AMS 595.03 MATLAB Part 2

Table of Contents

Author: Oliver Yang

For further reading, Ch. 2 of *Attoway* and Ch. 5 of *Lockhart/Tilleson* also cover some of the topics of this lecture.

Arrays

The Memory Hierarchy and Cache Performance

To return information about a variable, use the `whos` function:

```
whos( 'a')
```

When you double-click a variable in the Workspace or use Open Variable in the Variable tab of the Toolstrip, it opens the Variables window. Here you can manually edit the values of the variable in a spreadsheet.

Note that MATLAB calls the variable "a" a 1×1 double. This is because MATLAB, unlike most other languages, treats all objects as arrays.

Arrays are representations of mathematical objects called tensors. 0-dimensional arrays (rank-0 tensors) are **scalars**, 1-dimensional arrays (rank-1 tensor) are **vectors**, and 2-dimensional arrays (rank-2 tensors) are **matrices**.

MATLAB stands for "matrix laboratory." It was written by applied mathematicians to be used by applied mathematicians. MATLAB therefore takes matrices as its base data structure and treats scalars as 1×1 matrices, **column** vectors as $n \times 1$ matrices, and **row** vectors as $1 \times n$ matrices.

Arrays are useful when you want to store together large numbers of objects of the same type. The objects are stored in a **continguous** piece of memory. In MATLAB, you do not need to **allocate** or **deallocate** memory manually. In most other languages, you have to call functions to ask the operating system to give you a chunk of memory, e.g. in Fortran (allocate), C (malloc), and C++ (new), and deallocate the memory once you are finished using it. (Operating systems are covered in CSE 506.) In Python, the equivalents are NumPy arrays.

In scientific computing, data sizes can be very large, e.g. much larger than one million entries. In CAM or Bio, the data might be a state variable (density, momentum, energy) for fluid simulations at each vertex of a 3D mesh discretizing a physical domain. In Stat or QF, it could be samples in a massive genetic, social media, or financial data set. And in OR, it could be traffic through a national cellular network, or shipping data for a large online retailer like Amazon or Walmart.

The bottleneck in computing nowadays is not actually in the floating-point operations, it is in the data motion. Moving data has become much more expensive (both in terms of time and energy) than computation.

Reading from and writing to the hard disk is called **I/O** (input/output). I/O is extremely slow. You should avoid it whenever possible in scientific computing. Usually all the I/O is done at the beginning of a program and files are written to disk during the execution of the program only for the purposes of checkpointing or periodically saving the state of a time-dependent simulation.

Data from the disk are read into **RAM** (random access memory). Reading from and writing to RAM is much faster than the disk. RAM (which could be ~16GB on your computer) is much smaller than the disk (which could be ~1TB).

Blocks of data from RAM are then pulled into the **cache**. Cache, in turn, is much faster than RAM. Cache itself usually has a few levels, L1, L2, and L3. The hard-disk/RAM/cache chain is called the **memory hierarchy**. Data in cache are finally read into **registers** in the CPU, where the computations are actually done. Ideally, you want all the data that you need to be read in from cache, since that is fast. You don't want to have to go back to RAM to fetch another block of data, since that is slow. When the data that you need is found in cache, that is called a **cache hit**. When it is not found in cache, that is called a **cache miss**. You want to store your data in such a way that pieces of data that tend to be used together lie close to each other in the physical memory, so that you maximize cache hits and minimize cache misses.

The most important thing to worry about when optimizing code is **cache performance**. There are other data structures, such as **linked lists**, in which the data are not stored in continguous memory, but instead every piece of data could be located in a different part of the physical memory. Such an organization of the data could seriously hit the performance. For large problems, it could make the difference between your program finishing after a few hours or after a few days. In C/C++, you can analyze the cache performance of your program using a tool called *Cachegrind*, which you should ask Wenbin to teach you to use.

