TACC
TEXAS ADVANCED COMPUTING CENTER
WWW.TACC.UTEXAS.EDU

TEXAS
The University of Texas at Austin

Numerical Python

S. Charlie Dey, Director of Training and Professional Development

Science on the Cloud, 2019

Linear Algebra

Applications

- Matrices in Engineering, such as a line of springs.
- Graphs and Networks, such as analyzing networks.
- Markov Matrices, Population, and Economics, such as population growth.
- Linear Programming, the simplex optimization method.
- Fourier Series: Linear Algebra for functions, used widely in signal processing.
- Linear Algebra for statistics and probability, such as least squares for regression.
- Computer Graphics, such as the various translation, rescaling and rotation of images.

Linear Algebra

Linear algebra is about linear combinations.

Using math on columns of numbers called *vectors* and arrays of numbers called *matrices* to create new columns and arrays of numbers.

Linear algebra is the study of lines and planes, vector spaces and mappings that are required for linear transforms.

Linear Algebra

Linear algebra is the mathematics of data.
Matrices and vectors are the language of data.

Let's look at the following:

```
y = 4 * x + 1
```

describes a line on a two-dimensional graph

Linear Algebra

Linear algebra is the mathematics of data.

Matrices and vectors are the language of data.

Let's look at the following:

```
y = 0.1 * x1 + 0.4 * x2
y = 0.3 * x1 + 0.9 * x2
```

line up a system of equations with the same form with two or more unknowns

Linear Algebra

Linear algebra is the mathematics of data.
Matrices and vectors are the language of data.

Let's look at the following:

```
1 = 0.1 * x1 + 0.4 * x2
3 = 0.3 * x1 + 0.9 * x2
```

line up a system of equations with the same form with two or more unknowns

Linear Algebra

Linear algebra is the mathematics of data.

Matrices and vectors are the language of data.

Let's look at the following, Ax = b :

```
 5 = 0.1 * x1 + 0.4 * x2 + x3
10 = 0.3 * x1 + 0.9 * x2 + 2.0 * x3
 3 = 0.2 * x1 + 0.3 * x2 - .5 * x3
```

Is there a x1, x2, x3 that solves this system?

Linear Algebra

Gaussian Elimination

The goals of Gaussian elimination are to make the upper-left corner element a 1

use elementary row operations to get 0s in all positions underneath that first 1

get 1s for leading coefficients in every row diagonally from the upper-left to lower-right corner, and get 0s beneath all leading coefficients.

you eliminate all variables in the last row except for one, all variables except for two in the equation above that one, and so on and so forth to the top equation, which has all the variables. Then use back substitution to solve for one variable at a time by plugging the values you know into the equations from the bottom up..

Linear Algebra

Gaussian Elimination, Rules

- **You can multiply any row by a constant (other than zero).**
- $-2r_3 \rightarrow r_3$
- **You can switch any two rows.**
- $r_1 \leftrightarrow r_2$
- **You can add two rows together.**
- $r_1 + r_2 \rightarrow r_2$

Linear Algebra

Transpose

A defined matrix can be transposed, which creates a new matrix with the number of columns and rows flipped.

This is denoted by the superscript “T” next to the matrix.

An invisible diagonal line can be drawn through the matrix from top left to bottom right on which the matrix can be flipped to give the transpose.

Linear Algebra

Inversion

Matrix inversion is a process that finds another matrix that when multiplied with the matrix, results in an identity matrix.

Given a matrix A, find matrix B, such that AB or BA = In.

The operation of inverting a matrix is indicated by a -1 superscript next to the matrix; for example, A^-1. The result of the operation is referred to as the inverse of the original matrix; for example, B is the inverse of A.

Linear Algebra

Trace

A trace of a square matrix is the sum of the values on the main diagonal of the matrix (top-left to bottom-right).

Linear Algebra

Determinant

The determinant of a square matrix is a scalar representation of the volume of the matrix.

> *The determinant describes the relative geometry of the vectors that make up the rows of the matrix. More specifically, the determinant of a matrix A tells you the volume of a box with sides given by rows of A.*

— Page 119, No Bullshit Guide To Linear Algebra, 2017

Linear Algebra

Matrix Rank

The rank of a matrix is the estimate of the number of linearly independent rows or columns in a matrix.

