MODELS USING MATRICES WITH PYTHON

José M. Garrido
Department of Computer Science

May 2016

College of Computing and Software Engineering
Kennesaw State University

1 Matrices

In general, a matrix is a two-dimensional array of data values and is organized in rows and columns. The following array, Y, is a *two-dimensional array* organized as an $m \times n$ matrix; its elements are arranged in m rows and n columns.

$$Y = \begin{bmatrix} y_{11} & y_{12} & \cdots & y_{1n} \\ y_{21} & y_{22} & \cdots & y_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ y_{m1} & y_{m2} & \cdots & y_{mn} \end{bmatrix}$$

The first row of Y consists of elements: $y_{11}, y_{12}, \ldots, y_{1n}$. The second row consists of elements: $y_{21}, y_{22}, \ldots, y_{2n}$. The last row of Y consists of elements: $y_{m1}, y_{m2}, \ldots, y_{mn}$. In a similar manner, the elements of each column can be identified.

1.1 Basic Concepts

A matrix is defined by specifying the rows and columns of the array. An m by n matrix has m rows and n columns. A *square* matrix has the same number of rows and columns, n rows and n columns, which is denoted as $n \times n$. The following example shows a 2×3 matrix, which has two rows and three columns:

$$\begin{bmatrix} 0.5000 & 2.3500 & 8.2500 \\ 1.8000 & 7.2300 & 4.4000 \end{bmatrix}$$

A matrix of dimension $m \times 1$ is known as a *column vector* and a matrix of dimension $1 \times n$ is known as a *row vector*. A vector is considered a special case of a matrix with one row or one column. A row vector of size n is typically a matrix with one row and n columns. A column vector of size m is a matrix with m rows and one column.

The elements of a matrix are denoted with the matrix name in lower-case and two indices, one corresponding to the row of the element and the other index corresponding to the column of the element. For matrix Y, the element at row i and column j is denoted by y_{ij} or by $y_{i,j}$.

The *main diagonal* of a matrix consists of those elements on the diagonal line from the top left and down to the bottom right of the matrix. These elements have the same value of the two indices. The diagonal elements of matrix Y are denoted

by y_{ii} or by y_{jj}. For a square matrix, this applies for all values of i or all values of j. For a square matrix X (an $n \times n$ matrix), the elements of the main diagonal are:

$$x_{1,1},\ x_{2,2},\ x_{3,3},\ x_{4,4},\ \ldots,\ x_{n,n}$$

An *identity matrix* of size n, denoted by I_n is a square matrix that has all the diagonal elements with value 1, and all other elements (off-diagonal) with value 0. The following is an identity matrix of size 3 (of order n):

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

1.2 Arithmetic Operations

The arithmetic matrix operations are similar to the ones discussed previously for vectors. The multiplication of a matrix Y by a *scalar* λ calculates the multiplication of every element of matrix Y by the scalar λ. The result defines a new matrix.

$$Y = \begin{bmatrix} y_{11} & y_{12} & \cdots & y_{1n} \\ y_{21} & y_{22} & \cdots & y_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ y_{m1} & y_{m2} & \cdots & y_{mn} \end{bmatrix} \qquad \lambda Y = \begin{bmatrix} \lambda y_{11} & \lambda y_{12} & \cdots & \lambda y_{1n} \\ \lambda y_{21} & \lambda y_{22} & \cdots & \lambda y_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \lambda y_{m1} & \lambda y_{m2} & \cdots & \lambda y_{mn} \end{bmatrix}$$

In a similar manner, the addition of a scalar λ to matrix Y, calculates the addition of every element of the matrix with the scalar λ. The subtraction of a scalar λ from a matrix Y, denoted by $Y - \lambda$, computes the subtraction of the scalar λ from every element of matrix Y

The *matrix addition* is denoted by $Y + Z$ of two $m \times n$ matrices Y and Z, calculates the addition of every element of matrix Y with the corresponding element of matrix Z. The result defines a new matrix. This operation requires that the two matrices have the same number of rows and columns.

$$Y + Z = \begin{bmatrix} y_{11} + z_{11} & y_{12} + z_{12} & \cdots & y_{1n} + z_{1n} \\ y_{21} + z_{21} & y_{22} + z_{22} & \cdots & y_{2n} + z_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ y_{m1} + z_{m1} & y_{m2} + z_{m2} & \cdots & y_{mn} + z_{mn} \end{bmatrix}$$

Similarly, the subtraction of two matrices Y and Z, denoted by $Y - Z$, subtracts every element of matrix Z from the corresponding element of matrix Y. The result defines a new matrix.

The *element-wise multiplication* (also known as the Hadamard product or Schur product) of two matrices, multiplies every element of a matrix X by the corresponding element of the second matrix Y and is denoted by $X \circ Y$. The result defines a new matrix, Z. This operation requires that the two matrices have the same number of rows and columns. The following is the general form of the element-wise multiplication of matrix X multiplied by matrix Y.

$$Z = X \circ Y = \begin{bmatrix} x_{11} \times y_{11} & x_{12} \times y_{12} & \cdots & x_{1n} \times y_{1n} \\ x_{21} \times y_{21} & x_{22} \times y_{22} & \cdots & x_{2n} \times y_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} \times y_{m1} & x_{m2} \times y_{m2} & \cdots & x_{mn} \times y_{mn} \end{bmatrix}$$

The *determinant* of a matrix A, denoted by $\det A$ or by $\mid A \mid$, is a special number that can be computed from the matrix. The determinant is useful to describe various properties of the matrix that are applied in systems of linear equations and in calculus.