The relative costs of computation and data motion also extend to **supercomputers**, also called **clusters**. The number of transistors that can put onto a single wafer of silicon is reaching the physical limit. Chips nowadays force so much electricity into such a small space that they have greater a energy density than the surface of the sun. The only way to continue Moore's Law (that computing power roughly doubles every 16 months) is to group together many chips into a supercomputer.

Supercomputing, also known as **parallel computing** or **high-performance computing** (HPC), is the use of more than one independent processing unit to a perform a single computational task. In fact, you are most likely parallel processing right now on your mobile phone, tablet, laptop, or desktop. Most laptops and desktops 15 years ago still contained single-core processors (the speed of a single core has plateaued at about 2 GHz). Nowadays, most machines contain processors with more than one core. My MacBook Air, for example, has 4 cores that can do computations simultaneously.

Scientific simulations of any significant size will have to run on a cluster. Stony Brook has a cluster called Seawulf with ~5,000 cores that operates at ~180 teraflops (180×10^{12} flops) per second. The fastest machine in the world right now is called Summit, located at Oak Ridge National Laboratory (ORNL), which has ~200,000 cores and operates at ~200 petaflops (200×10^{15} flops) per second.

The 4 cores in my laptop share the same memory. That is, all cores read data from and write data to the same block of memory. The most common way to write programs for **shared memory** is called **OpenMP**. Shared memory programming is taught in CSE 613.

The number of cores that can share the same memory, however, is limited. To have more cores in a single machine, each core (or each small group of cores) needs to have its own memory. This is called **distributed memory** (or **heterogeneous** architectures). The cores (or group of cores) must then be linked through a **network topology** in order to pass data between them, since they do not all share the same memory. The most common way to write programs for distributed memory is called **MPI** (Message Passing Interface). MPI is taught in AMS 530.

The cost in running time and energy consumption in supercomputers is dominated by the communication between its cores, so much so, that computation essentially occurs "for free". The algorithm with the fastest running time and lowest energy cost will most likely be the one that minimizes the amount of communication. This has led to an entire field called **communication-avoiding algorithms**.

Defining, Accessing, and Resizing Vectors

An array is denoted by square brackets. Entries in the same row may be separated by spaces or by a comma. These are row-vectors:

```
[1 2]
```

```
ans = 1×2
     1     2
```

```
a = [single( 1), single( 2)]
```

```
a = 1×2 single row vector
     1     2
```

To signal that you want to move the next row, use a semicolon. This is a column-vector:

```
[single( 1); 2]
```

```
ans = 2×1 single column vector
     1
     2
```

Note that only one entry needs to be cast as single for all the entries to be.

It is also possible to create an empty array, which we will use later to delete rows and columns:

```
[]
```

```
ans =
     []
```

To access entries of a vector, use the vector name, follow by the **index** in parentheses:

```
a( 1)
```

```
ans = single

     1
```

```
a( 2)
```

```
ans = single

     2
```

Character arrays are accessed the same way:

```
b = 'abc'
```

```
b =
'abc'
```

```
b( 3)
```

```
ans =
'c'
```

Note that array indices start at 1 in MATLAB. This is also true of Fortran. C/C++, Java (which is based on C), and Python, however, have indices starting at 0. So `a( 1)` in MATLAB would be equivalent to `a[ 0]` in C. This always must be remembered when calling functions from other languages.

Accessing entries this way also allows you to write to that entry of the vector:

```
a( 1) = 2
```

```
a = 1×2 single row vector
     2     2
```

In C++, you will learn that this can be forbidden by using the `const` keyword.

Note that the double is cast as a single when assigned to a single array.

Trying to access an entry beyond the size of the array causes an error:

```
a( 3)
```

This is the most common type of memory bug that you will encounter. (We will start to use the debugger next week.) In compiled languages, this results in a **segmentation fault** when running an **executable**. You should ask Wenbin to teach you to use a tool called *Valgrind* with C/C++ to help catch these bugs.