Linear Algebra - Matrix Arithmetic

Matrix Addition

Two matrices with the same dimensions can be added together to create a new third matrix.

```
C = A + B
```

```
C[0,0] = A[0,0] + B[0,0]
C[1,0] = A[1,0] + B[1,0]
C[2,0] = A[2,0] + B[2,0]
C[0,1] = A[0,1] + B[0,1]
C[1,1] = A[1,1] + B[1,1]
C[2,1] = A[2,1] + B[2,1]
```

Linear Algebra - Matrix Arithmetic

Matrix Subtraction

Similarly, one matrix can be subtracted from another matrix with the same dimensions.

```
C = A - B
```

```
C[0,0] = A[0,0] - B[0,0]
C[1,0] = A[1,0] - B[1,0]
C[2,0] = A[2,0] - B[2,0]
C[0,1] = A[0,1] - B[0,1]
C[1,1] = A[1,1] - B[1,1]
C[2,1] = A[2,1] - B[2,1]
```

Linear Algebra - Matrix Arithmetic

Matrix Multiplication (Hadamard Product)

Two matrices with the same size can be multiplied together, and this is often called element-wise matrix multiplication or the Hadamard product.

It is not the typical operation meant when referring to matrix multiplication, therefore a different operator is often used, such as a circle “o”.

```
C = A o B
C[0,0] = A[0,0] * B[0,0]
C[1,0] = A[1,0] * B[1,0]
C[2,0] = A[2,0] * B[2,0]
C[0,1] = A[0,1] * B[0,1]
C[1,1] = A[1,1] * B[1,1]
C[2,1] = A[2,1] * B[2,1]
```

Linear Algebra - Matrix Arithmetic

Matrix Division

One matrix can be divided by another matrix with the same dimensions.

```
C = A / B
C[0,0] = A[0,0] / B[0,0]
C[1,0] = A[1,0] / B[1,0]
C[2,0] = A[2,0] / B[2,0]
C[0,1] = A[0,1] / B[0,1]
C[1,1] = A[1,1] / B[1,1]
C[2,1] = A[2,1] / B[2,1]
```

Linear Algebra - Matrix Arithmetic

Matrix-Matrix Multiplication (Dot Product)

Matrix multiplication, also called the matrix dot product is more complicated than the previous operations and involves a rule as not all matrices can be multiplied together.

> *One of the most important operations involving matrices is multiplication of two matrices. The matrix product of matrices A and B is a third matrix C. In order for this product to be defined, A must have the same number of columns as B has rows. If A is of shape $m \times n$ and B is of shape $n \times p$, then C is of shape $m \times p$.*

— Page 34, Deep Learning, 2016.

Linear Algebra - Matrix Arithmetic

Matrix-Matrix Multiplication (Dot Product)

```
        a11, a12
A =     a21, a22
        a31, a32

        b11, b12
B =     b21, b22

        a11 * b11  +  a12 * b21, a11 * b12  +  a12 * b22
C =     a21 * b11  +  a22 * b21, a21 * b12  +  a22 * b22
        a31 * b11  +  a32 * b21, a31 * b12  +  a32 * b22
```

Numerical Linear Algebra, Two Different Approaches

- Solve $Ax = b$
- Direct methods:
  - Deterministic
  - Exact, up to machine precision
  - Expensive (in time and space)
- Iterative methods:
  - Only approximate
  - Cheaper in space and (possibly) time
  - Convergence not guaranteed

Iterative Methods

Choose any x_0 and repeat

$$x^{k+1} = Bx^k + c$$

until

$$\|x^{k+1} - x^k\|_2 < \epsilon$$

or until

$$\frac{\|x^{k+1} - x^k\|_2}{\|x^k\|} < \epsilon$$

Example of Iterative Solution

Example system

$$\begin{pmatrix} 10 & 0 & 1 \\ 1/2 & 7 & 1 \\ 1 & 0 & 6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 21 \\ 9 \\ 8 \end{pmatrix}$$

with solution (2,1,1)

Suppose you know (physics) that solution components are roughly the same size, and observe the dominant size of the diagonal, then

$$\begin{pmatrix} 10 & & \\ & 7 & \\ & & 6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 21 \\ 9 \\ 8 \end{pmatrix}$$

might be a good approximation: solution (2.1, 9/7, 8/6)

Iterative Example

Example system

$$\begin{pmatrix} 10 & 0 & 1 \\ 1/2 & 7 & 1 \\ 1 & 0 & 6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 21 \\ 9 \\ 8 \end{pmatrix}$$

with solution (2,1,1)

