing

- There's a one-to-one correspondence for these indexing types:

$$k = i + N_i(j-1)\ ; \qquad \begin{cases} i = 1 + (k-1) \bmod N_i \\ j = 1 + (k-i)/N_i \end{cases}$$

- However, we can ignore the previous formulas because we can implicitly pass from one indexing to the other using `reshape()` command:

```
M_k   = reshape( M_ij , N , 1 )
M_ij  = reshape( M_k , Ni , Ni )
```

`M_k:`	`N`×`1` column vector (linear indexing);
`M_ij:`	`Ni`×`Ni` matrix (spatial indexing);
`N, Ni:`	N, N_i (total FV # and FV # for each dimension).

Domain discretization

- Domain discretization variables:

```
% FV number for each dimension and FV total number
Ni = 100 ;
N  = Ni * Ni ;

% FV side length
dx = L / Ni ;

% FV 'volume' (surface)
dA = dx * dx ;

% FV centroids coordinates
X = dx * ( (1:Ni) - 0.5 )' ;
Y = X ;
```

Equation discretization

- FV discretization of eq. (10) with 2nd order CDS and midpoint integration approximation for both diffusive and advective fluxes:

$$A_P\phi_P + A_E\phi_E + A_W\phi_W + A_N\phi_N + A_S\phi_S = S_P \tag{11}$$

```
% FV equation coefficients (constants for each FV)
A_E =  rho * dx * w_x / 2 - Gamma ;
A_W = -rho * dx * w_x / 2 - Gamma ;
A_N =  rho * dx * w_y / 2 - Gamma ;
A_S = -rho * dx * w_y / 2 - Gamma ;
A_P = -( A_E + A_W + A_N + A_S ) ;
```

- Eq. (11) holds for each of the N finite volumes except for the boundary FV (FV with at least one side lying on the boubdary), where this equation must be corrected because some of the neighbour cells don't exist: BC will be imposed using a *ghost cell*.

Equation discretization

- Writing eq. (11) for each of the N finite volumes, in a matrix notation we have:

$$\begin{bmatrix} \mathbf{A}_1^{\mathrm{WPE}} & \mathbf{A}_1^{\mathrm{N}} & & 0 \\ \mathbf{A}_2^{\mathrm{S}} & \ddots & \ddots & \\ & \ddots & \ddots & \mathbf{A}_{N_i-1}^{\mathrm{N}} \\ 0 & & \mathbf{A}_{N_i}^{\mathrm{S}} & \mathbf{A}_{N_i}^{\mathrm{WPE}} \end{bmatrix} \begin{Bmatrix} \phi_1 \\ \phi_2 \\ \vdots \\ \phi_N \end{Bmatrix} = \begin{Bmatrix} S_1 \\ S_2 \\ \vdots \\ S_N \end{Bmatrix} \tag{12}$$

where the numerical subscript j of each $\mathbf{A}_j$ entry in the coefficient matrix is referred to the j-th row of FV elements .
- We note from eq. (12) that the coefficient matrix is *block tridiagonal*.

Equation discretization

- Explicitly, the matrix entries $\mathbf{A}_j$ of the coefficient matrix have the following form (using spatial indexing for the coefficients):

$$\mathbf{A}_j^{\mathrm{WPE}} = \begin{bmatrix} A_{P(1,j)} & A_{E(1,j)} & & 0 \\ A_{W(2,j)} & \ddots & \ddots & \\ & \ddots & \ddots & A_{E(N_i-1,j)} \\ 0 & & A_{W(N_i,j)} & A_{P(N_i,j)} \end{bmatrix} \tag{13}$$

$$\mathbf{A}_j^{\mathrm{S}} = \begin{bmatrix} A_{S(1,j)} & & 0 \\ & \ddots & \\ 0 & & A_{S(N_i,j)} \end{bmatrix} \qquad \mathbf{A}_j^{\mathrm{N}} = \begin{bmatrix} A_{N(1,j)} & & 0 \\ & \ddots & \\ 0 & & A_{N(N_i,j)} \end{bmatrix}$$