In most languages, writing to an entry beyond the size of the array causes an error. In MATLAB, though, it does NOT cause an error! MATLAB will **dynamically resize** the array to accomodate for the **out-of-bounds** entry:

```
a( 3) = 3
```

```
a = 1×3 single row vector
     2     2     3
```

Entries in between, that are not specified, will be filled with 0's:

```
a( 3) = 5
```

```
a = 1×3 single row vector
     2     2     5
```

There are advantages and disadvantages to this approach. The disadvantage is that if you did not intend to write to an out-of-bounds entry (i.e. there is an indexing bug in your code), the bug will be much harder to catch, because it does not produce an error.

In scientific computing, you should avoid dynamic resizing whenever possible. It is much better to determine the sizes of all the arrays that you will need at the beginning of the program and **preallocate** memory for them at the beginning. The problem with dynamic resizing that what MATLAB is really doing in the background every time is creating a new larger array, copying the old array entries into the new one, and destroying the old one. When the array sizes are large, this is TERRIBLY inefficient.

Functions of Vectors

The `norm` function returns the Euclidean norm of the vector, also called the 2-norm:

```
norm( [1, 2, 3])
```

```
ans =
     3.741657386773941e+00
```

The value of p for the p-norm of a vector can be specified as the second input argument:

```
v = [1; 2; 3]
```

```
v = 3×1
     1
     2
     3
```

```
norm( v, 2)
```

```
ans =
     3.741657386773941e+00
```

Note that MATLAB functions allow optional input arguments. This is not allowed in some languages, such as Fortran, in which you must create functions with different names in order to have different inputs.

The function works for both row and column vector inputs.

The 1-norm of a vector is the sum of the absolute values of the entries:

```
norm( v, 1)
```

```
ans =
     6
```

The infinity-norm of a vector is the largest absolute value of its entries:

```
norm( v, Inf)
```

```
ans =
```

```
     3
```

The negative-infinity-norm of a vector is the smallest absolute value of its entries:

```
norm( v, -Inf)
```

```
ans =
     1
```

If you are taking AMS 526 this semester, the proof of some of these facts will be on your 526 homework.

Matrices

This is a matrix:

```
[ int32( 1), 2; 3, 4]
```

```
ans = 2×2 int32 matrix
   1   2
   3   4
```

It is good style to denote vectors by lower-case names and matrices by upper-case names.

Not having the correct number of entries produces an error:

```
       [1, 2; 3]
```

There are some commonly used built-in functions to create matrices:

The `zeros` function creates arrays with all entries equal to 0:

```
A = zeros( 2)
```

```
A = 2×2
     0     0
     0     0
```

Note that MATLAB does not use the standard English irregular plural "zeroes."

Functions that have a return value can be assigned to a variable.

Note that what MATLAB is doing above is **declaring** the array (allocating memory for it) and **initializing** all of its entries to 0. The initialization process takes time, especially if the array is large. Often, the entries do not need to be initialized, since they will be assigned later before they will be used. If you have MATLAB Coder installed and you will compile the MATLAB code into C code, you can only make the declaration and leave the entries uninitialized in the generated C code by using the `coder.nullcopy` function, which allocates memory for the size and type of array passed in as input:

```
A = coder.nullcopy( ones( 2));
```

Note that this creates a 2×2 matrix, not a 2×1 vector. To create a vector, you have to set one of the dimensions to be 1. The `ones` function creates arrays with all entries equal to 1:

```
ones( 2, 1, 'int32')
```

```
ans = 2×1 int32 column vector
   1
   1
```

The data type can be specified as a character array in the last input argument.

In languages such as Fortran, you must write separate routines for computations in single and double-precision. Usually this involves a lot of copy-and-pasting. But in MATLAB, you can define a variable to be passed into a function, which will determine the precision to be used:

```
data_type = 'single'
```

```
data_type =
'single'
```

```
A = rand( 1, 2, data_type)
```

```
A = 1×2 single row vector
   8.1472367e-01   9.0579194e-01
```

You can also create **multidimensional** arrays:

```
randn( 2, 2, 2)
```

```
ans =
ans(:,:,1) =
    -2.258846861003648e+00     3.187652398589808e-01
     8.621733203681206e-01    -1.307688296305273e+00
ans(:,:,2) =
    -4.335920223056836e-01     3.578396939725760e+00
     3.426244665386499e-01     2.769437029884877e+00
```

Note that MATLAB displays the array in 2D xy-slices, called **pages** in MATLAB.

```
randi( 10, 2)
```

```
ans = 2×2
     2    10
    10     5
```

The `rand` and `randn` functions are useful to create floating-point matrices to test on.