Also easy to solve:

$$\begin{pmatrix} 10 & & \\ 1/2 & 7 & \\ 1 & 0 & 6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 21 \\ 9 \\ 8 \end{pmatrix}$$

with solution (2.1, 7.95/7, 5.95/6)

Iterative Example

- Instead of solving $Ax = b$ we solved $L\tilde{x} = b.$
- Look for the missing part: $\tilde{x} = x + \Delta x$, then $A\Delta x = A\tilde{x} - b \equiv r$
- Solve again $L\widetilde{\Delta x} = r$ and update $\tilde{\tilde{x}} = \tilde{x} - \widetilde{\Delta x}$

iteration	1	2	3
x_1	2.1000	2.0017	2.000028
x_2	1.1357	1.0023	1.000038
x_3	0.9833	0.9997	0.999995

- Two decimals per iteration. *This is not typical*
- Exact system solving: $O(n^3)$ cost; iteration: $O(n^2)$ per iteration. Potentially cheaper if the number of iterations is low.

Abstract Presentation

- To solve $Ax = b$; too expensive; suppose $K \approx A$ and solving $Kx = b$ is possible
- Define $Kx_0 = b$, then error correction $x_0 = x + e_0$, and $A(x_0 - e_0) = b$
- so $Ae_0 = Ax_0 - b = r_0$; this is again unsolvable, so
- $K\tilde{e}_0$ and $x_1 = x_0 - \tilde{e}_0$
- Now iterate: $e_1 = x_1 - x$, $Ae_1 = Ax_1 - b = r_1$ et cetera

Error Analysis

- One step

$$\begin{aligned} r_1 &= Ax_1 - b = A(x_0 - \tilde{e}_0) - b && (2)\\ &= r_0 - AK^{-1}r_0 && (3)\\ &= (I - AK^{-1})r_0 && (4) \end{aligned}$$

- Inductively:

$$r_n = (I - AK^{-1})^n r_0 \text{ so } r_n \downarrow 0 \text{ if } |\lambda(I - AK^{-1})| < 1$$

- Geometric reduction (or amplification!)
- This is 'stationary iteration': every iteration step the same. Simple analysis, limited applicability

Computationally

If $A = K - N$

then $Ax = b \Longrightarrow Kx = Nx + b \Longrightarrow Kx_{i+1} = Nx_i + b$

(because $Kx = Nx + b$ is a "fixed point" of an iteration)

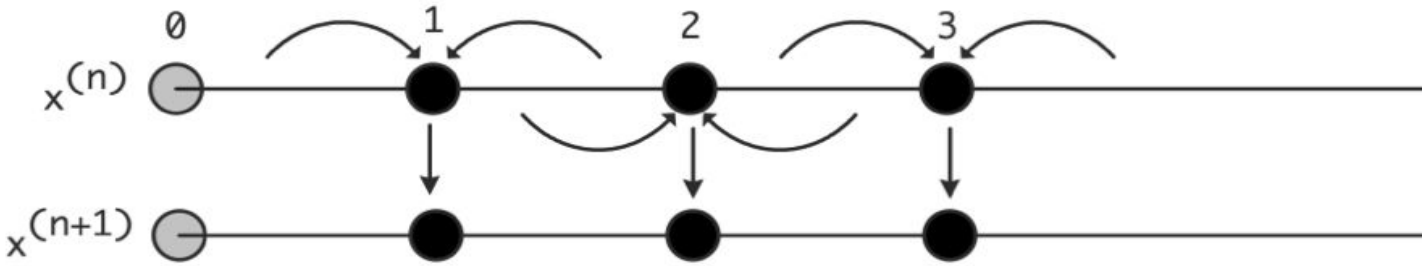
Equivalent to the above, and you don't actually need to form the residual

Choice of K

- The closer K is to A, the faster the convergence
- Diagonal and lower triangular choice mentioned above: let $A = D_A + L_A + U_A$ be a splitting into diagonal, lower triangular, upper triangular part, then
- Jacobi method: $K = D_A$ (diagonal part),
- Gauss-Seidel method: $K = D_A + L_A$ (lower triangle, including diagonal)
- SOR method:

$$K = \omega D_A + L_A$$

Jacobi in Pictures



Jacobi Method

Given a square system of *n* linear equations:

$$A\mathbf{x} = \mathbf{b}$$

where:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \qquad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

