Laboratory 1 - 10/10/16 - Solution

The matrix A can be built with the following code, based on the `for` loop:

```
n = 10;
h = 1/n;
dim = n-1;
A = zeros(dim,dim);
for i = 1:dim
   A(i,i) = 2;
end
for i = 1:dim-1
   A(i,i+1) = -1;
   A(i+1,i) = -1;
end

A = h^(-2) * A;
```

Another possible choice is to build the matrix using the command `diag`:

```
n = 10;
h = 1/n;
A = 2*diag(ones(n-1,1))-diag(ones(n-2,1),-1)-diag(ones(n-2,1),1);
A = h^(-2) * A;
```

where the command `diag(v,k)` builds a square matrix of size `length(v)+abs(k)`, with `v` an array and `k` an integer number. The k-th diagonal of matrix A is given by the array `v`. A zero value of `k` refers to the principal diagonal of `A`, whereas positive and negative values of `k` refers to the upper and lower diagonals of the matrix, respectively. We remark that the main diagonal of matrix A could be obtained also with the command `2*eye(n-1)`, where `eye(n-1)` produces the identity matrix of size $n-1 \times n-1$.

We remark that both the approaches are correct. The main difference consists in the computational time, since the second method is faster with respect to the first one. This is due to the fact that Matlab is an interpreted language that executes instructions directly, without previously compiling a program into machine-language instructions (read the instruction, run the instruction, go to the successive instruction). Therefore, the `for` loop does not perform well and is recommended only if necessary. Moreover, the second method is faster since it uses a “built-in” function, that is, a function already compiled that does not need to be interpreted. Therefore, we recommend using all the features of Matlab and all the “built-in” functions.

The memory consumption of the matrix A can be evaluated through the command `whos`. The results of the command is the following:

```
Name      Size                    Bytes  Class

  A         9x9                       648  double array
```

From the output, we notice that the matrix A is stored in double precision, including the zeros. This causes a substantial waste of memory when we consider bigger matrices. To avoid this, we can store the matrix in the *sparse* format, that consists in storing only the positions and the values of the matrix elements different from zero. To convert the matrix A in a sparse format we can use the command `sparse`:

```
A_sparse = sparse(A)
```

Now, the command `whos` will produce the following result:

```
   Name           Size                    Bytes  Class

  A              9x9                       648  double array
  A_sparse       9x9                       480  double array (sparse)
```

where we can notice a gain in memory of about 30%. Moreover, the benefit of storing the matrix in the sparse format increases when increasing the size of the matrix. Indeed, the number of elements stored in the classic format are on the order of n^2, while we can obtain a memory consumption on the order of $m = O(n)$ by using the sparse format.

The command `sparse` allows us to go from the *full* format to the *sparse* one, while the command `full` allows to go back to the full format. However, the aim is to write the matrix directly in the sparse format, since using the full format typically requires unnecessary and sometimes unavailable quantity of memory. To this aim, the command `spdiags` allows to directly create the matrix in sparse format. The syntax is `spdiags(B,v,n_row,n_col)`, where `B` is a matrix and `v` is a row array with integer numbers. The i-th element of v represents the diagonal of B (with the same rules used by the parameter `k` in `diag`) where we put the i-th column of the matrix B. The last two parameters, `n_row` and `n_col`, represent the number of rows and columns of the matrix B, respectively. We can then build the matrix A in sparse format with the following code:

```
n = 10;
h = 1/n;
e = ones(n-1,1);
A = h^(-2) * spdiags([2*e,-e,-e],[0,-1,1],n-1,n-1);
```

To prove that the matrix has the correct structure we can check the pattern of the matrix with the command `spy`, which will produce a picture containing a dot for every non-zero element of the matrix A:

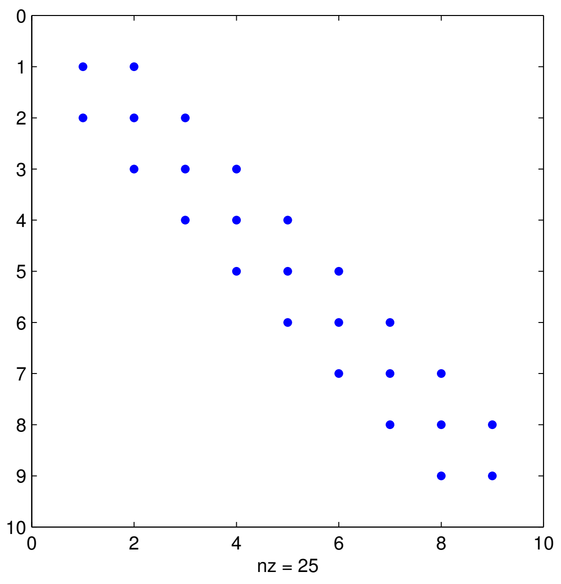

Figura 1: Pattern of the matrix A

The command `nnz(A)` will tell you how many non-zero elements there are in the matrix A (25 in this case).