The determinant provides important information when the matrix consists of the coefficients of a system of linear equations. If the determinant is nonzero the system of linear equations has a unique solution. When the determinant is zero, there are either no solutions or many solutions.

One of several techniques to compute the determinant of a matrix is applying Laplace's formula, which expresses the determinant of a matrix in terms of its minors. The minor $M_{i,j}$ is defined to be the determinant of the submatrix that results from matrix A by removing the ith row and the jth column. Note that this technique is not very efficient. The expression $C_{i,j} = (1)^{i+j} M_{i,j}$ is known as a *cofactor*. The determinant of A is computed by

$$\det A = \sum_{j=1}^{n} C_{i,j} \times a_{i,j} \times M_{i,j}$$

The *matrix multiplication* of an $m \times n$ matrix X and an $n \times p$ matrix Y, denoted by XY, defines another matrix Z of dimension m by p and the operation is denoted by $Z = XY$. In matrix multiplication, the number of rows in the first matrix has to equal the number of columns in the second matrix. An element of the new matrix Z is determined by:

$$z_{ij} = \sum_{k=1}^{n} x_{ik}\ y_{kj}$$

that is, element z_{ij} of matrix Z is computed as follows:

$$z_{ij} = x_{i1}\ y_{1j} + x_{i2}\ y_{2j} + \ldots + x_{in}\ y_{nj}$$

The matrix multiplication is not normally commutative, that is, $XY \neq YX$. The following example defines a 2 by 3 matrix, X, and a 3 by 3 matrix, Y. The matrix multiplication of matrix X and Y defines a new matrix Z.

$$X = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 2 & 1 \end{bmatrix} \quad Y = \begin{bmatrix} 1 & 2 & 0 \\ 0 & 3 & 1 \\ -2 & 1 & 1 \end{bmatrix}$$

$$Z = XY = \begin{bmatrix} -1 & 9 & 3 \\ -2 & 7 & 3 \end{bmatrix}$$

The *transpose* of an $m \times n$ matrix X is an $n \times m$ matrix X^T formed by interchanging the rows and columns of matrix X. For example, for the given matrix X, the transpose (X^T) is:

$$X = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 5 & 3 \end{bmatrix} \qquad X^T = \begin{bmatrix} 1 & 0 \\ 2 & 5 \\ 1 & 3 \end{bmatrix}$$

The *conjugate* of an $m \times n$ complex matrix Z is a matrix $\overline{Z}$ with all its elements conjugate of the corresponding elements of matrix Z. The *conjugate transpose* of an $m \times n$ matrix Z, is an $n \times m$ matrix that results by taking the transpose of matrix Z and then the complex conjugate. The resulting matrix is denoted by Z^H or by Z^* and is also known as the *Hermitian transpose*, or the *adjoint matrix* of matrix Z. For example:

$$Z = \begin{bmatrix} 1.5 + 2.3i & 2.1 - 1.4i & 1 + 0.7i \\ 0 + 3.2i & 5.2 - 1.5i & 3.7 + 3.5i \end{bmatrix} \quad Z^H = \begin{bmatrix} 1.5 - 2.3i & 0 - 3.2i \\ 2.1 + 1.4i & 5.2 + 1.5i \\ 1 - 0.7i & 3.7 - 3.5i \end{bmatrix}$$

A square matrix Y is *symmetric* if $Y = Y^T$. The following example shows a 3×3 symmetric matrix Y.

$$Y = \begin{bmatrix} 1 & 2 & 7 \\ 2 & 3 & 4 \\ 7 & 4 & 1 \end{bmatrix}$$

A square matrix X is *triangular* if all the elements above or below its diagonal have value zero. A square matrix Y is *upper triangular* if all the elements below its diagonal have value zero. A square matrix Y is *lower triangular* if all the elements

above its diagonal have value zero. The following examples shows a matrix Y that is upper triangular.

$$Y = \begin{bmatrix} 1 & 2 & 7 \\ 0 & 3 & 4 \\ 0 & 0 & 1 \end{bmatrix}$$

The *rank* of a matrix Y is the number of independent columns in matrix Y, or the number of linearly independent rows of matrix Y. The *inverse* of an $n \times n$ matrix X is another matrix X^{-1} (if it exists) of dimension $n \times n$ such that their matrix multiplication results in an identity matrix or order n. This relation is expressed as:

$$X^{-1}X = XX^{-1} = I_n$$

A square matrix X is *orthogonal* if for each column x_i of X, $x_i^T x_j = 0$ for any other column x_j of matrix X. If the rows and columns are orthogonal unit vectors (the norm of each column x_i of X has value 1), then X is *orthonormal*. A matrix X is orthogonal if its transpose X^T is equal to its inverse X^{-1}. This implies that for orthogonal matrix X,

$$X^T X = I$$

Given matrices A, X, and B, a general matrix equation is expressed by $AX = B$. This equation can be solved for X by pre-multiplying both sides of the matrix equation by the inverse of matrix A. The following the expression shows this:

$$A^{-1}AX = A^{-1}B$$

This results in

$$X = A^{-1}B$$

2 Matrix Manipulation with Numpy

Matrices are created and manipulated in Python by calling the various library function in the Numpy and Scipy packages. Before using a matrix, it needs to be created. Matrices are created in a similar manner than the one used to create vectors.

2.1 Creating, Initializing, and Indexing Matrices

The most straightforward way to create a matrix with Numpy is to call function *array* and specify the values in the matrix arranged in rows and columns. Numpy function *ones* creates a matrix of the specified number of rows and columns, and returns a new matrix. Numpy function *zeros* creates a matrix of the specified number of rows and columns, and returns a new matrix. Matrices are stored in row-major order, the elements of each row form a contiguous block in memory. When a matrix is no longer needed in the program, it can be destroyed by calling the *del* command.

