# matalg27 User Manual

- Package matalg27
- Module name matalg27
- class name Matrix
- File matalg27.py
- format restructuredText

# Purpose

## Basic Matrix algebra in pure Python 2.7

The objective of this matrix algebra module is to provide elementary matrix operations of linear algebra, including the solution of linear equations and matrix inversion. All this is programmed in pure Python 2.7 and does not depend on other third party packages. This package is particularly useful for people planning to upgrade to Python 3.x and wishing to investigate the suitability of pure Python package for programs currently using the more elaborate systems for matrix algebra.

## Usage

First step is to import the module:

```
>>> from matalg27 import Matalg27 as _m
```

Now change to test directory and run the test program:

```
cd matalg27-tst/
testMat27.py
This works, if no failures reported...
```

In the following we are providing examples. Preferably, the examples should be entered by hand in the Python Shell of the "Idle" IDE, executed and verified. They can also be entered into a Python Shell that opens up on the terminal after Python is invoked.

At this stage all examples are solved by Python 2.7. In all likelihood they will remain valid in Python 2.x with x >= 4. A Linux platform is assumed, but the examples should run under other OS's as Python was and remains a programming environment that runs on on most OS's.

# Basic Introduction to Matrix Algebra.

## Matrices from Simultaneous Equations

For matalg27 we assume that all entries in matrices are floating point numbers, or integers, which will be automatically converted to floats. Indeed, that is the most widely used kind of matrices.

Matrices can be used to represent simultaneous linear equations. Consider an example:

```
c11*x1 + c12*x2 + c13*x3 = rhs1
c21*x1 + c22.x2 + c23*x3 = rhs2
c31*x2 + c32.x2 + c33*x3 = rhs3
```

In matrix form this can be written as three matrices:

```
[cmat] * [x] = [rhs]
```

where brackets indicate a matrix. **cmat** has 9 terms arranged in 3 columns of rows, each row with 3 entries. We say that it is a 3 by 3 matrix. [x] has 3 terms and is usually written as a column and we say that it is a 3 by 1 matrix with entries x1, x2 and x3. The (3x1) matrix x are the unknowns, whilst the (2x1) [rhs] matrix are the given data. Most frequently we will want to determine the unknowns [x] - that is the most basic and frequent operation of matrix algebra.

Consider a numerical example:

```
5 *x1 + 6 *x2 + 7 *x3 = 18
10*x1 + 12*x2 + 3 *x3 = 25
20*x1 + 17*x2 + 19*x3 = 56
```

All the numbers are floats, though in the example their values are whole numbers, so it is permissible to enter them as floats. A (3x3) matrix of is created in one statement:

```
>>> cmat = _m.enterdata(3, 3, [[5,  6, 7], [10, 12, 3], [20,  17, 19]])
  ('Echo check of enterdata',
  [[5.0, 6.0, 7.0], [10.0, 12.0, 3.0], [20.0, 17.0, 19.0]])
```

We have tacitly assumed that Matalg27 has been imported as **_m**, as shown earlier. Similarly we enter right hand sides, (3x1) matrix, rhs

```
>>> rhs = _m.enterdata(3, 1, [[18], [25], [56]])
  ('Echo check of enterdata', [[18.0], [25.0], [56.0]])
```

The data to enterdata procedure starts with the two dimensions of the matrix followed by data entries in a list of lists.

Each inner list is a row of the matrix. The automatic echo check can be disabled by specifying the last argument as False. We left it as True, so we have the verification of data in a printed format in "Echo check".

The rows and columns are numbered with a number sequence starting with zero in the usual Python manner.