Data organization

- In the 2D case it's natural to store the coefficients $A_{P(i,j)}$, $A_{E(i,j)}$, $A_{W(i,j)}$, $A_{N(i,j)}$, $A_{S(i,j)}$ and source terms $S_{(i,j)}$ in $N_i \times N_i$ matrices `D_P`, `D_E`, `D_W` , `D_N`, `D_S`, and `S` (spatial indexing):

```
% Preparation of the 5 diagonals of A, stored as Ni x Ni matrices
% (spatial indexing)
D_W = A_W * ones( Ni , Ni ) ;
D_E = A_E * ones( Ni , Ni ) ;
D_N = A_N * ones( Ni , Ni ) ;
D_S = A_S * ones( Ni , Ni ) ;
D_P = A_P * ones( Ni , Ni ) ;

% RHS preparation (midpoint second order integration)
% stored as Ni x Ni matrix (spatial indexing)
S = s( X , Y ) * dA ;
```

Again, `s( X , Y )` is the $N_i \times N_i$ matrix of source term s evaluated at FV centers.

Boundary conditions

To impose BC at the boundary sides, we must correct the coefficients and source term for the boundary FV using *ghost cells*, in the same way we did for the 1D case, eqs. (5)-(8).

- $y = 0$, FV on south side $(j = 1)$, *Dirichlet* BC:

```
% Diagonals correction in order to impose BC
%   South side ( y=0, j=1 ) (Dirichlet)
D_P( : , 1 ) = D_P( : , 1 ) - D_S( : , 1 ) ;
  S( : , 1 ) =   S( : , 1 ) - 2 * D_S( : , 1 ) * phi_BC ;
D_S( : , 1 ) = 0 ;
```

As we can see, the implementation of BC with spatial indexing is straightforward because we can access the variables with two separate space indexes.

Boundary conditions

- $y = L$, FV on north side $(j = N_i)$, *Neumann* BC:

```
%   North side ( y=L, j=Ni ) (Neumann)
D_P( : , Ni ) = D_P( : , Ni ) + D_N( : , Ni ) ;
  S( : , Ni ) =   S( : , Ni ) - D_N( : , Ni ) * dx * Jd_BC / Gamma ;
D_N( : , Ni ) = 0 ;
```

- $x = 0$, FV on west side $(i = 1)$, *Neumann* BC:

```
%   West side ( x=0, i=1 ) (Neumann)
D_P( 1 , : ) = D_P( 1 , : ) + D_W( 1 , : ) ;
  S( 1 , : ) =   S( 1 , : ) - D_W( 1 , : ) * dx * Jd_BC / Gamma ;
D_W( 1 , : ) = 0 ; % <= this is very important!!
```

- $x = L$, FV on east side $(i = N_i)$, *Neumann* BC:

```
%   East side ( x=L, i=Ni ) (Neumann)
D_P( Ni , : ) = D_P( Ni , : ) + D_E( Ni , : ) ;
  S( Ni , : ) =   S( Ni , : ) - D_E( Ni , : ) * dx * Jd_BC / Gamma ;
D_E( Ni , : ) = 0 ; % <= this is very important!!
```

Reshaping

- From eqs. (12)-(13) we note the *pentadiagonal* structure of coefficient matrix **A**, thus we can still use `spdiags()` to build **A** from its diagonals `D_S`, `D_W`, `D_P` , `D_E` and `D_N` that we just calculated.
  Since `spdiags()` needs linear indexed diagonals, we use the `reshape()` command:

```
% Diagonals (and RHS) reshape to get vectors (linear indexing)
D_P = reshape( D_P , N , 1 ) ;
D_W = reshape( D_W , N , 1 ) ;
D_E = reshape( D_E , N , 1 ) ;
D_N = reshape( D_N , N , 1 ) ;
D_S = reshape( D_S , N , 1 ) ;
S   = reshape(   S , N , 1 ) ;
```

- In the last code line we also reshaped the source vector `S` because the final system needs also a linear indexed RHS.

Shifting

- As in the 1D case, diagonal shifting is needed to get the correct ordering of upper and lower diagonals before the `spdiags()` call:

```
% Lower diagonals (South/West) and upper diagonals (East/North)
% shifting
D_S = circshift( D_S , -Ni ) ;
D_W = circshift( D_W ,  -1 ) ;
D_E = circshift( D_E ,   1 ) ;
D_N = circshift( D_N ,  Ni ) ;
```

- The shift value used in `circshift()` equals the diagonal shift from the main diagonal: for `D_W` and `D_E` we have -1 and 1 (just as in the 1D case), while for `D_S` and `D_N` we have $-N_i$ and N_i.