The following program shows how to create and manipulate matrices in Python and is stored in file `tmat1.py`. In line 3, matrix *amat* is created given the values of the elements arranged into three rows and two columns. In line 6, matrix *bmat* is created by calling Numpy function *ones* and specifying three rows and two columns with all elements initialized to 1. In line 9, calling Numpy function *ones* creates matrix *cmat* with two rows and three columns and all elements initialized to 0. An element of matrix *amat* is accessed by specifying index 2 and index 1, which indicate the element in row 2 and column 1. Finally, in line 14 the assignment gets the element in row 0 and column 1 and assigns the value to *q*.

```
 1 import numpy as np
 2 print "Create and manipulate a matrix"
 3 amat = np.array([[1.0, 3.0], [2.0, 5.0], [4.0, 7.0]])
 4 print "Matrix amat: "
 5 print amat
 6 bmat = np.ones([3,2])
 7 print "Matrix bmat: "
 8 print bmat
 9 cmat = np.zeros([2,3])
10 print "Matrix cmat:"
11 print cmat
12 p = amat[2,1]
13 print "Value of p: ", p
14 q = amat[0,1]
15 print "Value of q: ", q
```

Executing the Python interpreter and running the program in file `tmat1.py`, yields the following output. Note that matrices *amat* and *bmat* are two-dimensional array with three rows and two columns, matrix *cmat* is a matrix of two rows and three columns.

```
$ python tmat1.py
```

```
Create and manipulate a matrix
Matrix amat:
[[ 1.  3.]
 [ 2.  5.]
 [ 4.  7.]]

Matrix bmat:
[[ 1.  1.]
 [ 1.  1.]
 [ 1.  1.]]

Matrix cmat:
[[ 0.  0.  0.]
 [ 0.  0.  0.]]

Value of p:  7.0
Value of q:  3.0
```

Numpy function *identity* creates an identity matrix and sets it to the specified matrix and number of rows. Therefore, the elements of the main diagonal will have a value of 1, and all other elements will have a value of zero. The following segment of Python code creates identity matrix *imat* of four rows.

```
import numpy as np
print "Create an identity matrix of four rows"
imat = np.identity(4)
print imat
```

Running the code with the Python interpreter produces the following output.

```
Create an identity matrix of four rows
[[ 1.  0.  0.  0.]
 [ 0.  1.  0.  0.]
 [ 0.  0.  1.  0.]
 [ 0.  0.  0.  1.]]
```

2.2 Element Addition and Subtraction Operations

The basic operations on matrices are performed with a matrix and a scalar or with two matrices. These are carried out by calling several GSL functions in C.

In a similar as with vectors, adding a scalar to a matrix, involves adding the scalar value to every element of the matrix. With Python and Numpy, the addition operator (+) is applied directly to add a value to a matrix.

In the following portion of Python code, matrix *amat* the constant value 5.5 is added to the matrix and the result is used to create matrix *cmat*. In a similar manner, subtracting a scalar from a matrix is specified.

```
import numpy as np
print "Simple matrix operations"
amat = np.array([[1.0, 3.0], [2.0, 5.0], [4.0, 7.0]])
print "Matrix amat: "
print amat
print "Scalar addition and subtration: "
cmat = amat + 5.5
dmat = amat - 3.2
print "Matrix cmat:"
print cmat
print "Matrix dmat:"
print dmat
```

Running the code with the Python interpreter produces the following output.

```
Simple matrix operations
Matrix amat:
[[ 1.  3.]
 [ 2.  5.]
 [ 4.  7.]]
Scalar addition and subtration:
Matrix cmat:
[[  6.5   8.5]
 [  7.5  10.5]
 [  9.5  12.5]]
Matrix dmat:
[[-2.2 -0.2]
 [-1.2  1.8]
 [ 0.8  3.8]]
```

Matrix addition adds two matrices and involves adding the corresponding elements of each matrix. This addition operation on matrices is only possible if the two matrices are of the same size. *Matrix subtraction* is carried out by subtracting the corresponding elements of each of two matrices. The matrices must be of the same size.

The following program is stored in file `tmat4.py` and creates matrix *amat* and matrix *bmat*, which are both matrices of size 3×2. The statement in line 9 adds the elements of matrix *amat* with the elements of matrix *bmat* and then creates the

resulting matrix *cmat*. In line 12, the elements of matrix *bmat* are subtracted from the elements of matrix *amat* and creates the resulting matrix *dmat*.

```
 1 import numpy as np
 2 print "Matrix addition and subtraction"
 3 amat = np.array([[1.0, 3.0], [2.0, 5.0], [4.0, 7.0]])
 4 print "Matrix amat: "
 5 print amat
 6 bmat = np.ones([3,2])
 7 print "\nMatrix bmat: "
 8 print bmat
 9 cmat = amat + bmat # matrix addition
10 print "\nMatrix cmat:"
11 print cmat
12 dmat = amat - bmat
13 print "\nMatrix dmat: "
14 print dmat
```

Executing the Python interpreter and running the program in file `tmat4.py`, yields the following output. Note that matrices *amat* and *bmat* are two-dimensional array with three rows and two columns.

```
$ python tmat4.py
Matrix addition and subtraction
Matrix amat:
[[ 1.  3.]
 [ 2.  5.]
 [ 4.  7.]]

Matrix bmat:
[[ 1.  1.]
 [ 1.  1.]
 [ 1.  1.]]

Matrix cmat:
[[ 2.  4.]
 [ 3.  6.]
 [ 5.  8.]]

Matrix dmat:
[[ 0.  2.]
 [ 1.  4.]
 [ 3.  6.]]
```

2.3 Element Multiplication and Division

Matrix scalar multiplication involves multiplying each element of the matrix by the value of the scalar. With Numpy, the multiplication operator ($*$) is directly applied to multiply the specified matrix by the constant factor.