It is possible to refer to each element of a matrix in the Pythonic way as **rhs[i][j]** or in a matrix customary way as **rhs[i, j]** . **It's your choice!**

## Solution of equations

We are now ready to get the solution as matrix solution:

```
>>> solution = cmat ** rhs
```

and we can print out the results

```
>>> _m.printmat('Solution =', solution)
Solution =
[1.0]
[1.0]
[1.0]
```

The simplicity of solution is not really a coincidence-we chose the rhs matrix to give a simple answer :)

As you may have noticed, there are many ways of printing a matrix. The user can choose which command is appropriate. Many commands can be classified as high and low. This is best shown with an example. Consider a matrix operation:

```
cmat = amat * bmat
```

This is a high level command. It is easy to see that cmat is the product of amat and bmat. In matrix algebra we would say that amat is post-multiplied by bmat. If you look into the program, you would find that the above high level command calls a low lever command:

```
cmat = amat.matmult(bmat)
```

I think most will agree that this is not as clear as the high level command, but we could use it directly and would get exactly the same answer. So we recommend that whenever there is a choice of a high level command, use it, rather than its lower level equivalent.

Some commands are only available in low level, as there is no equivalent in algebra of scalar numbers. For instance, for a transpose of a matrix there is only a low lever command, viz:

```
bmat = amat.mattranspose()
```

We could use some contrived symbol to designate the transposition, but that is not very useful, as an artificial symbol would not help us to remember the command.

In the following, when there are several commands that give the same result, we will mention which are the high and which are the low level. We will now describe all commands that Matalg module offers to the user.

# Commands

For brevity of writing the commands, import all functions from the module as follows:

```
from matalg27.Matalg27 import *
```

## Make a matrix:

Command:

```
amat = Matrix(m, n)
```

creates a (mxn) matrix with **m** rows, each row **n** entries long. All entries are filled with zeros. The dimensions m and n are stored within the matrix amat as amat.m (no of rows) and amat.n (length of rows). This instantiation creates a Matrix object, named amat. As a consequence object amat has access to all Matrix methods and other attributes - matrix dimensions and entries.

We already discussed another way of creating a matrix and at the same time fill it with entries:

```
amat = enterdata(m, n, [list of lists of entries])
```

## Output

The module provides several methods of outputting a matrix. Output can be obtained simply by using the print command:

```
print(amat)
```

This is a high level command. Let us look at an example:

```
>>> amat = Matrix(2, 3)
>>> print(amat)
[0.0, 0.0, 0.0]
[0.0, 0.0, 0.0]
```

This looks OK when all terms are about the same size, but let us look at an example when some entries are much longer that others. Consider the following example:

```
>>> amat = enterdata(2, 3, [[7, 2.0/3, 15], [4, 5, 6]], False)
>>> print(amat)
[7, 0.6666666666666666, 15]
[4, 5, 6]
>>>
```

Now that does not look so good, does it? Let us call a low lever command to help with a neater formatting. The command has an appropriate name, **neatprint**

```
>>> amat.neatprint()
A matrix of dimensions (m x n), where m, n, LineLen =  2 3 5
   7.00000E+00   6.66667E-01   1.50000E+01
   4.00000E+00   5.00000E+00   6.00000E+00
```

I think you will agree that this is a case where a lower level facility is better suited than the high level one! You will note that the neatprint is available to all instances of **Matrix** with the **dot expansion** feature of Python.

There is also the **printmat** command, which takes a message and a matrix as arguments:

```
>>> amat = Matrix(2, 3)
>>> printmat('printing again with a title message', amat)
printing again with a title message
A matrix of dimensions (m x n), where m, n, LineLen =  2 3 5
  [0.0, 0.0, 0.0]
  [0.0, 0.0, 0.0]
```

**printmat** is a function, rather than a method, so amat does not directly know about it. The function is part of the Matalg module. Because we have imported Matalg as _m, we have the command available as **_m.printmat.**

A special print function that is handy for gui development is **printline**. Printline simulates appending text to a plainText widget of PyQt:

```
line = string          printline(line)
```

There is another matrix print command that is obsolete:

```
amat.matprint()
```

## Matrix Addition

```
cmat = amat + bmat
```

The above is the high level version. The same result can be had with the following low level matrix addition command:

```
cmat = amat.matadd(bmat)
```

## Matrix Subtraction

```
cmat = amat - bmat
```

with its low level equivalent:

```
cmat = amat.matsub(bmat)
```

## Matrix Multiplication

```
cmat = amat * bmat
```

with its low level equivalent:

```
cmat = amat.matmult(bmat)
```

A reminder: amat * bmat is in general not the same as bmat * amat. What about multiplication of a matrix by a scalar? In high level there is no change in syntax:

```
cmat = scalar * amat
```

The low level version is:

```
cmat = amat.scalarmult(scalar)
```

Low level syntax in this case is not as clear, is it? Please notice that matrix multiplication by a scalar scales the matrix in-situ, so that in the two formulae above the **amat** matrix is scaled by the scalar factor.