`spdiags()` call

- At this point we have the 5 diagonals with correct indexing and correct ordering, so we can call `spdiags()`:

```
% Concatenation of the (column) vectors of the diagonals of A
D = [ D_S D_W D_P D_E D_N ] ;

% Diagonals shifts from main diagonal
d = [ -Ni  -1   0   1  Ni ] ;

% spdiags() call to build sparse pentadiagonal A
A = spdiags( D , d , N , N ) ;
```

- As we can see, the diagonals shifts from the main diagonal (`d`) are the same used in `circshift()` in the previous shifting phase.
- Now we have sparse $N \times N$ pentadiagonal coefficient matrix **A** and $N \times 1$ source term vector **S**, thus we can proceed to the solution phase.

Remarks

- The shifting operation on `D_W` and `D_E` diagonals with `circshift()`, together with the null assignation to the same diagonals in the BC imposition on west and east sides, is crucial in 2D case even if the coefficients A are the same for every FV;
- Again, the circular property of `circshift()` is not strictly needed;
- The shift correction with `circshift()` could also be avoided inverting the diagonals order and taking the transpose (`'`):

```
% Concatenation of the (column vectors) diagonals of A, inverse order
D = [ D_N D_E D_P D_W D_S ] ;

% Diagonals shifts from main diagonal
d = [ -Ni  -1   0   1  Ni ] ;

% spdiags() call with transpose (')
A = spdiags( D , d , N , N )' ;
```

Solution phase

- We can now calculate the solution vector $\mathbf{\Phi}$ in the same way as in the 1D case, using the backslash $\backslash$ operator:

```
% Direct solution (backslash \): A * phi = S => phi = A\S
phi = A\S ;
```

- Alternatively, we can compute an approximate solution using one of the MATLAB® built-in iterative solvers (`pcg`, `minres`, `symmlq`, etc.);
- For example, we can use the Preconditioned Conjugate Gradient method (`pcg`) with an Incomplete Cholesky factorization (`ichol`) as preconditioner:

```
% PCG solution with Incomplete Cholesky factorization preconditioner
tol    = relative tolerance on residual ;
n_iter = maximum number of iterations ;
L      = ichol( A ) ;
phi = pcg( A , S , tol , n_iter , L , L' ) ;
```

Solution visualization: contour lines

- For the graphical visualization of the solution, we need a spatial indexed solution `phi` ($N_i \times N_i$ matrix), thus we use `reshape()`:

```
% Solution reshape to get again the spatial indexing
phi = reshape( phi , [ Ni , Ni ] ) ;
```

- We can display solution contour lines using `contourf()` command, which also fills the spaces with solution related colours (note the transpose (`'`) on `phi` to get the right axes order):

```
% Contour line plot (orthogonal to the diffusive flux)
contourf( X , Y , phi' ) ;

% Naming axes
xlabel( 'x' ) ;
ylabel( 'y' ) ;
```

Solution visualization: 3D surface plot

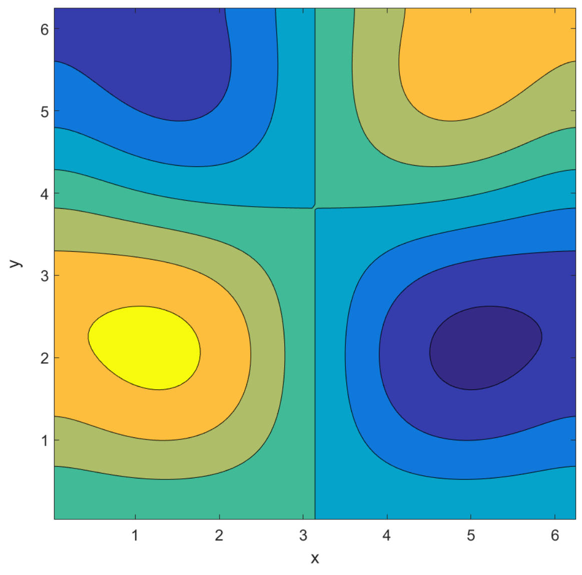
- We can also display a 3D solution surface using `surf()` command, which also colours the surface with solution related colours (again, note the transpose (`'`) on `phi` to get the right axes order):