Element by element matrix multiplication involves multiplying the corresponding elements of two matrices. With Numpy, the multiplication operator ($*$) is directly applied to multiply two matrices of equal size ($m \times n$). The operation multiplies the elements of the first matrix specified by the elements of the second specified matrix.

The following program is stored in file `tmat5.py` and creates matrix *amat* and matrix *bmat*, which are both matrices of size 3×2. The statement in line 9 multiplies the elements of matrix *amat* by the constant 2 and the resulting matrix is *cmat*. In line 12, the elements of matrix *amat* are multiplied by the elements of matrix *bmat* and creates the resulting matrix *dmat*.

```
 1 import numpy as np
 2 print "Matrix scalar and elementwise multiplication"
 3 amat = np.array([[1.0, 3.0], [2.0, 5.0], [4.0, 7.0]])
 4 print "Matrix amat: "
 5 print amat
 6 bmat = np.array([[2.5, 1.5], [3.0, 1.5], [2.0, 4.25]])
 7 print "\nMatrix bmat: "
 8 print bmat
 9 cmat = amat * 2.0 # matrix scalar multiplication
10 print "\nMatrix cmat:"
11 print cmat
12 dmat = amat * bmat
13 print "\nMatrix dmat: "
14 print dmat
```

Executing the Python interpreter and running the program in file `tmat5.py`, yields the following output. Note that matrices *amat* and *bmat* are two-dimensional array with three rows and two columns.

```
$ python tmat5.py
Matrix scalar and elementwise multiplication
Matrix amat:
[[ 1.  3.]
 [ 2.  5.]
 [ 4.  7.]]

Matrix bmat:
```

```
[[ 2.5   1.5 ]
 [ 3.    1.5 ]
 [ 2.    4.25]]

Matrix cmat:
[[  2.   6.]
 [  4.  10.]
 [  8.  14.]]

Matrix dmat:
[[  2.5    4.5 ]
 [  6.     7.5 ]
 [  8.    29.75]]
```

Matrix scalar division involves dividing each element of the matrix by the value of the scalar. With Numpy, the division operator (/) is applied directly for dividing the specified matrix by the constant factor.

Element by element matrix division involves dividing the corresponding elements of two matrices. With Numpy, the division operator (/) is applied directly to two matrices of equal size ($m \times n$) and it divides one matrix by the second matrix. This operation divides the elements of the first specified matrix by the elements of the second specified matrix.

The following program is stored in file `tmat6.py` and creates matrix *amat* and matrix *bmat*, which are both matrices of size 3×2. The statement in line 9 divides the elements of matrix *amat* by the constant 2 and the resulting matrix is *cmat*. In line 12, the elements of matrix *amat* are divided by the elements of matrix *bmat* and creates the resulting matrix *dmat*.

```
 1 import numpy as np
 2 print "Matrix scalar and elementwise division"
 3 amat = np.array([[1.0, 3.0], [2.0, 5.0], [4.0, 7.0]])
 4 print "Matrix amat: "
 5 print amat
 6 bmat = np.array([[2.5, 1.5], [3.0, 1.5], [2.0, 4.25]])
 7 print "\nMatrix bmat: "
 8 print bmat
 9 cmat = amat / 2.0 # matrix scalar multiplication
10 print "\nMatrix cmat:"
11 print cmat
12 dmat = amat / bmat
13 print "\nMatrix dmat: "
14 print dmat
```

Executing the Python interpreter and running the program in file `tmat6.py`, yields the following output. Note that matrices *amat* and *bmat* are two-dimensional array with three rows and two columns.

```
$ python tmat6.py
Matrix scalar and elementwise division
Matrix amat:
[[ 1.  3.]
 [ 2.  5.]
 [ 4.  7.]]

Matrix bmat:
[[ 2.5   1.5 ]
 [ 3.    1.5 ]
 [ 2.    4.25]]

Matrix cmat:
[[ 0.5  1.5]
 [ 1.   2.5]
 [ 2.   3.5]]

Matrix dmat:
[[ 0.4         2.        ]
 [ 0.66666667  3.33333333]
 [ 2.          1.64705882]]
```

2.4 Additional Matrix Functions

Various additional operations can be applied to matrices. Recall that simple assignment of matrices results in two references to the same matrix and is considered another *view* of the matrix. Copying one matrix to another matrix is performed by calling the Numpy function *copy*. This function copies the elements of the specified source matrix into the specified destination matrix. The two matrices must have the same length $m \times n$. In the following example, matrix *amat* is copied to matrix *bmat*.

```
bmat = np.copy (amat)
```