So what happens if we now want to multiply two scalars? Well, there is no confusion - scalar multiplication will be performed correctly, because Matalg passes the operation to *standard* Python. On the other hand, if the two matrices are incompatible for multiplication, an exception will be raised.

## Solution of Simultaneous Equations

We already saw an example of solution of linear simultaneous equations in the introduction:

```
solution = amat ** rhs
```

Its low level equivalent is:

```
solution = amat.solve(rhs)
```

This is probably the most frequently used facility of the Matalg module. It can be used to determine the inverse on account of the following matrix equation:

```
solution = amat ** I
```

where amat is **amat** is **(nxn)** matrix, and I is a **(nxn)** unit matrix. It is trivial to prove that the **solution** of this system of equations is the inverse of **amat**.

## Inversion of a Matrix

```
inverse = ~ amat
```

Take care, **~** is a tilde. The low level equivalent is:

```
inverse = amat.matinvert()
```

**amat** is a square matrix and **amat** is its inverse. Caution - not all matrices are invertable and even if they are invertable, they can be badly conditioned and our methods may be inadequate and yield inaccurate results.

A small example:

```
>>> amat = enterdata(3, 3, [[3, 2, 1], [1, 4, 1], [1, 2, 6]])
('Echo check of enterdata', [[3.0, 2.0, 1.0], [1.0, 4.0, 1.0], [1.0, 2.0, 6.0]])
>>> cmat = ~ amat
>>> dmat = cmat * amat
>>> dmat.neatprint()
A matrix of dimensions (m x n), where m, n, LineLen =  3 3 5
  1.00000E+00   0.00000E+00   0.00000E+00
  0.00000E+00   1.00000E+00   2.77556E-17
  0.00000E+00   0.00000E+00   1.00000E+00
```

The calculated inverse of **amat** is **cmat** and their product must be a unit matrix. It is nearly a unit matrix, **2.77556E-17** being nearly zero. The deviation from zero is the effect of small roundoff errors.

## Transpose of a Matrix

```
transpose = amat.mattranspose()
```

where **transpose** is the **amat** transposed.

# Convenience Methods

These are methods that are probably not essential, such as a method for creation of a unit matrix of a given size.

## Convert a Matrix to a Unit Matrix

```
amat.matunit()
```

An example:

```
>>> bmat = Matrix(3,3)
>>> bmat.matunit()
[[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]]
>>> bmat.neatprint()
A matrix of dimensions (m x n), where m, n, LineLen =  3 3 5
  1.00000E+00   0.00000E+00   0.00000E+00
  0.00000E+00   1.00000E+00   0.00000E+00
  0.00000E+00   0.00000E+00   1.00000E+00
```

## Deep Copy of a Matrix

```
bmat = amat.matcopy()
```

Makes a deep copy of matrix **amat** and stores it in bmat.

## Matrix Equality

The equality of two floats is debatable. Equality of two matrices of floats is even more debatable, on account of the round off errors that often render the equality meaningless, or rather the inequality meaningless. To assure the equality we need to measure every term in each matrix. If the terms are results of calculation, the matrices will appear unequal, even if theoretically they should be equal. So perhaps the equality check should not be included in a matrix algebra package. For what it is worth, Matalg has an equality checking facility which returns **True** if all terms are equal, otherwise it returns **False.**

A numerical example:

```
>>> amat = enterdata(3, 3, [[3, 2, 1], [1, 4, 1], [1, 2, 6]], False)
>>> bmat = amat.matcopy()
>>> print(bmat == amat)
True
```

Enjoy Python!

Algis Kabaila, 2011-08-01