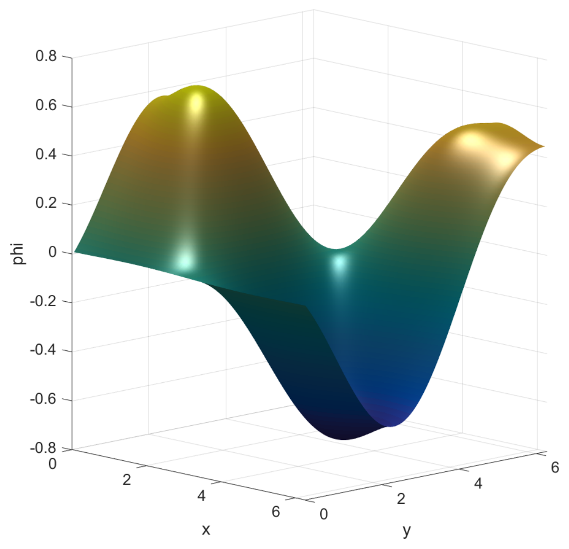
```
% Surface plot on figure 2
surf( X , Y , phi' ) ;

% Naming axes
xlabel( 'x' ) ;
ylabel( 'y' ) ;
zlabel( 'phi' ) ;
```

- With 3D surfaces it's useful to tune some graphic properties (Edge Colors, Face Lighting, Lights, etc.) to get pretty and more understandable plots.

Solution visualization

- For example, with source term $s = \sin(x)\sin(y)$ and some graphical tuning, our solution looks like this:





Iterative methods

- In 2D and, overall, 3D cases where a large number of unknowns N is employed, generic direct solvers may be too expensive in terms of both time and memory consumption;
- Furthermore, in many practical applications we can accept a “good” solution, i.e., an approximation of the exact (discretized) solution with a “moderate” error (convergence error);
- Under these circumstances, we can use *iterative solvers* to get an approximate solution in reasonable time;
- Most iterative solvers are *matrix-free*: they don’t require the explicit storage of coefficient matrix $\mathbf{A}$;
- We’ll focus on *Jacobi* and SOR (*Successive Over-Relaxation*) methods.

2D Poisson problem

- In the implementation of iterative solvers we’ll focus on a 2D *Poisson* equation:

$$\nabla^2 \phi = s \tag{14}$$

on a square domain Ω:

$$\Omega = [0, L] \times [0, L]$$

with *Neumann* boundary conditions[1] on the whole boundary (4 sides):

$$\nabla \phi \cdot \mathbf{n} = \frac{\partial \phi}{\partial n} = 0 \tag{15}$$

- Problem (14)-(15) is very important in CFD: for example using *Projection* methods, ϕ is the *correction pressure* for Navier-Stokes equations.

[1] An additional condition is required: for example $\phi = 0$ in some point;

Domain discretization

- We discretize the domain Ω in the same way as in the previous 2D case, with N_i finite volumes for each dimension:

```
% Domain square side length
L = 2*pi ;

% FV number for each dimension
Ni = 100 ;

% FV side length
dx = L / Ni ;

% FV 'volume' (surface)
dA = dx * dx ;

% FV centroids coordinates
X = dx * ( (1:Ni) - 0.5 )' ;
Y = X ;
```

Equation discretization

- FV discretization of eq. (14) with 2nd order CDS and midpoint integration approximation for diffusive flux:

$$A_P\phi_P + A_E\phi_E + A_W\phi_W + A_N\phi_N + A_S\phi_S = S_P \tag{16}$$

```
% FV equation coefficients (constants for each FV)
A_E =  1 ;
A_W =  1 ;
A_N =  1 ;
A_S =  1 ;
A_P = -4 ;
```

- Eq. (16) holds for every FV, even for the boundary FV where coefficient corrections are not needed anymore: BC will be enforced using *ghost cells* in a direct way.