Jacobi Method

Then *A* can be decomposed into a diagonal component *D*, and the remainder *R*:

$$A = D+R \qquad \text{where} \qquad D = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix} \text{ and } R = \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ a_{21} & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{bmatrix}.$$

Jacobi Method

The solution is then obtained iteratively via

$$\mathbf{x}^{(k+1)} = D^{-1}(\mathbf{b} - R\mathbf{x}^{(k)}),$$

where $\mathbf{x}^{(k)}$ is the kth approximation or iteration of $\mathbf{x}$ and $\mathbf{x}^{(k+1)}$ is the next or $k + 1$ iteration of $\mathbf{x}$. The element-based formula is thus:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right), \quad i = 1, 2, \ldots, n.$$

The computation of $x_i^{(k+1)}$ requires each element in $\mathbf{x}^{(k)}$ except itself. Unlike the Gauss–Seidel method, we can't overwrite $x_i^{(k)}$ with $x_i^{(k+1)}$, as that value will be needed by the rest of the computation. The minimum amount of storage is two vectors of size n.

Jacobi Method

Algorithm.

- Choose your initial guess, x[0]
- Start iterating, k=0
  - While not converged do
    - Start your i-loop (for i = 1 to n)
      - sigma = 0
        - Start your j-loop (for j = 1 to n)
          - If j not equal to i
            - sigma = sigma + a[i][j] * $x[j]_k$
        - End j-loop
      - $x[i]_k$ = (b[i] – sigma)/a[i][i]
    - End i-loop
  - Check for convergence
- Iterate k, ie. k = k+1

What about the Lower and Upper Triangles?

If we write D, L, and U for the diagonal, strict lower triangular and strict upper triangular and parts of A, respectively,

$$D=\begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & a_{nn} \end{bmatrix}, \quad L=\begin{bmatrix} 0 & 0 & \cdots & 0 \\ a_{21} & 0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ a_{n1} & \cdots & a_{nn-1} & 0 \end{bmatrix} \text{ and } U=\begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ 0 & 0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & a_{n-1n} \\ 0 & \cdots & 0 & 0 \end{bmatrix},$$

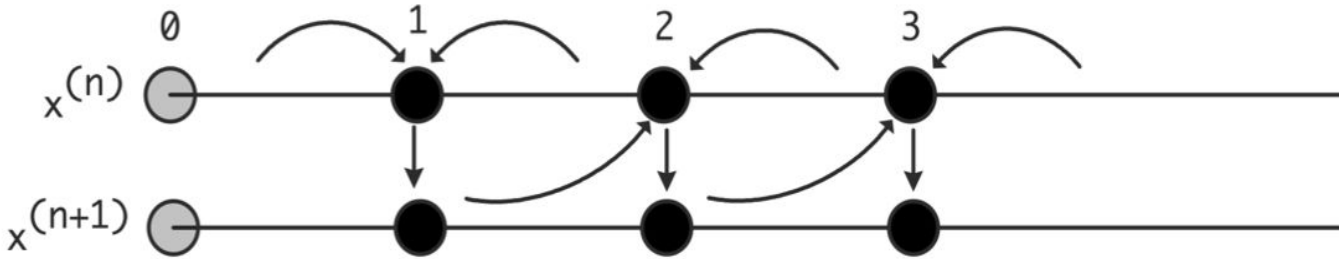
then Jacobi's Method can be written in matrix-vector notation as

$$D\boldsymbol{x}^{(k+1)} + (L+U)\boldsymbol{x}^{(k)} = \boldsymbol{b}$$

so that

$$\boldsymbol{x}^{(k+1)} = D^{-1}[(-L-U)\boldsymbol{x}^{(k)} + \boldsymbol{b}].$$

GS in Pictures



Gauss-Seidel

$K = D_A + L_A$

$Ax=b \Rightarrow (D_A+L_A+U_A)x=b$

$(D_A+L_A)x^{k+1} = -U_A x^k + b$

$\{D_A\}_{ii}=a_{ii} \quad \{U_A \text{ or } L_A\}_{ij}=a_{ij} \quad i \neq j$

Algorithm:

for k = 1, ... until convergence, do:

for i = 1 ... n:

$$//a_{ii}x_i^{(k+1)} + \sum_{j<i} a_{ij}x_j^{(k+1)}) = \sum_{j>i} a_{ij}x_j^{(k)} + b_i \Rightarrow$$
$$x_i^{(k+1)} = a_{ii}^{-1}(-\sum_{j<i} a_{ij}x_j^{(k+1)}) - \sum_{j>i} a_{ij}x_j^{(k)} + b_i)$$