The slicing operator (:) allows referencing of a subset of a matrix. The following program is stored in file `tmat7a.py` and creates matrix *amat*, which is a matrix of size 3×2. The statement in line 6 applies slicing to reference column 0 of matrix

amat and assigned to vector *bmat*. In line 9, rows 0 and 1 of matrix *amat* are referenced and assigned to *cmat*. In line 11, row 1 of matrix *amat* is referenced and assigned to *dmat*. In line 14, row 2 of matrix *amat* is assigned new values from a specified vector.

```
 1 import numpy as np
 2 print "Applying slicing on a matrix"
 3 amat = np.array([[1.0, 3.0], [2.0, 5.0], [4.0, 7.0]])
 4 print "Matrix amat: ", amat.shape
 5 print amat
 6 bmat = amat[:, 0]       # column 0
 7 print "\nColumn 0 of matrix amat: ", bmat.shape
 8 print bmat
 9 cmat = amat[0:2,1] # rows 0 and 1 of col 1
10 print "\nRows 0 and 1 of col 1 Matrix amat: ", cmat
11 dmat = amat[1,:]   # row 1 all cols
12 print "\nRow 1 of matrix amat: ", dmat
13 # assign new values for row 2
14 amat[2, :] = [21.35, 8.55]
15 print "Updated matrix amat: "
16 print amat
```

Executing the Python interpreter and running the program in file `tmat7a.py`, yields the following output. Note that matrix *amat* is two-dimensional array with three rows and two columns.

```
$ python tmat7a.py
Applying slicing on a matrix
Matrix amat:  (3, 2)
[[ 1.  3.]
 [ 2.  5.]
 [ 4.  7.]]

Column 0 of matrix amat:  (3,)
[ 1.  2.  4.]

Rows 0 and 1 of col 1 Matrix amat:  [ 3.  5.]

Row 1 of matrix amat:  [ 2.  5.]
Updated matrix amat:
[[  1.     3.  ]
 [  2.     5.  ]
 [ 21.35   8.55]]
```

Copying rows and columns of a matrix to a vector or another matrix is performed by slicing with the : operator and using function *copy*. The following program is stored in file `tmat7.py` and creates matrix *amat*, which is a matrix of size 3×2. In line 6, matrix *amat* is copied to matrix *bmat*. In line 9, rows 0 and 1 in column 0 of matrix *amat* are copied to to matrix *cmat*. In line 11, row 1 of matrix *amat* is copied to to *dmat*.

```
 1 import numpy as np
 2 print "Copy a matrix, row, or column"
 3 amat = np.array([[1.0, 3.0], [2.0, 5.0], [4.0, 7.0]])
 4 print "Matrix amat: "
 5 print amat
 6 bmat = np.copy(amat)       # copy entire matrix
 7 print "\nMatrix bmat: "
 8 print bmat
 9 cmat = np.copy(amat[0:2,0]) # rows 0 to 1 of col 0
10 print "\nMatrix cmat: ", cmat
11 dmat = np.copy(amat[1,:])   # row 1 all cols
12 print "\nMatrix dmat: ", dmat
```

Executing the Python interpreter and running the program in file `tmat7.py`, yields the following output. Note that matrix *amat* is two-dimensional array with three rows and two columns.

```
$ python tmat7.py
Copy a matrix, row, or column
Matrix amat:
[[ 1.  3.]
 [ 2.  5.]
 [ 4.  7.]]

Matrix bmat:
[[ 1.  3.]
 [ 2.  5.]
 [ 4.  7.]]

Matrix cmat:  [ 1.  2.]

Matrix dmat:  [ 2.  5.]
```

To exchange rows and columns of a matrix, a copy of the specified row or column is first defined then an in-place copied is performed to the specified. The following

program is stored in file `tmat8c.py` and exchanges rows 0 and 1 of matrix *amat*. Function *exchange_col* is defined in lines 4–7 and is called in line 13 to exchange columns 0 and 1 of matrix *amat*.

```
 1 import numpy as np
 2 def exchange_col (x, ncol1, ncol2):
 3    tmat = np.copy(x[:,ncol1])
 4    x[:,ncol1] = x[:,ncol2]
 5    x[:,ncol2] = tmat
 6
 7 print "Exchange columns of a matrix"
 8 amat = np.array([[1.0, 3.0], [2.0, 5.0], [4.0, 7.0]])
 9 print "Matrix amat: ", amat.shape
10 print amat
11 exchange_col(amat, 0, 1)
12 print "Updated matrix amat: "
13 print amat
```

Executing the Python interpreter and running the program in file `tmat8c.py`, yields the following output. Note that matrix *amat* is two-dimensional array with three rows and two columns.

```
$ python tmat8c.py
Exchange columns of a matrix
Matrix amat:  (3, 2)
[[ 1.  3.]
 [ 2.  5.]
 [ 4.  7.]]
Updated matrix amat:
[[ 3.  1.]
 [ 5.  2.]
 [ 7.  4.]]
```

In a similar manner, exchange of two rows is performed by calling function *exchange_row*, which is defined in program `tmat8r.py`

The comparison of two matrices using relational operations is performed by calling the Numpy functions or directly using the standard operators `==`, `!=`, `>`, `>=`, `<`, and `<=`. These are element-wise operations on the matrices. Function *array_equal* returns the single truth value *True* if two arrays have the same shape and elements and *False* otherwise.