Use the `eye` function to create nxn identity matrices:

```
eye( 3)
```

```
ans = 3×3
```

```
     1     0     0
     0     1     0
     0     0     1
```

The `magic` function creates $n \times n$ magic squares (i.e. all the row-sums and column-sums are the same):

```
magic( 3)
```

```
ans = 3×3
     8     1     6
     3     5     7
     4     9     2
```

The `randi` and `magic` functions are useful to create integer matrices to test on.

The `hilb` function generates the $n \times n$ Hilbert matrix with $\frac{1}{i+j-1}$ as the (i, j)-th entry.

```
hilb( 3)
```

```
ans = 3×3
     1.000000000000000e+00     5.000000000000000e-01     3.333333333333333e-01
     5.000000000000000e-01     3.333333333333333e-01     2.500000000000000e-01
     3.333333333333333e-01     2.500000000000000e-01     2.000000000000000e-01
```

It is the classic example of a matrix whose entries have different scales, which makes it ill-conditioned.

In a Vandermonde matrix, the entries of successive columns of the matrix are successive powers of the corresponding entries of the original column. They often arise in numerical analysis. The `vander` function creates a Vandermonde matrix with the input vector as its last (original) column:

```
a = [1; 2; 3]
```

```
a = 3×1
     1
     2
     3
```

```
vander( a)
```

```
ans = 3×3
     1     1     1
     4     2     1
     9     3     1
```

The function works for both row and column vector inputs.

Toeplitz matrices have the same entry along each diagonal. They come up in various applications.
The `toeplitz` function returns the matrix with the first input argument as its first column vector and the second input argument as the row vector. The first element of the column and row vectors should be the same.

```
a = [0; 1; 2]
```

```
a = 3×1
```

```
     0
     1
     2
```

```
b = [0, -1, -2]
```

```
b = 1×3
     0    -1    -2
```

```
toeplitz( a, b)
```

```
ans = 3×3
     0    -1    -2
     1     0    -1
     2     1     0
```

Symmetric Toeplitz matrices only need the row vector as input:

```
toeplitz( a)
```

```
ans = 3×3
     0     1     2
     1     0     1
     2     1     0
```

Accessing Arrays

There are built-in functions that return information about the sizes of arrays:

```
A = [1, 2; 3, 4; 5, 6]
```

```
A = 3×2
     1     2
     3     4
     5     6
```

The `ndims` function returns the number of dimensions of the array. Trailing dimensions of size 1 are not counted:

```
ndims( A)
```

```
ans =
     2
```

The `size` function returns a row-vector of the sizes of the dimensions:

```
size( A)
```

```
ans = 1×2
     3     2
```

The desired dimension can be specified by a second input argument:

```
size( A, 1)
```

```
ans =
     3
```

The `length` function returns the size of the largest dimension:

```
length( A)
```

```
ans =
     3
```

The `numel` function returns the total number of elements in the array.

```
numel( A)
```

```
ans =
     6
```

To access entries of an array, you must specify the index for each dimension:

```
A = [1, 2; 3, 4]
```

```
A = 2×2
     1     2
     3     4
```

```
A( 1, 2)
```

```
ans =
     2
```

```
A( 2, 1)
```

```
ans =
     3
```

Multidimensional arrays are actually not stored in memory as such. They are **flattened** into a vector and stored that way. (Note that even though Fortran and C are interoperable, Fortran multidimensional arrays are not interoperable with C array-of-arrays.)

