2DO:

- Follow Python programming standard, especially the use of spaces

1 Python and linear algebra

Many people use Matlab as their primary tool for linear algebra. We will here see that all things we can do with Matlab can also be done in a simple way with Python and Numpy. Python also addresses some things that may seen a bit strange in Matlam: As an example, everything is viewed as a matrix in Matlab, so that if we have a tuple of numbers, is is viewed as a two-dimensional object. In Python we distinguish more clearly between the two, in that a list can have either one or two dimensions. This prevents Python from misinterpreting a column vector for a row vector and vice versa, which easily occurs with Matlab.

1.1 Simple operations

If `A` is a matrix, `A.T` is its transpose. When we want to verify that a result is true, we often generate a matrix at random, and show that the result holds for this matrix. As an example, consider the following code.

```
A = matrix(random.rand(4,4))
B = matrix(random.rand(4,4))

tol = 1E-14
assert abs((A + B).T - A.T - B.T).max() < tol
assert abs((A * B).T - B.T * A.T).max() < tol
```

Which two known results are verified by this code?

The inverse of the matrix `A` can be computed with `linalg.inv(A)`.

The eigenvectors and eigenvalues of the matrix `A` can be computed by writing `D,P = linalg.eig(A)`. `A` must here be a square matrix. The return value `D` is an array which holds the eigenvalues of `A`, while the return value `P` is a matrix where the columns are the (corresponding) eigenvectors of `A` (the eigenvectors are normalized so that they have length 1, which can make them seem more complicated than necessary in some cases).

In `numpy` we can compute the scalar product and cross product of two vectors (arrays) `a` and `b` by writing `dot(a, b)` and `cross(a, b)`, respectively. The latter assumes that the vectors are three-dimensional. The (Euclidean) norm of a vector can be computed with `linalg.norm(a)`. The angle between `a` and `b` can now be computed by writing

```
arccos(dot(a,b)/(linalg.norm(a)*linalg.norm(b)))
```

If `c` is an an array or matrix, the sum and product of all its components can be found by writing ‘c.sum()‘ and `c.prod()`. These functions also take an optional parameters which represents the axis we should sum over: `c.sum(0)` returns a row vector where each element is the sum of the corresponding column, and `c.sum(1)` returns a column vector where each element is the sum of the corresponding row. `sum()` can be used to compute the mean value of a matrix, and what is called the Frobenius norm of a matrix, as follows:

```
m,n = shape(A)
A.sum()/(m*n)         # The mean value of A
sqrt(sum(A**2))       # The Frobenius norm of A
```

In the same way we can compute the maximum value in a matrix `c` by writing `c.max()`, and along different axes: `c.max(0)` returns a row vector where each element is the maximum value of the corresponding column, and `c.max(1)` returns a column vector where each element is the maximum value of the corresponding row. The minimum can be computed similarly.

1.2 Elementary row operations and Gaussian elimination

Let us see how we can bring a matrix to reduced row echelon form. We can swap rows ‘i‘ and `j` in a matrix `A` by writing `A[[i, j]] = A[[j, i]`. We can multiply row `i` with a constant `k` by writing `A[i] *= k`. We can add row `i` multiplied with `k` to row `j` by writing `A[j] += k*A[i]`.

The thress operations can be used to implement Gaussian elimination as follows.

```
m, n = shape(A)
for j in range(n - 1):
    for i in range(j + 1, m):
        A[i, j:] -= (A[i,j]/A[j,j])*A[j, j:]
```

In this code, we first eliminate the entries below the diagonal in the first column, by adding a scaled version of the first row to the other rows, Then the same procedure is applied for the second row, and so on. The result is an upper triangular matrix. The code can fail if some of the entries `A[j, j]` become zero along the way. To avoid this, we can swap rows if this is the case. The following code does this, and will not fail even if some of the columns are zero.

```
m, n = shape(A)
i = 0
for j in range(n):
    p = argmax(abs(A[i:m, j]))
    if p > 0: # Swap rows
        A[[i, p + i]] = A[[p + i, i]]
    if A[i, j] != 0:
        for r in range(i + 1, m):
            A[r, j:] -= (A[r, j]/A[i, j])*A[i, j:]
        i += 1
    if i>m:
        break
```

The rank of a matrix can be computed by first performing Gaussian elimination, and then count the number of pivot columns in the code above. A more reliable way to compute the rank is to compute the singular value decomposition of `A`, and check how many of the singular values which are larger than a threshold `epsilon`.

