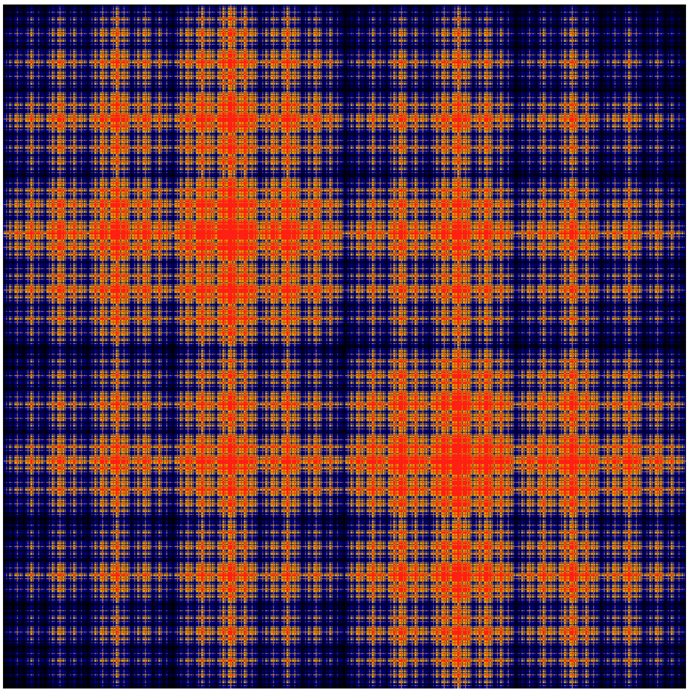
Applied Mathematics and Computing

Volume I



List of Contributors

J. Humpherys
Brigham Young University

J. Webb
Brigham Young University

R. Murray
Brigham Young University

J. West
University of Michigan

R. Grout
Brigham Young University

K. Finlinson
Brigham Young University

A. Zaitzeff
Brigham Young University

Preface

This lab manual is designed to accompany the textbook *Foundations of Applied Mathematics* by Dr. J. Humpherys.

Lab 5

Algorithms: Matrix Operations and Algorithmic Complexity

Lesson Objective: *This section explains how to create specific types of large matrices. It also introduces the concept of temporal complexity. Finally, it explores SciPy's special methods for working with sparse matrices.*

Temporal Complexity

One of the most important questions in scientific computing is: How long will this operation take? The concept of temporal complexity attempts to answer this question by determining how much time a function needs to operate on a given size of input. For example suppose calculating the inverse of a matrix of size n requires the following number of calculations.

$$f(n) = \frac{3n^3}{2} + 75n^2 + 250n + 30$$

What is the most important part of this expression? When our input gets very large the only relevant term in this equation is n^3. For this reason we say that $f(n) \in O(n^3)$, or more commonly that $f(n)$ is $O(n^3)$ (spoken "Big O of n cubed" or "Order of n cubed"). This notation is borrowed from analysis. This notation captures the salient behavior of our temporal complexity, or more precisely the growth rate we can expect of the execution time of our algorithm. We will discuss this concept later, but this is a simple introduction to the notion of complexity and Big O. Spatial complexity is the amount of memory an algorithm uses, and is defined similarly.

Advanced Matrix Tools

We now introduce a few different ways to build matrices. Two important methods available for building matrices are `zeros()` and `ones()`. These commands allow us to build matrices populated entirely with zeros or ones, respectively. For example, to build a 3-vector filled with zeros we enter the following command:

```
: import scipy as sp
: sp.zeros((3,1))
array([[ 0.],
       [ 0.],
       [ 0.]])
```

To find additional options for these methods, you can use the help system.

One important use of the `zeros()` method is to allow us to pre-allocate memory. Pre-allocation is simply the practice of reserving a chunk of memory for later use. We can always add more space to a matrix using the methods we learned in lab 1, but this requires many extra internal operations because of way arrays are stored in memory. Thus, it is generally faster to allocate a matrix with its final size and modify its values rather than building an array as you go.