Following tradition, let's call the first dimension the "row" and the second dimension the "column". Then there are two ways to flatten a matrix:

1. Storing the columns together, which is called **column-major** or "first index changes fastest". This is done in MATLAB and Fortran. When A is flattened, it would be $(1, 3, 2, 4)$.
2. Storing the rows together, which is called **row-major** or "last index changes fastest". This is done in C/C++, Java (which is based on C), and Python. When A is flattened, it would be $(1, 2, 3, 4)$.

Since columns are stored together in the physical memory in MATLAB, using algorithms that operate on the rows, and that therefore need to pull rows into cache, can have a *very* negative impact on cache performance!

You can also use **linear indexing** to access entries (recalling that MATLAB is column-major):

```
A( 1)
```

```
ans =
     1
```

```
A( 2)
```

```
ans =
     3
```

```
A( 3)
```

```
ans =
     2
```

```
A( 4)
```

```
ans =
     4
```

There is one operator that we did not mention on Tuesday. To access a row or column, you can use the colon operator to specify the entire range:

```
A = magic( 5)
```

```
A = 5×5
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
```

```
A( 2, :)
```

```
ans = 1×5
    23     5     7    14    16
```

```
A( :, 3)
```

```
ans = 5×1
     1
     7
    13
    19
    25
```

```
A( :, :)
```

```
ans = 5×5
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
```

You can also use the colon to denote a user-specified range:

```
A( 2, 3 : 5)
```

```
ans = 1×3
     7    14    16
```

The `end` keyword denotes the final index.

```
A( 3 : end, 3)
```

```
ans = 3×1
    13
    19
    25
```

All the elements specified can also be assigned to another value:

```
A( 1 : 4, 1 : 2) = 0
```

```
A = 5×5
     0     0     1     8    15
     0     0     7    14    16
     0     0    13    20    22
     0     0    19    21     3
    11    18    25     2     9
```

You can also use the colon to create a row vector whose elements are evenly spaced. This will be important for loops. The first number is the first entry, the second number is the difference between entries, and the third number is the upper limit for the entries:

```
1 : 2 : 6
```

```
ans = 1×3
     1     3     5
```

Note that the upper limit may not be reached.

Also, the operands do not have to be integers:

```
pi : pi : 5 * pi
```

```
ans = 1×5
     3.141592653589793e+00     6.283185307179586e+00     9.424777960769379e+00 ···
```

This can also be achieved with the `linspace` function, with inputs for the first entry, the last entry, and the number of entries:

```
linspace( 1, 5, 3)
```

```
ans = 1×3
     1     3     5
```

There is also a function equivalent to the colon operator:

```
colon( 1, 4)
```

```
ans = 1×4
     1     2     3     4
```

```
colon( 1, 2, 6)
```

```
ans = 1×3
     1     3     5
```

The colon operator also exists in Fortran.

Reshaping Arrays

Arrays can be **concatenated** horizontally and vertically by using square brackets:

```
A = [1, 2; 3, 4]
```

```
A = 2×2
     1     2
     3     4
```

```
B = [5, 6; 7, 8]
```

```
B = 2×2
     5     6
     7     8
```

```
C = [5, 6, 7; 8, 9, 10]
```

```
C = 2×3
     5     6     7
     8     9    10
```

```
D = [5, 6; 7, 8; 9, 10]
```

```
D = 3×2
     5     6
     7     8
     9    10
```

```
[A, C]
```

```
ans = 2×5
     1     2     5     6     7
     3     4     8     9    10
```

```
[A; D]
```

```
ans = 5×2
     1     2
     3     4
     5     6
     7     8
     9    10
```

In fact, the definitions of the original matrices themselves use concatenation.