One can ask what other use we can have for performing row operations manually, besides that an implementation of Gaussian elimination uses them. Often one have matrices with a particular structure, where there for instance are many zeros (also called sparse matrices). When the location of the non-zeros then are known, it is more efficient to perform row operations for these locations manually, rather than perform the full Gaussian elimination.

If `A` is non-singular and quadratic, and `b` is a vector, then the equation `Ax=b` can be solved by writing `linalg.solve(A,b)`

The components in a matrix can be obtained with simple commands such as

```
A[0,1]    # The element in A at row 0, column 1
A[0,:]    # The first row in A
A[:,1]    # The second column in A
```

You can also use the colon-notation to pick out other parts of a matrix. If `C` is a 3×5-matrix,

```
C[1:3, 0:4]
```

gives a submatrix consisting of the two rows of `C` after the first, and the first four columns of `C`. The command

```
C[ ix_([0,2],[1,4])]
```

gives a matrix consisting of the first and third rows and second and fifth column of `A`. Note the specific function `ix\_` here to extract a certain set of rows/columns. You can also use this notation to permute the rows and columns in a matrix. If you write `C[[2,0,1]]`, the third row is placed first, follows by the first and second rows. If you write `C[:,[2,0,1,4,3]]`, the third column is placed first, followed by the first, second, fifth, and fourth, respectively.

Here are some other commands for producing new matrices:

```
A=ones((3,4))        # 3x4-matrix with only ones
A=eye(3)             # 3x3-matrix with ones on the main diagonal,
                     # zeros elsewhere
triu(A)              # Creates an upper triangular matrix from A, i.e.
```

```
                   # all elements under the main diagonal is set to 0
tril(A)            # Creates a lower triangular matrix from A, i.e.
                   # all elements over the main diagonal is set to 0
diag(c)            # A diagonal matrix with diagonal elements
                   # given by the vector c.
random.permutation(5) # Creates an arbitrary permutation of the
                      # first five numbers.
```

If `A` and `B` are matrices with equally many rows, the matrix where `A` and `B` are placed next to each other can be computed by writing

```
C=hstack([A,B])
```

The resulting matrix thus has the same number of rows as `A` and `B`. Similarly, if ‘A‘ and `B` have the same number of columns, the matrix where ‘A‘ and `B` are placed on top of oneanother can be computed by writing

```
C=vstack([A,B])
```

The resulting matrix has the same number of columns as `A` and `B`. While these two functions are specifically for matrices, one also has the function `concatenate`, which can be applied both for vectors and matrcies.

The operators +/- computes the componentwise sum/difference of vectors or matrices.

Multiplication and division have different meanings for `array` and ‘matrix‘ objects. For array objects, mutiplication and division are performed componentwise, i.e. if `C=A*B`, the components of the matrices are related by $c_{ij} = a_{ij} * b_{ij}$ (and similarly for division). This is also called the *Hadamard product* of `A` and `B`. For `matrix`-objects, mutiplication is performed as defined in linear algebra (section 5.7.5. Include definition). The following is a straightforward implementation of matrix multiplication

```
def mult(A,B):
    m, n = shape(A)
    n1, k =shape(B)
    assert n == n1, 'The dimensions of the matrices do not match!'
    C=zeros((m,k))
    for r in xrange(m):
        for s in xrange(k):
            for t in xrange(n):
                C[r,s] += A[r,t]*B[t,s]
    return C
```

If you compare this with running `A*B`, with `A` and ‘B‘ being matrix objects, you will notice a huge difference, in particular when the matrices are large. Clearly then, Python does not compute matrix multiplication with such an implementation.

Since multiplication has a different meaning for arrays and matrices, we may need to convert between the two. A matrix `A` can be converted to an array by writin `A = asarray(A)`.

