

## 8.3: Using Sparse Matrices

Sparse matrix formats should be used, instead of conventional 2D arrays, when dealing sparse matrices of size  $10^3 \times 10^3$  or larger. (For matrices of size  $100 \times 100$  or less, the differences in performance and memory usage are usually negligible.)

Usually, it is good to choose either the CSR or CSC format, depending on what mathematical operations you intend to perform. You can construct the matrix either (i) directly, by means of a `(data, idx)` tuple as described above, or (ii) by creating an LIL matrix, filling in the desired elements, and then converting it to CSR or CSC format.

If you are dealing with a sparse matrix that is "strongly diagonal" (i.e., the non-zero elements occupy a very small number of diagonals, such as a tridiagonal matrix), then you can consider using the DIA format. The main advantage of the DIA format is that it is very easy to construct, by supplying a `(data, offsets)` input to the `dia_matrix` function, as described above. However, the format usually does not perform significantly better than CSR/CSC; and if the matrix is not strongly diagonal, its performance is much worse.

Another common way to construct a sparse matrix is to use the `scipy.sparse.diags` or `scipy.sparse.spdiags` functions. These functions let you specify the contents of the matrix in terms of its diagonals, as well as which sparse format to use. The two functions have slightly different calling conventions; see the documentation for details.

### 8.3.1 The `dot` method

Each sparse matrix has a `dot` method, which calculates the product of the matrix with an input (in the form of a 1D or 2D array, or sparse matrix), and returns the result. For sparse matrix products, this method should be used instead of the stand-alone function, similarly named `dot`, which is used for non-sparse matrix products. The sparse matrix method makes use of matrix sparsity to speed up the calculation of the product. Typically, the CSR format is faster at this than the other sparse formats.

For example, suppose we want to calculate the product  $\mathbf{A}\vec{x}$ , where

$$\mathbf{A} = \begin{bmatrix} 0 & 1.0 & 0 & 0 & 0 \\ 0 & 2.0 & -1.0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 6.6 & 0 & 0 & 0 & 1.4 \end{bmatrix}, \quad \vec{x} = \begin{bmatrix} 1 \\ 1 \\ 2 \\ 3 \\ 5 \end{bmatrix}. \quad (8.3.1)$$

This could be accomplished with the following program:

```
from scipy import *
import scipy.sparse as sp

data = [1.0, 2.0, -1.0, 6.6, 1.4]
rows = [0, 1, 1, 3, 3]
cols = [1, 1, 2, 0, 4]
A = sp.csr_matrix((data, [rows, cols]), shape=(4,5))
x = array([1.,1.,2.,3.,5.])

y = A.dot(x)

print(y)
```

Running this program gives the expected result:

```
[ 1.   0.   0.  13.6]
```

### 8.3.2 spsolve

The `spsolve` function, provided in the `scipy.sparse.linalg` module, is a solver for a sparse linear system of equations. It takes two inputs, **A** and **b**, where **A** should be a sparse matrix; **b** can be either sparse, or an ordinary 1D or 2D array. It returns the **x** which solves the linear equations **x**. The return value can be either a conventional array or a sparse matrix, depending on whether **b** is sparse.

For sparse problems, you should always use `spsolve` instead of `scipy.linalg.solve` (the usual solver for non-sparse problems). Here is an example program showing `spsolve` in action:

```
from scipy import *
import scipy.sparse as sp
import scipy.sparse.linalg as spl

## Make a sparse matrix A and a vector x
data = [1.0, 1.0, 1.0, 1.0, 1.0]
rows = [0, 1, 1, 3, 2]
cols = [1, 1, 2, 0, 3]
A = sp.csr_matrix((data, [rows, cols]), shape=(4,4))
b = array([1.0, 5.0, 3.0, 4.0])

## Solve Ax = b
x = spl.spsolve(A, b)
print(" x = ", x)

## Verify the solution:
print("Ax = ", A.dot(x))
print(" b = ", b)
```

Running the program gives:

```
x = [ 4.  1.  4.  3.]
Ax = [ 1.  5.  3.  4.]
b = [ 1.  5.  3.  4.]
```

### 8.3.3 eigs

For eigenvalue problems involving sparse matrices, one typically does not attempt to find *all* the eigenvalues (and eigenvectors). Sparse matrices are often so huge that solving the full eigenvalue problem would take an impractically long time, even if we receive a speedup from sparse matrix arithmetic. Luckily, in most situations we only need to find a subset of the eigenvalues (and eigenvectors). For example, after discretizing the 1D Schrödinger wave equation, we are normally only interested in the several lowest energy eigenvalues.

The `eigs` function, provided in the `scipy.sparse.linalg` module, is an eigensolver for sparse matrices. Unlike the eigensolvers we have previously discussed, such as `scipy.linalg.eig`, the `eigs` function only returns a specified subset of the eigenvalues and eigenvectors.

The `eigsh` function is similar to `eigs`, except that it is specialized for Hermitian matrices. Both functions make use of a low-level numerical eigensolver library named [ARPACK](#), which is also used by GNU Octave, Matlab, and many other numerical tools. We will not discuss how the algorithm works.

The first input to `eigs` or `eigsh` is the matrix for which we want to find the eigenvalues. Several other optional inputs are also accepted. Here are the most commonly-used ones:

- The optional parameter named `k` specifies the number of eigenvalues to find (the default is 6).
- The optional parameter named `M`, if supplied, specifies a right-hand matrix for a generalized eigenvalue problem.
- The optional parameter named `sigma`, if supplied, should be a number; it means to find the `k` eigenvalues which are closest to that number.
- The optional parameter named `which` specifies which eigenvalues to find, using a criteria different from `sigma`: `'LM'` means to find the eigenvalues with the largest magnitudes, `'SM'` means to find those with the smallest magnitudes, etc. You cannot simultaneously specify both `sigma` and `which`. When finding small eigenvalues, it is usually better to use `sigma` instead of `which` (see the discussion in the next section).
- The optional parameter named `return_eigenvectors`, if `True` (the default), means to return both eigenvalues and eigenvectors. If `False`, the function returns the eigenvalues only.

For the full list of inputs, see the full function documentation for `eigs` and `eigsh`.

By default, `eigs` and `eigsh` return two values: a 1D array (which is complex for `eigs` and real for `eigsh`) containing the found eigenvalues, and a 2D array where each column is one of the corresponding eigenvectors. If the optional parameter named `return_eigenvectors` is set to `False`, then only the 1D array of eigenvalues is returned.

In the next section, we will see an example of using `eigsh`.

---

This page titled [8.3: Using Sparse Matrices](#) is shared under a [CC BY-SA 4.0](#) license and was authored, remixed, and/or curated by [Y. D. Chong](#) via [source content](#) that was edited to the style and standards of the LibreTexts platform.