The rows or columns must be equal, respectively, otherwise it produces an error:

```
[A; C]
[A, D]
```

Character arrays can be concatenated, just like numeric arrays:

```
a = 'Hello, '
```

```
a =
'Hello, '
```

```
b = 'World!'
```

```
b =
'World!'
```

```
[a, b]
```

```
ans =
'Hello, World!'
```

Note that an ellipsis cannot be inserted between single quotes:

```
'Hello, ...
    World!'
```

Rows and columns can be deleted by setting them to the empty array:

```
A = magic( 4)
```

```
A = 4×4
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

```
A( 2, :) = []
```

```
A = 3×4
    16     2     3    13
     9     7     6    12
     4    14    15     1
```

```
A( :, 2) = []
```

```
A = 3×3
    16     3    13
     9     6    12
     4    15     1
```

Trying to delete a single entry in the matrix causes an error:

```
A( 2, 2) = []
```

However, you can use the linear index to remove the corresponding entry and output a row vector:

```
A( 4) = []
```

```
A = 1×8
    16     9     4     6    15    13    12     1
```

Note that concatention and removing rows and columns create new arrays, just as resizing does.

Sparse Matrices

Some matrices have a relatively small number of non-zero entries. Such matrices are called **sparse**, as opposed to **dense**. The `nnz` function returns the number of non-zero entries of its matrix input:

```
A = [1, 0; 0, 2; 3, 0]
```

```
A = 3×2
     1     0
     0     2
     3     0
```

```
nnz( A)
```

```
ans =
     3
```

The `nonzeros` function returns a column vector with the non-zero entries of its matrix input:

```
nonzeros( A)
```

```
ans = 3×1
     1
     3
```

```
     2
```

In most computational science problems, matrices are sparse. If matrices are dense, they are usually small, e.g. sub-blocks of a sparse matrix. It is not very common that you see large dense matrices. There are a lot of tricks that numerical linear algebra algorithms can use to increase efficiency for sparse matrices.

When creating a **data structure** for sparse matrices, most of the entries of the matrix do not need to be stored, since most of them are 0. You only need to store the locations and values of the non-zero entries. MATLAB has a special class of sparse matrices for this purpose. The `sparse` function takes as inputs vectors of the i-indices of the non-zero entries, the j-indices of the non-zero entries, and their values. This is called **A_ij** notation for sparse matrices:

```
a = [1; 3; 2]
```

```
a = 3×1
     1
     3
     2
```

```
b = [1; 1; 3]
```

```
b = 3×1
     1
     1
     3
```

```
c = [1; 3; 2]
```

```
c = 3×1
     1
     3
     2
```

```
B = sparse( a, b, c)
```

```
B =
   (1,1)        1
   (3,1)        3
   (2,3)        2
```

Note that the coordinates and values of the non-zero entries are displayed. Also note that the entries can be in any order.

(A_ij is actually not the most efficient way to store a sparse matrix. Better formats are **compressed row storage** (CRS) or compressed column storage (CCS), or their block versions. But since MATLAB does not take those as inputs, we will not discuss them here.)

Having duplicated indices in the vectors adds the values for that entry:

```
a = [1; 3; 2; 2]
```

```
a =
     1
     3
```

```
     2
     2
```

```
b = [1; 1; 3; 3]
```

```
b =
     1
     1
     3
     3
```

```
c = [1; 3; 2; 1]
```

```
c =
     1
     3
     2
     1
```

```
C = sparse( a, b, c)
```

```
C =
   (1,1)        1
   (3,1)        3
   (2,3)        3
```

The `sparse` function only allocates memory for the number of entries given by the vectors. Once again, you should try to determine the size of sparse matrices at the beginning to preallocate enough storage to avoid having to resize them.

The operations for dense matrices extend to sparse ones:

```
B * C
```

```
ans =
   (1,1)        1
   (2,1)        6
   (3,1)        3
```

To specify the size of the dimensions of the sparse matrix (e.g. to include trailing zero rows or columns), add them as inputs:

```
B = sparse( a, b, c, 4, 4)
```

```
B =
   (1,1)        1
   (3,1)        3
   (2,3)        3
```

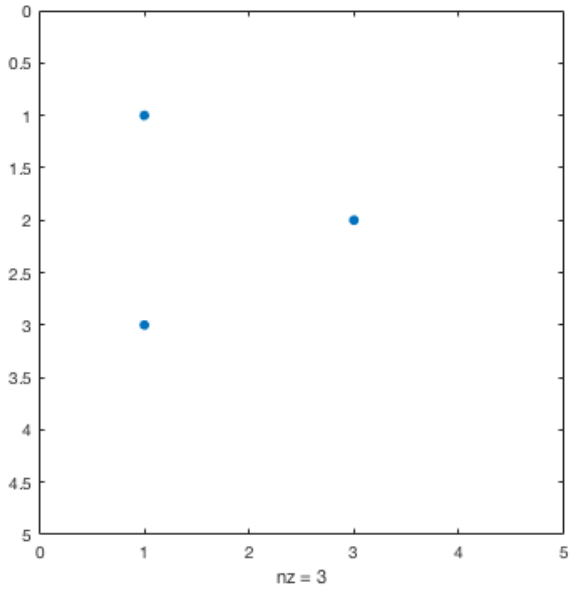
```
whos( 'B')
```

```
  Name      Size             Bytes  Class     Attributes

  B         4x4                104  double    sparse
```

The `spy` function plots the sparsity pattern of a sparse matrix:

```
spy( B)
```



This function is particularly useful for very large sparse matrices, since the sparsity pattern can help determine which linear solver will be optimal. For example, MATLAB has specialized solvers for tridiagonal, Hessenberg, banded, triangular, and symmetric matrices.

Sparse matrices are accessed the same way as dense matrices:

```
B( 3, 1)
```

```
ans =
   (1,1)        3
```

```
B( 3, 2)
```

```
ans =
   All zero sparse: 1×1
```

The `sparse` function with a dense matrix input returns its conversion to a sparse matrix:

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

```
A =
   (1,1)        1
   (3,1)        3
   (2,2)        2
```

The `full` function with a sparse matrix input returns its conversion to a dense matrix:

```
A = full( A)
```

```
A =
     1     0
     0     2
     3     0
```

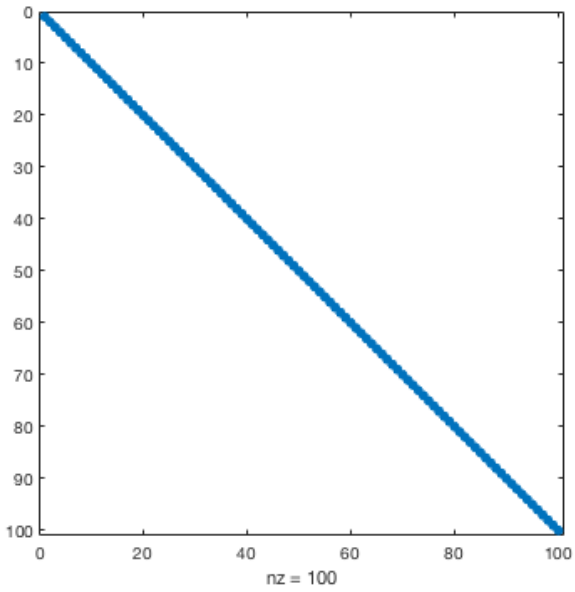
If there are only two scalar inputs, a zero square matrix is returned (i.e. there are no entries):

```
sparse( 2, 2)
```

```
ans =
   All zero sparse: 2×2
```

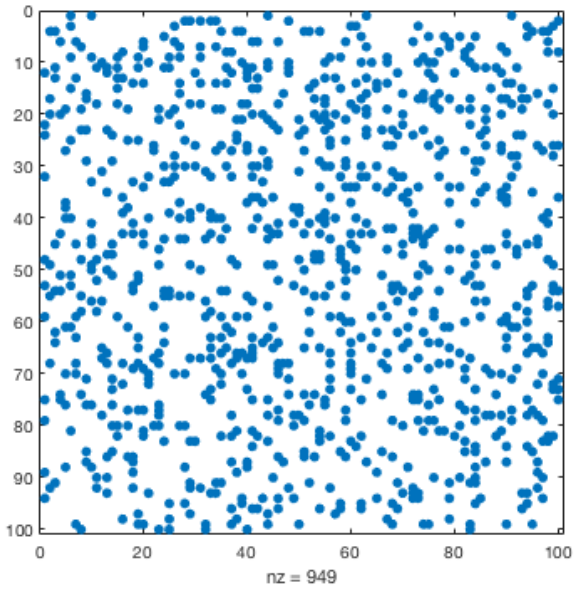
There is a special function `speye` that returns a sparse identity matrix:

```
spy( speye( 100))
```



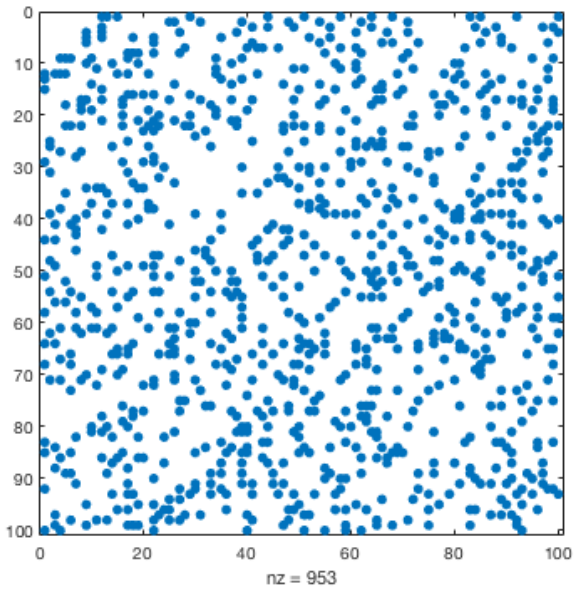
The `sprand` and `sprandn` functions return random sparse matrices with uniformly and normally-distributed entries with the first input as the number of rows, the second input as the number of columns, and the third input as the density of non-zero entries:

```
spy( sprand( 100, 100, 0.1))
```



The `sprandsym` function is the symmetric version of sprandn with the first input as the number of rows and column, and the second input as the density:

```
spy( sprandsym( 100, 0.1))
```



Note the symmetry about the diagonal.

The following function exists for testing types:

```
issparse( B)
```

```
ans = logical
```

```
    1
```

Operators on Arrays

Most arithmetic operators have been overloaded for arrays:

```
A = [1, 2; 3, 4]
```

```
A =
     1     2
     3     4
```

```
B = [5, 6; 7, 8]
```

```
B =
     5     6
     7     8
```

```
A + B
```

```
ans =
     6     8
    10    12
```

```
A - B
```

```
ans =
    -4    -4
    -4    -4
```

Note that the vectors must have compatible sizes, otherwise it gives an error:

```
C = [9, 10; 11, 12; 13, 14]
```

```
C =
     9    10
    11    12
    13    14
```

```
A + C
```

Multiplication uses the naive (grade-school) matrix multiplication algorithm for dense matrices:

```
A * B
```

```
ans =
    19    22
    43    50
```

For multiplication, the number of columns of the first matrix must equal the number of rows of the second matrix, otherwise it produces an error:

```
A * C
```

Of course, matrix multiplication is associative, $(A \times B) \times C = A \times (B \times C)$, but not commutative, $A \times B \neq B \times A$.

Right division gives a matrix `Y` such that $A = Y \times B$:

```
Y = A / B
```

```
Y =
     3.000000000000002e+00    -2.000000000000001e+00
     2.000000000000002e+00    -1.000000000000001e+00
```

```
Y * B
```

```
ans =
     1     2
     3     4
```

Left division gives a matrix `X` such that $A \times X = B$:

```
X = A \ B
```

```
X =
    -3    -4
     4     5
```

```
A * X
```

```
ans =
     5     6
     7     8
```

These operators solve linear systems. This is covered in AMS 526. The numerical solution of elliptic or parabolic PDEs (the subject of AMS 528) usually reduces to solving a large, sparse linear system. MATLAB does checking of the coefficient matrix to look for certain structures (e.g. triangular, symmetric), and then chooses the best linear solver. See the documentation for mldivide, also known as **backslash**, and search for "Systems of Linear Equations" in the documentation. Actually, my dissertation topic in the area of linear solvers.