The following program is stored in file `tmat9.py` and applies the various relational operators on matrices. In line 9, function *array_equal* is called to compare

the shape and elements of matrices *amat* and *bmat*. In the rest of the program the relational operators are directly applied.

```
 1 import numpy as np
 2 print "Relational operations on  matrices"
 3 amat = np.array([[1.0, 3.0], [2.0, 5.0], [4.0, 7.0]])
 4 print "Matrix amat: ", amat.shape
 5 print amat
 6 bmat = np.copy(amat)
 7 print "Matrix bmat: "
 8 print bmat
 9 flageq = np.array_equal(amat, bmat)
10 print "Flag matrix equal shape and elements: ", flageq
11 flageq2 = np.equal(amat, bmat) # or amat == bmat
12 print "Flag equal: ", flageq2
13 flaggt = np.greater(amat, bmat)
14 print "Flag greater: ", flaggt
15 flaggte = amat >= bmat
16 print "Flag greater equal: ", flaggte
17 flaglt = amat < bmat
18 print "Flag less than: ", flaglt
19 flagle = amat <= bmat
20 print "Flag less equal: ", flagle
```

Executing the Python interpreter and running the program in file `tmat9.py`, yields the following output.

```
$ python tmat9.py
Relational operations on  matrices
Matrix amat:  (3, 2)
[[ 1.  3.]
 [ 2.  5.]
 [ 4.  7.]]
Matrix bmat:
[[ 1.  3.]
 [ 2.  5.]
 [ 4.  7.]]
Flag matrix equal shape and elements:  True
Flag equal2:  [[ True  True]
 [ True  True]
 [ True  True]]
Flag greater:  [[False False]
 [False False]
```

```
 [False False]]
Flag greater equal:  [[ True  True]
 [ True  True]
 [ True  True]]
Flag less than:  [[False False]
 [False False]
 [False False]]
Flag less equal:  [[ True  True]
 [ True  True]
 [ True  True]]
```

Relational operations can also be applied to evaluate a boolean expression on the elements or to select the elements that result from a relational expression. The following program is stored in file `tmat9b.py` and applies the various relational operators on a matrix. In line 6, the boolean relational expression `>= 1.25` is applied on the elements of matrix *amat* and results in a matrix of boolean values. In line 8, the boolean expression is applied on matrix *amat* to select the elements that make the boolean expression *True*. Lines 10 and 11, applies the boolean expression only to column 0 of matrix *amat*.

```
 1 import numpy as np
 2 print "Relational operations on elements with specific
     values"
 3 amat = np.array([[1.0, 3.0], [2.0, 5.0], [4.0, 7.0]])
 4 print "Matrix amat: ", amat.shape
 5 print amat
 6 bmat = amat >= 1.25
 7 print "Elements >= 1.25 in amat: ", bmat
 8 cmat = amat[amat >= 1.25]
 9 print "Elements >= 1.25 in amat: ", cmat
10 dmat = amat[0,:]
11 emat = dmat[dmat >= 1.25]
12 print "Elements >= 1.25 in amat in row 0: ", emat
```

Executing the Python interpreter and running the program in file `tmat9b.py`, yields the following output.

```
$ python tmat9b.py
Relational operations on elements with specific values
Matrix amat:  (3, 2)
[[ 1.  3.]
 [ 2.  5.]
```

```
 [ 4.  7.]]
Elements >= 1.25 in amat:  [[False  True]
 [ True  True]
 [ True  True]]
Elements >= 1.25 in amat:  [ 3.  2.  5.  4.  7.]
Elements >= 1.25 in amat in row 0:  [ 3.]
```

Numpy function *amax* computes the maximum value stored in the specified matrix. The following function call gets the maximum value of all elements in matrix p and assigns this value to variable x. When the *axis* is specified as the second argument in the function call, it indicates that the maximum value is computed every column (`axis=0`) or in every row (`axis=1`). Therefore in the following example, xc and xr are vectors.

```
x = numpy.amax(p)
xc = numpy.amax(p, axis=0) # max in columns
xr = numpy.amax(p, axis=1) # max in columns
```

In addition to the maximum value in a matrix, the index of the element with that value may be desired in every row and every column. Calling function *argmax* computes the index values of the element with the maximum value in the specified in every column (`axis=0`) matrix and in every row (`axis=1`). In the following function call, the index values of the element with the maximum value in matrix p in every column are stored in vector *idxc* and the index values of maximum in every row in vector *idxr*.

```
idxc = numpy.argmax (p, axis=0)
idxr = numpy.argmax (p, axis=1)
```

In a similar manner, function *amin* computes the maximum value stored in the specified matrix. Function *argmin* gets the index values (row and column) of the minimum value in the specified matrix.

Listing 3 shows a Python program stored in file `array2d_ops.py` that performs the operations discussed for computing maximum values in a matrix.

```
1 import numpy as np
2 # Maximun values and indices in a matrix
3 # Program: array2d_ops.py
4 print "Sum, maximum values in a matrix"
5 amat = np.array([[3.11, 5.12, 2.13], [1.21, 8.22, 5.23],
```

```
     [6.77, 2.88, 7.55]])
 6 print "Array amat: "
 7 print amat
 8 suma = np.sum(amat)
 9 print "Sum of all elements of amat: ", suma
10 sumac = np.sum(amat, axis=0)
11 print "Sum of columns: ", sumac
12 sumar = np.sum(amat, axis=1)
13 print "Sum of rows: ", sumar
14 maxa = np.amax(amat)
15 print "Maximum of all elements of amat: ", maxa
16 maxac = np.amax(amat, axis=0)
17 print "Maximum of columns: ", maxac
18 maxar = np.amax(amat, axis=1)
19 print "Maximum of rows: ", maxar
20 idxc = np.argmax(amat, axis=0)
21 print "Indices of maximum in columns: ", idxc
22 idxr = np.argmax(amat, axis=1)
23 print "Indices of minimum in rows: ", idxr
```