Table 1.3 gives a few commands that allow us to build types of useful matrices.

Function	Description	Usage
`eye()`	Identity matrix	sp.eye(m, n)
`zeros()`	Zero matrix	sp.zeros((m, n))
`ones()`	One matrix	sp.ones((m, n))
`diag()`	Building (or retrieving) along a diagonal	
`linalg.toeplitz()`	Matrix with constant diagonals	la.toeplitz()
`linalg.triu()`	Upper triangular	
`linalg.tril()`	Lower triangular	
`rand`	Psuedo-random matrix, uniformily distributed	
`randn`	Psuedo-random matrix, normally distributed	
`random.randint()`	Psuedo-random matrix, uniformily distributed integers	sp.random.randint()
`tile()`	Copy across a given dimension	sp.tile(A, reps)

Table 5.1: Special matrix creation commands

For example, suppose that we want to create a matrix with -2 on the diagonal, and ones on the super and sub diagonal. We can do this by using the following command:

```
: from scipy import linalg as la
: la.toeplitz([-2,1,0])
array([[-2,  1,  0],
       [ 1, -2,  1],
       [ 0,  1, -2]])
```

This matrix is useful because it numerically approximates the second derivative of a function. We investigate some properties of this matrix in Problem 6 of this lab, and explain more about this matrix later.

Problem 1 Use the `diagflat()` method to create the following matrices. All of these matrices should be easily scaleable (ie only minor modification would be required to change the size).

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 0 & 1 & 2 & 3 & 4 \\ 0 & 0 & 1 & 2 & 3 \\ 0 & 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \qquad \begin{pmatrix} 1 & 1/2 & 1/3 & 1/4 & 1/5 \\ 1/2 & 1 & 1/2 & 1/3 & 1/4 \\ 1/3 & 1/2 & 1 & 1/2 & 1/3 \\ 1/4 & 1/3 & 1/2 & 1 & 1/2 \\ 1/5 & 1/4 & 1/3 & 1/2 & 1 \end{pmatrix}$$

Problem 2 Create the matrices from Problem 1 using the methods `linalg.toeplitz ()` or `linalg.triu()`. Which method is easier? Now use whichever command is easiest to create the matrix:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 & 5 \end{pmatrix}$$

Sparse Matrices

In this section we discuss how sparse matrices are used and constructed. A sparse matrix is a matrix that has few non-zero entries (where few is generally relative to the number of entries in the matrix). SciPy has several different ways of storing sparse matrices. Each way has it pros and cons (the reader is encouraged to read the help for way).

Type the following into IPython.

```
: from scipy import sparse as spar
: A = sp.diagflat([2,3,4])
: B = spar.csc_matrix(A)
: C = B.todense()
```

Notice that the matrix A has only three non-zero entries, and so we can consider it sparse. In memory, an array stores a bit of data (be it an integer, float,

Function	Description
`sparse.bsr()`	Compressed Block Sparse Row
`sparse.coo()`	Coordinate
`sparse.csc()`	Compressed Sparse Column
`sparse.csr()`	Compress Sparse Row
`sparse.dia()`	Sparse Diagonal
`sparse.dok()`	Dictionary of Keys
`sparse.lil()`	Linked List

Table 5.2: Sparse matrix representations in SciPy

or complex number) each entry, meaning that a 3×3 matrix requires a total 9 blocks of memory. However, if we leverage the sparsity of A we realize that we only need to store 3 numbers. The `sparse` methods do exactly this: they store only the non-zero entries and their locations in the matrix. No longer are we working with array. SciPy has many methods for performing operations on sparse arrays. To convert back to a dense matrix, we use the `.todense()` property of the sparse matrix. We can also convert between the different types sparse arrays.