Implementation:

for k = 1, ... until convergence, do:

for i = 1 ... n:

$$x_i = a_{ii}^{-1}(-\sum_{j \neq i} a_{ij}x_j + b_i)$$

Gauss-Seidel Method

Given a square system of *n* linear equations:

$$A\mathbf{x} = \mathbf{b}$$

where:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \qquad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

Gauss-Seidel Method

$$A = L_* + U \qquad \text{where} \qquad L_* = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \quad U = \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ 0 & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}.$$

The system of linear equations may be rewritten as:

$$L_* \mathbf{x} = \mathbf{b} - U\mathbf{x}$$

Gauss-Seidel Method

It is defined by the iteration

$$L_* \mathbf{x}^{(k+1)} = \mathbf{b} - U\mathbf{x}^{(k)},$$

where $\mathbf{x}^{(k)}$ is the *k*th approximation or iteration of $\mathbf{x}$, $\mathbf{x}^{k+1}$ is the next or $k+1$ iteration of $\mathbf{x}$, and the matrix A is decomposed into a lower triangular component L_*, and a strictly upper triangular component U: $A = L_* + U$.[2]

Which gives us: $\mathbf{x}^{(k+1)} = L_*^{-1}(\mathbf{b} - U\mathbf{x}^{(k)}).$

However, by taking advantage of the triangular form of L_*, the elements of $\mathbf{x}^{(k+1)}$ can be computed sequentially using forward substitution:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j<i} a_{ij} x_j^{(k+1)} - \sum_{j>i} a_{ij} x_j^{(k)} \right), \quad i, j = 1, 2, \ldots, n.$$

Gauss-Seidel Method

Algorithm:

- Choose your initial guess, theta[0]
- While not converged do:
  - Start your i-loop (for i = 1 to n)
    - sigma = 0
    - Start your j-loop (for j = 1 to n)
      - If j not equal to i
        - sigma = sigma + a[i][j] * theta[j]
      - End j-loop
      - theta[i] = (b[i] – sigma)/a[i][i]
    - End i-loop
  - Check for convergence
- iterate

Stopping Tests

When to stop converging? Can size of the error be guaranteed?

- Direct tests on error $e_n = x - x_n$ impossible; two choices
- Relative change in the computed solution small:

$$\|x_{n+1} - x_n\| / \|x_n\| < \epsilon$$

- Residual small enough:

$$\|r_n\| = \|Ax_n - b\| < \epsilon$$

Without proof: both imply that the error is less than some other ϵ'

Python – NumPy

"Numerical Python"

open source extension module for Python
provides fast precompiled functions for mathematical and numerical routines
adds powerful data structures for efficient computation of multi-dimensional arrays and matrices.

NumPy, First Steps

Let build a simple list, turn it into a numpy array and perform some simple math.

```
import numpy as np
cvalues = [25.3, 24.8, 26.9, 23.9]
C = np.array(cvalues)
print(C)
```

NumPy, First Steps

Let build a simple list, turn it into a numpy array and perform some simple math.

```
print(C * 9 / 5 + 32)
```

vs.

```
fvalues = [ x*9/5 + 32 for x in cvalues]
print(fvalues)
```

NumPy, Cooler things

```
import time
size_of_vec = 1000
def pure_python_version():
    t1 = time.time()
    X = range(size_of_vec)
    Y = range(size_of_vec)
    Z = []
    for i in range(len(X)):
        Z.append(X[i] + Y[i])
    return time.time() - t1
```

```
def numpy_version():
    t1 = time.time()
    X = np.arange(size_of_vec)
    Y = np.arange(size_of_vec)
    Z = X + Y
    return time.time() - t1
```

NumPy, Cooler things

Let's see which is faster.

```
t1 = pure_python_version()
t2 = numpy_version()
print(t1, t2)
```

NumPy, Multi-Dimension Arrays

```
A = np.array([ [3.4, 8.7, 9.9],
               [1.1, -7.8, -0.7],
               [4.1, 12.3, 4.8]])
print(A)
print(A.ndim)


B = np.array([ [[111, 112], [121, 122]],
               [[211, 212], [221, 222]],
               [[311, 312], [321, 322]] ])
print(B)
print(B.ndim)
```