Executing the Python interpreter and running the program in file `array2d_ops.py`, yields the following output.

```
$ python array2d_ops.py
Sum, maximum values in a matrix
Array amat:
[[ 3.11  5.12  2.13]
 [ 1.21  8.22  5.23]
 [ 6.77  2.88  7.55]]
Sum of all elements of amat:  42.22
Sum of columns:  [ 11.09  16.22  14.91]
Sum of rows:  [ 10.36  14.66  17.2 ]
Maximum of all elements of amat:  8.22
Maximum of columns:  [ 6.77  8.22  7.55]
Maximum of rows:  [ 5.12  8.22  7.55]
Indices of maximum in columns:  [2 1 2]
Indices of minimum in rows:  [1 1 2]
```

The *linalg* module of the Numpy package provides function *inv* that computes the inverse of a matrix. Function *transpose* rearranges the the matrix by permuting the rows and columns of a matrix. The following program stored in file `array2d_ops2.py` illustrates the use of these functions.

```
 1 import numpy as np
 2 # Inverse and transpose of a matrix
 3 # Program: array2d_ops2.py
 4 print "Inverse and transpose of a matrix"
 5 amat = np.array([[3.11, 5.12, 2.13], [1.21, 8.22, 5.23],
     [6.77, 2.88, 7.55]])
 6 print "Array amat: "
 7 print amat
 8 imat = np.linalg.inv(amat)
 9 print "Inverse of matrix amat: "
10 print imat
11 tamat = np.transpose(amat)
12 print "Transpose of matrix amat: "
```

Executing the Python interpreter and running the program in file `array2d_ops2.py`, yields the following output.

```
$ python array2d_ops2.py
Inverse and transpose of a matrix
Array amat:
[[ 3.11  5.12  2.13]
 [ 1.21  8.22  5.23]
 [ 6.77  2.88  7.55]]
Inverse of matrix amat:
[[ 0.27717054 -0.19179357  0.0546632 ]
 [ 0.15493469  0.053433   -0.0807239 ]
 [-0.30763662  0.15159675  0.11422715]]
Transpose of matrix amat:
[[ 3.11  1.21  6.77]
 [ 5.12  8.22  2.88]
 [ 2.13  5.23  7.55]]
```

Function *det* of the *linalg* module of the Numpy package computes the determinant of a square matrix. Function *dot* performs the matrix multiplication of two matrices. Recall that in this operation, the matrices have to be aligned, the number of columns of the first matrix must be equal to the number of rows in the second matrix. The first matrix is $n \times m$, the second matrix is $m \times k$, the resulting matrix is $n \times k$.

In the following program stored in file `array2d_ops3.py` creates a 3×3 matrix, *amat*, in line 5. The determinant of matrix *amat* is computed in line 8. A 3×2 matrix, *bmat*, is created in line 10. The dot product of matrix *amat* and matrix *bmat* is computed in line 12.

```
 1 import numpy as np
 2 # Matrix determinant and multiplication
 3 # Program: array2d_ops2.py
 4 print "Matrix determinant and multiplication "
 5 amat = np.array([[3.11, 5.12, 2.13], [1.21, 8.22, 5.23],
     [6.77, 2.88, 7.55]])
 6 print "Array amat: "
 7 print amat
 8 adet = np.linalg.det(amat)
 9 print "Determinant of amat: " , adet
10 bmat = np.array([[4.5, 6.5], [8.5, 1.5], [5.25, 7.35]])
11 print bmat
12 dpmat = np.dot(amat, bmat)
13 print "Dot product of amat and bmat: "
14 print dpmat
```

Executing the Python interpreter and running the program in file `array2d_ops3.py`, yields the following output.

```
$ python array2d_ops3.py
Matrix determinant and multiplication
Array amat:
[[ 3.11  5.12  2.13]
 [ 1.21  8.22  5.23]
 [ 6.77  2.88  7.55]]
Determinant of amat:  169.56564
[[ 4.5   6.5 ]
 [ 8.5   1.5 ]
 [ 5.25  7.35]]
Dot product of amat and bmat:
[[  68.6975   43.5505]
 [ 102.7725   58.6355]
 [  94.5825  103.8175]]
```

3 Solving Systems of Linear Equations

Several methods exist on how to solve systems of linear equations applying vectors and matrices. Some of these methods are: substitution, cancellation, and matrix manipulation. A system of n linear equations can be expressed by the general equations:

$$\begin{array}{lllll} a_{11}x_1 & + a_{12}x_2 & + \ldots & + a_{1n}x_n & = b_1 \\ a_{21}x_1 & + a_{22}x_2 & + \ldots & + a_{2n}x_n & = b_2 \\ \vdots & \vdots & & \vdots & \vdots \\ a_{n1}x_1 & + a_{n2}x_2 & + \ldots & + a_{nn}x_n & = b_n \end{array}$$

This system of linear equations is more conveniently expressed in matrix form in the following manner:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{2m} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} \tag{1}$$

This can also be expressed in a more compact matrix form as: $AX = B$. Matrix A is the coefficients matrix (of the variables x_i for $i = 1 \ldots n$). X is the vector of unknowns x_i, and B is the vector of solution. Consider a simple linear problem that consists of a system of three linear equations ($n = 3$):

$$\begin{array}{llll} 5x_1 & +2x_2 & +x_3 & = 25 \\ 2x_1 & +x_2 & +3x_3 & = 12 \\ -x_1 & +x_2 & +2x_3 & = 5 \end{array}$$

In matrix form, this system of three linear equations can be written in the following manner:

$$\begin{bmatrix} 5 & 2 & 1 \\ 2 & 1 & 3 \\ -1 & 1 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 25 \\ 12 \\ 5 \end{bmatrix} \tag{2}$$

Matrix A is a square ($n \times n$) of coefficients, X is a vector of size n, and B is the solution vector also of size n.

$$A = \begin{bmatrix} 5 & 2 & 1 \\ 2 & 1 & 3 \\ -1 & 1 & 2 \end{bmatrix} \quad X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad B = \begin{bmatrix} 25 \\ 12 \\ 5 \end{bmatrix} \tag{3}$$

Decomposing the coefficient matrix, A is the main technique used to compute the determinant, matrix inversion, and the solution a set of linear equations. Common numerical methods used are:

- Gaussian Elimination

- LU Decomposition
- SV Decomposition
- QR Decomposition

The Numpy function *linalg.solve* finds the solution to a linear matrix equation, or system of linear scalar equations, given the coefficient matrix a and the vector of dependent variable b. The function applies LU decomposition to the specified square matrix. The function returns the vector x, which has the same dimension as vector b.