We remark that if you want to make a sparse diagonal matrix, the best way to do it isn't to use `diagflat()` followed by `sparse`, it's actually better to use the `sparse.spdiags()` method:

```
: spar.spdiags([2,3,4],0,3,3)
```

This is because oftentimes when we are using sparse matrices we are dealing with matrices that are too large to be handled efficiently by python when represented in full form.

Banded Matrices

A banded matrix is one whose only non-zero entries are diagonal strips. For example, the matrix

$$A = \begin{pmatrix} 1 & 2 & 0 & 0 \\ 3 & 4 & 5 & 0 \\ 0 & 6 & 7 & 8 \\ 0 & 0 & 9 & 10 \end{pmatrix}$$

is banded because there are three nonzero diagonals. This particular type of banded matrix is called a tri-diagonal matrix.

You can easily create banded matrices using the `diagflat()` method. For example, the matrix A above can be created by entering

```
: sp.diagflat([3,6,9],-1) + sp.diagflat([1,4,7,10],0) + sp.diagflat
    ([2,5,8],1)
```

Often a better way to create a tri-diagonal is it use the `spar.spdiags()` method. This is because many diagonal matrices are sparse. For example, we create the same matrix in Python (while designating that it is sparse) using the command:

```
: Z = sp.array([[3, 1, 0],[6, 4, 2],[9, 7, 5],[0,10,8]]).T
: spar.spdiags(Z,[-1,0,1],4,4)
```

For more information, check the documentation by typing `spar.spdiags?`. For example we create a tri-diagonal array with uniformily distributed random entries. This example also demonstartes the efficiency of using sparse arrays.

```
: B = sp.rand(3,10000)
: A = spar.spdiags(B,range(-1,2),10000,10000)
: denseA = A.todense()  #only do this step if you have _lots_ of
    memory!
: A.data.nbytes
240000          #about 0.24MB of memory
: denseA.nbytes
800000000       #about 762.9MB of memory!
```

We can't use the `full` command in this case because the computer will almost certainly run out of memory (the matrix is 10,000 × 10,000). However, we can still visualize this matrix using the `plt.spy()` command from matplotlib, which essentially shows the location of non-zero entries in a matrix. The output of `plt.spy(A)` in this case is shown in Figure 1.2:

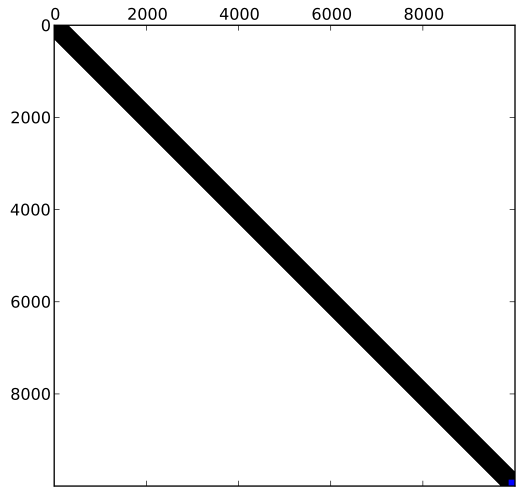


Figure 5.1: The output of the `spy` command.

Using Sparse Matrices

Consider the linear system $Ax = b$, where A is a 100,000 $\times$ 100,000 tri-diagonal matrix. To store a full matrix of that size in your computer, it would normally require 10 billion double-precision floating-point numbers. Since it takes 8 bytes to store a double, it would take roughly 80GB to store the full matrix. For most desktop computers, that fact alone makes the system numerically prohibitive to solve. The temporal complexity of the problem is even more problematic. Methods for directly solving an arbitrary linear system are usually $O(n^3)$. As a result, even if the computer could store an 80GB matrix in RAM, it would still take several weeks to solve the system. However, since we don't have computers with that much available RAM, most of the matrix would have to be stored on the hard drive, so the computation would probably take between 6 months to a year.