Data organization

- We store the solution `phi` as a $(N_i + 2) \times (N_i + 2)$ matrix (initialized with null values) because we need to store the ghost cells values explicitly, with spatial indexing:

```
% Solution matrix (spatial indexing + ghost cells)
phi = zeros( Ni+2 , Ni+2 ) ;
```

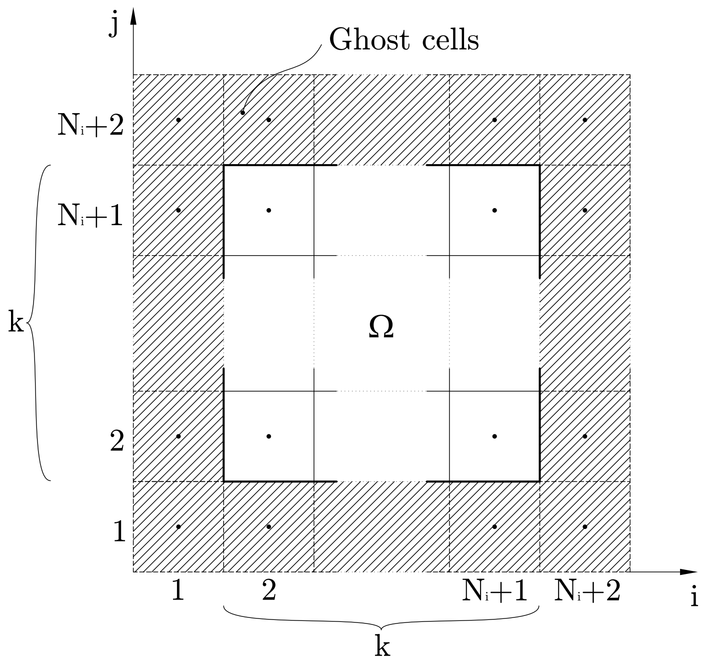
- We also need the vector `k` for the indexing in `phi`:

```
% Indexes of FV inside the domain (not ghost cells)
k = 2 : (Ni+1) ;
```

- For example, `phi(k,k)` is the $N_i \times N_i$ matrix of all ϕ values for the FV inside the domain.

Data organization

- Graphical representation of solution matrix `phi`:



Data organization

- The ghost cells values have the following access indexes:

Side	Ghost cells
South	`phi(k,1)`
West	`phi(1,k)`
North	`phi(k,Ni+2)` or `phi(k,end)`
East	`phi(Ni+2,k)` or `phi(end,k)`

- The source term is also spatial indexed and calculated with the classic midpoint second order integration approximation:

```
% Source term (midpoint second order integration)
% Ni x Ni matrix (spatial indexing)
S = s( X , Y ) * dA ;
```

Iterative cycle

- The structure of the iterative cycle is the following:

```
% Number of iterations
n_iter = 1000 ;

% Iterative cycle
for iter = 1 : n_iter
  UPDATE_SOLUTION() ;
  UPDATE_GHOST_CELLS_VALUES() ;
end
```

- The function `UPDATE_SOLUTION()` updates the solution values in `phi` using a specific iterative method (Jacobi or SOR);
- The function `UPDATE_GHOST_CELLS_VALUES()` only updates the ghost cells values in `phi` in order to enforce BC in a direct way.

Jacobi method

- Jacobi method updates each unknown from the previous iteration values:

$$\phi_P^{n+1} = \frac{S_P - A_E\phi_E^n - A_W\phi_W^n - A_N\phi_N^n - A_S\phi_S^n}{A_P} \tag{17}$$

- Since this method only requires ϕ^n values at previous iteration n, we can implement this iteration in a matrix way:

  `UPDATE_SOLUTION():`

```
% Jacobi iteration
ngbrs = A_E*phi( k+1 , k ) + A_W*phi( k-1 , k ) + ...
        A_N*phi( k , k+1 ) + A_S*phi( k , k-1 ) ;
phi( k , k ) = ( S - ngbrs ) / A_P ;
```

Jacobi method

- The previous matrix assignations used the vector `k` as index to access the neighbor cells in `phi`;
- For example, `phi(k+1,k)` is the $N_i \times N_i$ matrix of all ϕ_E values, because the first vectorial index is `k+1`;
- We first calculate the $N_i \times N_i$ matrix `ngbrs` which contains the sum of the 4 neighbor matrices multiplied by their respective coefficient A; then we update the solution `p(k,k)` ($N_i \times N_i$ matrix);
- The use of matrix assignations is efficient (no explicit `for` loops).