The determinant (include definition, perhaps only in terms of expansion along rows/columns, and only say some things about its applications) is another concept which can be implemented in a smarter way. The following program computes the determinant in a straightforward way using its definition.

```
def detdef(A):
    assert A.shape[0] == A.shape[1], 'The matrix must be quadratic!'
    n = A.shape[0]
    if n == 2: # The determinant of a 2x2-matrix is computed directly
        return A[0,0]*A[1,1] - A[0,1]*A[1,0]
    else: # For larger matrix we expand the determinant along column 0
        determinant = 0.0
        for k in xrange(n): # Create sub-matrix by removing column 0, row k.
            submatrix = vstack((A[0:k,1:n],A[k+1:n,1:n]))
            # Multiply with alternating sign
            determinant += (-1)**k * A[k,0] * detdef(submatrix)
        return determinant
```

The `detdef` function is recursive, and calls itself until we have a matrix where the determinant can be computed directly. **hpl 1**: recursive functions are not part of INF1100 of the book... The central part in the code is where $(n-1)\times(n-1)$-submatrices are constructed. Note that in the code we check that the matrices are quadratic. This code is not particularly fast either. The determinant can also be computed with the built-in method `linlag.det(A)`, and this runs much faster, as the following code verifies.

```
from detdef import *
from numpy import *
import time

A=random.rand(9,9)
e0=time.time()
linalg.det(A)
print time.time()-e0
e0=time.time()
detdef(A)
print time.time()-e0
```

Here an arbitrary 9×9-matrix is constructed, and the determinant is computed and timed in the two different ways. The computation times is then written to the display. Run the code and see how much faster the built-in determinant function is! If you want, also try with an arbitrary 10×10-matrix, but then you should be patient while the code executes.

1.3 Plotting in three dimensions

The function `mesh(X, Y, Z)` plots a function in three dimensions. The function `meshc(X, Y, Z)` additionally draws contour curves. To plot the function $z = f(x, y) = xy\sin(xy)$ over the area $-4 \leq x \leq 4, -2 \leq y \leq 2$, we first create a partition of the `x`- and `y`-values (we here use step size 0.05). We then create a grid of points in the xy-plane using the function `meshgrid()`, and plot the values in the grid.

```
from math import *
from numpy import *
from scitools.easyviz import *

x=arange(-4,4,0.05,float)
y=arange(-2,2,0.05,float)
X, Y = meshgrid(x, y,sparse=False,indexing='ij')
Z = X*Y*sin(X*Y)
mesh(X, Y, Z)
```

If you choose `surf(X, Y, Z)` instead of `mesh(X, Y, Z)`, you get a graph where the surface elements has been coloured. You can rotate the figure by holding the mouse cursor down. If you just want to see the level curves, you can use the function `contour(X, Y, Z)`. This program takes an additional parameter which can be an array describing the level values, or a scalar describing the number of levels. This parameter is often necessary, since Python is not able always to determine the most interesting level curves. Also, we may be interested in very concrete levels only.

With the function `clabel` the values on the different levels are also displayed:

```
clabel(contour(x,y,z,12))
```

Normally the level curves are drawn with different colours.

This is very useful on the display, but is less practical if you want to paste the figure into a black and white document. If you write

```
contour(x,y,z,8,'k')
```

you will obtain 8 level curves drawn in black ('k' is the symbol for black).

The function `plot3` is useful for plotting parametrized curves in three dimensions. If you write

```
t=linspace(0,10*pi,100)
x=sin(t)
y=cos(t)
z=t
plot3(x,y,z)
```

the curve $\mathbf{r}(t) = (\sin t, \cos t, t)$ for $t \in [0, 10\pi]$ is drawn.

There is also support for drawing vector fields. If you write

```
quiver(x,y,u,v)
```

a vector field will be drawn where the `x`- og `y`-vectors specify point where vectors are to be drawn, and where the `u`- and `v`-vectors specify the vectors to be drawn at these points. This means that at every point (x, y), the vector (u, v) is drawn.

If you drop the x- og y parameters, the vectors will be drawn in a standard grid where all points have integer coordinates larger than zero.

When we plot a vector field it can be smart not to use too many points, since one risks that some vectors are drawn that the collide with oneanother, rendering a very cluttered plot.