# Lab7_sol

October 11, 2017

## 1 Sample solutions for exercises of Lab 7

### 1.1 Exercise 1

Write a function `myDerivative` that takes four arguments - a name of a function, the value of $x$, the value of $h$ and the order of the derivative (1 or 2) - and computes the value of the specified derivative at $x$ using the finite difference approximation given above. Using this function, verify the accuracy of the error estimates in the case of several concrete functions and several values of $x$: compute the derivatives for $h = 1, \frac{1}{2}, \frac{1}{4}, \ldots, \frac{1}{2^{10}}$ and find the quotient of the error to $h^2$. The quotients should approach a constant value.

**Solution**

```
In [1]: import numpy as np
        from scipy import linalg

        ##Exercise 1
        def myDerivative(f,x,h,order):
            if order==1:
                answer=(f(x+h)-f(x-h))/(2*h)
            elif order==2:
                answer=(f(x-h)-2*f(x)+f(x+h))/h**2
            else:
                answer="not implemented!"
            return answer
```

Let us check that the function works correctly:

```
In [2]: print(myDerivative(np.exp,0,0.1,1)-np.exp(0))
        print(myDerivative(np.exp,0,0.1,2)-np.exp(0))
```

```
0.00166750019844
0.000833611160723
```

Now check if the error of the approximate derivative computed by `myDerivative` behaves like $c \cdot h^2$ for some constant c for small enough $h$:

```
In [3]: h=1/2**np.arange(11)
        print((myDerivative(np.exp,1,h,1)-np.exp(1))/h**2)
        print((myDerivative(np.sin,1,h,1)-np.cos(1))/h**2)
        print((myDerivative(np.sin,1,h,2)-(-np.sin(1)))/h**2)
```

```
[ 0.47624622  0.45874388  0.45446485  0.45340105  0.45313547  0.45306909
  0.4530525   0.45304835  0.45304732  0.45304705  0.45304687]
[-0.08565359 -0.08893143 -0.0897694  -0.08998006 -0.0900328  -0.09004599
 -0.09004929 -0.09005011 -0.09005032 -0.09005037 -0.09005042]
[ 0.06782644  0.06954083  0.06997666  0.07008607  0.07011345  0.0701203
  0.07012201  0.07012243  0.07012274  0.07012377  0.07013741]
```

As we can see, the quotients tend to a constant value in all cases.

## 1.2 Exercise 2

Consider the following problem: find $y$ such that

$$y''(x) = f(x),\ x \in [a, b]$$

$$y(a) = 1,\ y(b) = -1$$

One possibility to solve a linear system of equations is to use the Python command `linalg.solve(M,z)` from the subpackage `linalg` of SciPy (that has to be imported first with `from scipy import linalg`), which returns the solution $y$ of the system of equations $My = z$. Use this command to find the values of the approximate solution of the problem with the finite difference method in the case $n = 10$, $a = 0$, $b = 1$, $f(x) = -24x^2$. Find the errors between the approximate solution and the exact solution $y(x) = 1 - 2x^4$.

**Solution**

```
In [4]: n=10
        a=0
        b=1
        def f(x):
            return -24*x**2
        h=(b-a)/n
        #define the right hand side of the system
        F=np.zeros(n+1)
        F[0]=1
        F[n]=-1
        x=np.linspace(a,b,n+1)
        i=np.arange(1,n)
        F[i]=f(x[i])
        #define M
        M=np.zeros(shape=(n+1,n+1))
        #first and last row separately
        M[0,0]=1
        M[n,n]=1
        #all other together
```

```
        i=np.arange(1,n) #numbers of other rows
        #diagonal elements
        M[i,i]=-2/h**2
        #before the diagonal
        M[i,i-1]=1/h**2
        #after the diagonal
        M[i,i+1]=1/h**2
        #solve the system
        y=linalg.solve(M,F)
        print("y values:",y)
        def exact_sol(x):
            return(1-2*x**4)
        print("errors:",np.abs(y-exact_sol(x)))
```

```
y values: [ 1.      0.998   0.9936  0.9796  0.944   0.87    0.736   0.5156  0.1776
 -0.314  -1.    ]
errors: [ 0.      0.0018  0.0032  0.0042  0.0048  0.005   0.0048  0.0042  0.0032
  0.0018  0.    ]
```

### 1.3 Exercise 3 * (for advanced students)

If the system matrix has only some nonzero diagonals then it is actually a waste of computer memory to store the full matrix. For solving such system it is actually possible to use the command `linalg.solve_banded((l,u),Dgs,z)`, where the rows of the matrix $Dgs$ contain the diagonals of $M$ starting from the highest one, $l$ is the number of diagonals below the main diagonal and $u$ is the number of diagonals above the main diagonal (in our case $l = u = 1$).NB! In the matrix Dgs for the diagonals that are above the main diagonal the first elements are actually not used (for the diagonal directly above the main diagonal the first element is not used, for the next diagonal two steps above the main diagonal the first two elements are not used etc); for diagonals below the main diagonal last elements are not used. Write a function that for a given $n$ solves the problem of the previous exercise with the command `linalg.solve_banded((l,u),Dgs,z)` and returns the maximal error at the points $x_i$, $i = 1, \ldots, n-1$. Determine how many times the error is reduced if we increase the number of points two times.

**Solution**

```
In [5]: def maxError(n):
            a=0
            b=1
            h=(b-a)/n
            F=np.zeros(n+1)
            F[0]=1
            F[n]=-1
            x=np.linspace(a,b,n+1)
            i=np.arange(1,n)
            F[i]=f(x[i])
            #define Dgs
            i=np.arange(1,n)
```

```
    DGS=np.zeros(shape=(3,n+1))
    #above the diagonal
    DGS[0,i+1]=1/h**2
    #Main diagonal
    DGS[1,0]=1 #thefirst element on the main diagonal is 1
    DGS[1,i]=-2/h**2
    DGS[1,n]=1 #the last element is 1
    #below the diagonal
    DGS[2,i-1]=1/h**2

    y=linalg.solve_banded((1,1),DGS,F)
    return np.max(abs(y-exact_sol(x)))
```

Let us see, how the error is changing when n is increasing:

```
In [6]: ratio="undefined"
        previous=0
        for n in 2**np.arange(1,11):
            error=maxError(n)
            if previous>0:
                ratio=previous/error
            print("n=",n,"error=",error,"ratio=",ratio)
            previous=error
```

```
n= 2 error= 0.125 ratio= undefined
n= 4 error= 0.03125 ratio= 4.0
n= 8 error= 0.0078125 ratio= 4.0
n= 16 error= 0.001953125 ratio= 4.0
n= 32 error= 0.00048828125 ratio= 4.0
n= 64 error= 0.0001220703125 ratio= 4.0
n= 128 error= 3.0517578125e-05 ratio= 4.0
n= 256 error= 7.62939453125e-06 ratio= 4.0
n= 512 error= 1.90734863281e-06 ratio= 4.0
n= 1024 error= 4.76837158203e-07 ratio= 4.0
```