SOR method

- Successive Over-Relaxation (SOR) method updates each unknown using the updated values as soon as they're available, with an *over-relaxation* step to accelerate the convergence rate:

$$\begin{aligned}\phi_P^{n+1} &= (1-\omega)\phi_P^n + \\ &+ \omega\frac{S_P - A_E\phi_E^n - A_W\phi_W^{n+1} - A_N\phi_N^n - A_S\phi_S^{n+1}}{A_P}\end{aligned} \tag{18}$$

- Since this method requires ϕ^{n+1} values at new iteration $n+1$ as soon as they're available, we can't implement this iteration in a matrix way;
- We have to explicitly write some `for` loops spanning over the solution matrix `phi`.

SOR method

- The SOR code is therefore the following:

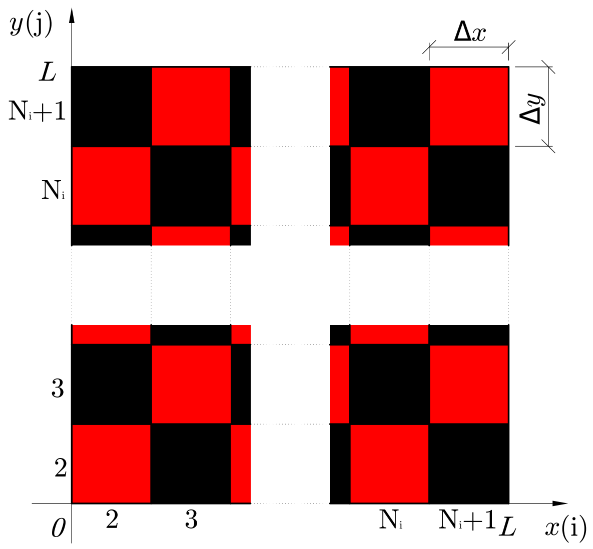
`UPDATE_SOLUTION()`:

```
% SOR over-relaxation parameter
w = 1.6 ;

% SOR iteration
for i = k
  for j = k
    ngbrs = A_E*phi( i+1 , j ) + A_W*phi( i-1 , j ) + ...
            A_N*phi( i , j+1 ) + A_S*phi( i , j-1 ) ;
    phi(i,j) = w*( S(i-1,j-1) - ngbrs )/A_P + (1-w)*phi(i,j) ;
  end
end
```

- We used two `for` loops to span over the FV in the domain (`i` and `j` assume all the values in `k`);
- In this case we update `phi` values one by one, not in a matrix way;
- The source term `S` has to be accessed with `i-1` and `j-1` because it doesn't have ghost cells.

Red-Black Ordering SOR

- SOR can be somehow vectorized considering the following cell ordering (*Red-Black ordering*):



  - From the side figure we note that every 'red' cell has only 'black' neighbors (W, E, N, S) and vice-versa;
  - Thus 'red' cells update process needs only 'black' cells and vice-versa, allowing a vectorization of the process.

Red-Black Ordering SOR

- Since neighbor cells have all been updated at the previous 'semi' iteration, we can over-relax the solution as we did with SOR method, eq. (18), to gain better convergence rate:

  UPDATE_SOLUTION():

```
% Indexes for Red-Black Ordering
si = [ 0 1 0 1 ] ;
sj = [ 0 1 1 0 ] ;

% Red-Black SOR iteration (Red cells: RB=1,2; Black cells: RB=3,4)
for RB = 1 : 4
  i = (2+si(RB)) : 2 : (Ni+1) ;
  j = (2+sj(RB)) : 2 : (Ni+1) ;
  ngbrs = A_E*phi( i+1 , j ) + A_W*phi( i-1 , j ) + ...
          A_N*phi( i , j+1 ) + A_S*phi( i , j-1 ) ;
  phi( i , j ) = w*( S(i-1,j-1) - ngbrs ) / A_P + (1-w)*phi(i,j) ;
end
```

- As we can see, we used only matrix assignations (as in Jacobi implementation) with over-relaxation (as in SOR).