The point is that even the next generation of computers will struggle with solving arbitrary linear systems of this size in a reasonable period of time. However, if we take advantage of the sparse structure of the tri-diagonal matrix, we can solve the linear system, even with a modest modern computer. This is because all of those zeros don't need to be stored and we don't need to do as many operations to row reduce the tri-diagonal system.

Let's first compute the spatial complexity of the above system when considered as a sparse matrix. There are three diagonals that have roughly 100,000 non-zero entries. That's 300,000 double-precision floating point numbers, which is about 2.4 MB (Less storage than your favorite song). As a result, it will easily fit into the computer's RAM. Furthermore, the temporal complexity for solving a tri-diagonal matrix is $O(n)$. Let's see how long it takes to solve the system for random data:

```
: from scipy.sparse import linalg as sparla
: from timer import timer
: D = sp.rand(3, 100000)
: b = sp.rand(1, 100000)
: A = spar.spdiags(D,[-1,0,1],100000,100000)
: def solSys():
....: return sparla.spsolve(A,b)
: with timer() as t:
....: t.time(solSys)
....: t.printTimes()
```

Problem 3 Write a function that returns a full $n \times n$ tri-diagonal array with 2's along the diagonal and -1's along the two sub-diagonals above and below the diagonal. Hint: Use the `la.toeplitz()` method. Note that this is the second derivative matrix that we discussed at the beginning of this lab.

Problem 4 Write another function that builds the same array as above, but as a sparse array. You must build this as a sparse matrix from the beginning. Hint: Use the `spar.spdiags()` method.

Problem 5 Solve the linear system $Ax = b$ where A is the $n \times n$ tri-diagonal array from the above two problems and b is randomly generated. How high can you go for each method? Make a table for several different values of n and the time it took to solve for each run. What conclusions can you draw?

Problem 6 Using the sparse array above and the method `la.eigs()`, calculate the smallest eigenvalue λ of the array as the array's size goes to infinity. What value does λn^2 approach? Hint: It's the square of an important number. This is related to operator theory: the second derivative operator has this eigenvalue in certain cases.

Other Sparse Commands

One important method of sparse array objects is the `nonzero()` method, which is related to the number of nonzero entries in an array. This number is important because it is an indicator of the amount of time and space that is required to operate on the sparse array. You should be aware that there is some overhead to using and storing the sparse array data structure. Sparsely represented arrays are very beneficial when the number of nonzero entries is relatively small compared to the total number of entries. When the array has many nonzero entries, a sparse representation becomes disadvantageous. To see this, create and execute a script with the following code:

```
: A = sp.rand(600,600); B = spar.csc_matrix(A)
: def square(A): return sp.power(A, 2)
: with timer() as t:
....: t.time(square, A)
....: t.time(square, B)
```

Run the script and note the two different runtimes. Notice that it takes much longer to square the sparse array. This is because the sparse array data structure is optimized for arrays that are actually sparse. The array A is entirely nonzero. Thus, you incur the overhead of the sparse array representation without any benefits since there are no entries you are not required to store or compute. To summarize, only use a sparse array when your array is in fact sparse. Using sparse arrays for mostly nonzero arrays will negatively impact performance and memory requirements.

Just as with dense arrays, we can pre-allocate sparse arrays. Sometimes it is necessary to create sparse matrices that do not have a nice banded pattern. We initialize a sparse array just like any other array. The most efficient sparse array for pre-allocation is LIL. Once you are done constructing you sparse array and wish to perform calculations, you should convert to a more efficient sparse array (CSR or CSC).

```
: Z = spar.lil_matrix((400,300))
<400x300 sparse matrix of type '<type 'numpy.float64'>'
        with 0 stored elements in LInked List format>
: Z[1,34] = 23
: Z[23,32] = 56
: Z[2,:] = 13.2
```

This code snippet creates a 400×300 LIL sparse array. We can then work with the sparse array as though it were a dense array. When the array is initialized all of the entries are assumed to be zero.