NumPy, Multi-Dimension Arrays

The shape function:

```
x = np.array([ [67, 63, 87],
               [77, 69, 59],
               [85, 87, 99],
               [79, 72, 71],
               [63, 89, 93],
               [68, 92, 78]])
print(np.shape(x))
```

NumPy, Multi-Dimension Arrays

The shape function can also *change* the shape:

```
x.shape = (3, 6)
print(x)


x.shape = (2, 9)
print(x)
```

NumPy, Multi-Dimension Arrays

A couple more examples of shape:

```
x = np.array(42)
print(np.shape(x))

B = np.array([ [[111, 112], [121, 122]],
               [[211, 212], [221, 222]],
               [[311, 312], [321, 322]] ])
print(B.shape)
```

NumPy, Multi-Dimension Arrays

indexing:

```
F = np.array([1, 1, 2, 3, 5, 8, 13, 21])

# print the first element of F, i.e. the element with the index 0
print(F[0])

# print the last element of F
print(F[-1])

B = np.array([ [[111, 112], [121, 122]],
               [[211, 212], [221, 222]],
               [[311, 312], [321, 322]] ])
print(B[0][1][0])
```

NumPy, Multi-Dimension Arrays

slicing:

```
A = np.array([
[11,12,13,14,15],
[21,22,23,24,25],
[31,32,33,34,35],
[41,42,43,44,45],
[51,52,53,54,55]])

print(A[:3,2:])


print(A[3:,:])
```

NumPy, Multi-Dimension Arrays

function to create an identity array

```
np.identity(4)
```

NumPy, By Example

The example we will consider is a very simple (read, trivial) case of solving the 2D Laplace equation using an iterative finite difference scheme (four point averaging, Gauss-Seidel or Gauss-Jordan). The formal specification of the problem is as follows. We are required to solve for some unknown function u(x,y) such that $\nabla 2u = 0$ with a boundary condition specified. For convenience the domain of interest is considered to be a rectangle and the boundary values at the sides of this rectangle are given.

```
def TimeStep(self, dt=0.0):
     """Takes a time step using straight forward Python loops."""
     g = self.grid
     nx, ny = g.u.shape
     dx2, dy2 = g.dx**2, g.dy**2
     dnr_inv = 0.5/(dx2 + dy2)
     u = g.u
     err = 0.0
     for i in range(1, nx-1):
         for j in range(1, ny-1):
             tmp = u[i,j]
             u[i,j] = ((u[i-1, j] + u[i+1, j])*dy2 +
                       (u[i, j-1] + u[i, j+1])*dx2)*dnr_inv
             diff = u[i,j] - tmp
             err += diff*diff
     return numpy.sqrt(err)
```

NumPy, By Example

The example we will consider is a very simple (read, trivial) case of solving the 2D Laplace equation using an iterative finite difference scheme (four point averaging, Gauss-Seidel or Gauss-Jordan). The formal specification of the problem is as follows. We are required to solve for some unknown function u(x,y) such that $\nabla 2u = 0$ with a boundary condition specified. For convenience the domain of interest is considered to be a rectangle and the boundary values at the sides of this rectangle are given.

```
def numericTimeStep(self, dt=0.0):
    """Takes a time step using a NumPy expression."""
    g = self.grid
    dx2, dy2 = g.dx**2, g.dy**2
    dnr_inv = 0.5/(dx2 + dy2)
    u = g.u
    g.old_u = u.copy() # needed to compute the error.

    # The actual iteration
    u[1:-1, 1:-1] = ((u[0:-2, 1:-1] + u[2:, 1:-1])*dy2 +
                     (u[1:-1,0:-2] + u[1:-1, 2:])*dx2)*dnr_inv

    return g.computeError()
```

NumPy, Exercise

Jacobi

```
Algorithm.

* Find D, the Diagonal of of A : diag(A)
* Find R, the Remainder of A - D : A - diagflat(A)

* Choose your initial guess, x[0]
    * Start iterating, k=0
        * While not converged do
           * Start your i-loop (for i = 1 to n)
               * sigma = 0
                * Start your j-loop (for j = 1 to n)
                   * If j not equal to i
                       * sigma = sigma + a[i][j] * x[j][k]
                 * End j-loop
               * x[i]k = (b[i] – sigma)/a[i][i] : x = (b - dot(R,x)) / D
           * End i-loop
        * Check for convergence
    * Iterate k, ie. k = k+1
```

Questions? Comments?