Updating ghost cells

- When the solution update phase is complete, we must update ghost cells values to directly enforce BC; we proceed side by side:

  `UPDATE_GHOST_CELLS_VALUES()`:

```
% Updating 'ghost cells' values to enforce BC
%   South side ( y=0, j=1 ) (Neumann)
phi( k , 1 ) = phi( k , 2 ) ;

%   North side ( y=L, j=end ) (Neumann)
phi( k , end ) = phi( k , end-1 ) ;

%   West side ( x=0, i=1 ) (Neumann)
phi( 1 , k ) = phi( 2 , k ) ;

%   East side ( x=L, i=end ) (Neumann)
phi( end , k ) = phi( end-1 , k ) ;
```

Solution shift

- Since the Neumann BC (15) doesn't specify any ϕ value, and Poisson equation (14) is invariant under solution shiftings ($\phi + c$), we must enforce a fixed ϕ value on a point in the domain to get an unique solution, for example $\phi(0, 0) = 0$:

  `UPDATE_GHOST_CELLS_VALUES()` continuation:

```
% Null solution on first FV [phi(2,2) = phi(x=0,y=0) to II order
%                            because of BC]
phi = phi - phi(2,2) ;
```

Solution extraction and visualization

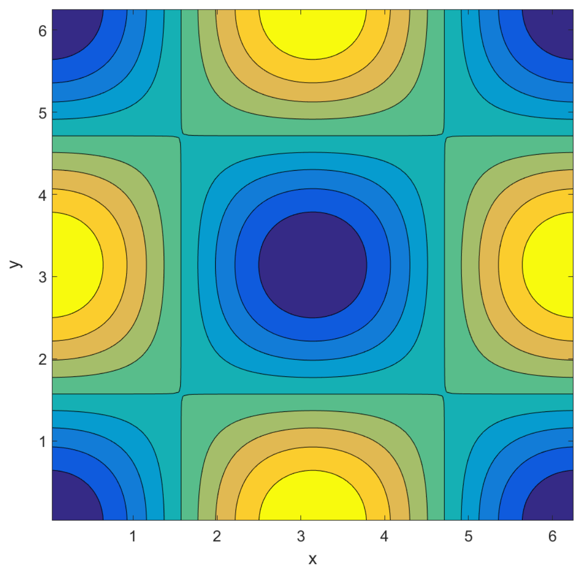
- At this point we accomplished the computation phase of the (approximate) solution of the discrete FV equations;
- If we're not interested in the ghost cells values, we can now extract the solution values for the FV inside the domain:

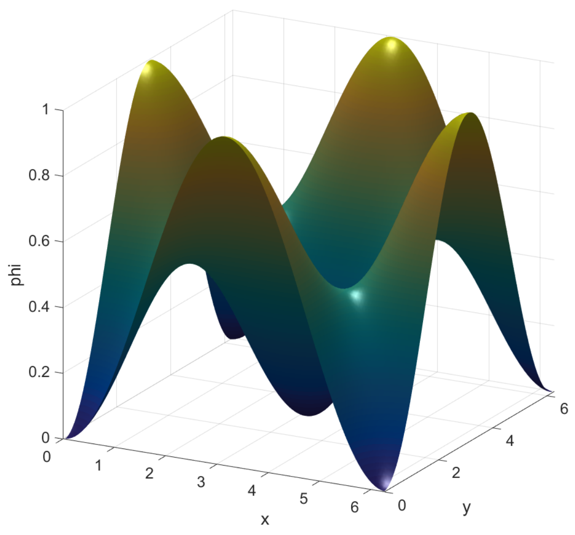
```
% Extraction of solution values for FV inside the domain
phi = phi( k , k ) ;
```

- Finally, we can display the solution as we did in the 2D advection-diffusion case using `contourf()` and `surf()` commands, for example.

Solution visualization

- For example, with source term $s = \cos(x)\cos(y)$, 5000 SOR ($\omega = 1.6$) iterations and some graphical tuning, our solution looks like this:




phi
x
y