The following program, stored in file `linsolve.py`, creates a square matrix *amat* in line 6. The solution vector b is defined in line 10. The vector of unknowns x is created in line 12 by calling function *linalg.solve*.

```
 1 import numpy as np
 2 # Solution to a set of linear equations
 3 # A X = B
 4 # Program: linsol.py
 5 print "Solving a set of linear equations"
 6 amat = np.array([[3.11, 5.12, 2.13], [1.21, 8.22, 5.23],
    [6.77, 2.88, 7.55]])
 7 print "Matrix a: "
 8 print amat
 9 print "Vector b: "
10 b = np.array([4.5, 6.5, 8.5])
11 print b
12 x = np.linalg.solve(amat, bmat)
13 print "Vector x: "
14 print x
```

Executing the Python interpreter and running the program in file `linsolve.py`, yields the following output.

```
$ python linsolve.py
Solving a set of linear equations
Matrix a:
[[ 3.11  5.12  2.13]
 [ 1.21  8.22  5.23]
 [ 6.77  2.88  7.55]]
Vector b:
[ 4.5  6.5  8.5]
Vector x:
[ 0.46524638  0.35836741  0.57194488]
```

4 Industrial Mixtures in Manufacturing

Manufacturing of various products require specified amounts of the several materials to produce products of acceptable quality. The optimization of such mixtures is discussed in detail in the chapters on linear optimization.

For example, every unit weight (in grams) of product A requires 0.59g of material P, 0.06g of material Q, 0.037g of material R, and 0.313g of material S. This can written in a general expression of the form:

$$A = b_1 + b_2 + b_3 + b_4$$

In the expression, b_1 is the amount material P, b_2 is the amount of material Q, b_3 is the amount of material R, and b_4 is the amount of material S. A similar expression can be used for the mix required for the manufacturing of product B, so on.

The materials needed in the manufacturing process are acquired as substances that contain various amounts of the materials mentioned previously. For example, the following table provides data on the unit content (1 g) of the substances used to obtain the materials needed for manufacturing of products A and B.

Substance	Material P	Material Q	Material R	Material S
S1	0.67	0.2	0.078	0.135
S2	0.87	0.04	0.0029	0.0871
S3	0.059	0.018	0.059	0.864
S4	0.72	0.02	0.03	0.23

From the data in the table, the expression that is used to compute the amount b_1 of material P using y_1 grams of $S1$, y_2 grams of $S2$, y_3 grams of $S3$, and y_4 grams of $S4$ is:

$$b_1 = 0.67\, y_1 + 0.87\, y_2 + 0.059\, y_3 + 0.72\, y_4$$

In a similar manner, the expression that is used to compute the amount b_2 of material Q using y_1 grams of $S1$, y_2 grams of $S2$, y_3 grams of $S3$, and y_4 grams of $S4$ is:

$$b_2 = 0.2\, y_1 + 0.04\, y_2 + 0.018\, y_3 + 0.02\, y_4$$

Similar expressions can be written for computing the amount of material R and S needed for the manufacturing of product A.

The system of 4 linear equations that corresponds to this problem can be expressed by:

$$\begin{array}{rrrrcl} 0.67\ y_1 & +\ 0.87\ y_2 & +\ 0.059\ y_3 & +\ 0.72\ y_4 & = & b_1 \\ 0.2\ y_1 & +\ 0.04\ y_2 & +\ 0.018\ y_3 & +\ 0.02\ y_4 & = & b_2 \\ 0.078\ y_1 & +\ 0.0029\ y_2 & +\ 0.059\ y_3 & +\ 0.03\ y_4 & = & b_3 \\ 0.135\ y_1 & +\ 0.0871\ y_2 & +\ 0.864\ y_3 & +\ 0.23\ y_4 & = & b_4 \end{array}$$

This system of linear equations is more conveniently expressed in matrix form in the following manner:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} \tag{4}$$

This can also be expressed in a more compact matrix form as: $AY = B$. Matrix A is the coefficients matrix (of the variables y_i for $i = 1 \ldots n$). Y is the vector of unknowns y_i, and B is the vector of solution.

The Python program `mixmanuf.py` solves the problem by solving the system of four equations. This provides the solution by computing the values of vector Y, which are the quantities necessary of substances S1, S2, S3, and S4 that are required to produce 1 gram of product A. The following listing shows running the program with the Python interpreter.

```
$ python mixmanuf.py
Mix for Manufactoring Products
Solving the set of linear equations
Matrix A:
[[ 0.67    0.87    0.059   0.72  ]
 [ 0.2     0.04    0.018   0.02  ]
 [ 0.078   0.0029  0.059   0.03  ]
 [ 0.135   0.0871  0.864   0.23  ]]
Vector B:
[ 0.59   0.06   0.037  0.313]
Vector Y:
[ 0.19145029  0.31650837  0.23670601  0.23944495]
```