GOMP0114 Inverse Problems in Imaging. Coursework 2 Hints

Introduction

This document is designed as support for Coursework 2, containing programming hints for Matlab and Python.

——————— *HINTS* ———————

1. Convolution and deconvolution

a.) **Matlab:** You can use the function `imread` to read an image from file. To convert the data type to double precision float, one may use the function `double`. One can display it with the function `imagesc` and a gray `colormap`.

Python: We require the library `numpy` and `matplotlib`:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
```

Use `mpimg.imread` to read an image from file. The function `np.float32` converts the data type to double precision. You can display the image with a gray colormap via

```
imgplot = plt.imshow(im,cmap='gray')
plt.colorbar()
plt.show()
```

b.) **Matlab:** The forward convolution can be done in several ways, but it's best if one writes a function with some parameters that one can input, such as:

$\texttt{BlurredIm} = \texttt{imblur}(\texttt{im}, \texttt{OtherParameters})$.

A possible way to construct `imblur` is first to generate a Gaussian kernel using `fspecial` then apply the kernel on the image using `imfilter`. We recommend an odd kernel size for the Gaussian kernel that ensures the transposed operator is the same as the original operator.

Note that above is only the convolution step. The noise should be added separately to make a particular data instantiation:

```
g = imblur(f,OtherParameters)
g = g + theta*randn(size(g));
```

Python: You can use the function `scipy.ndimage.filters.gaussian_filter` to perform Gaussian convolution with blur width σ and add noise as follow:

```
g = scipy.ndimage.filters.gaussian_filter(f,sigma)
w,h = g.shape
g = g + theta*numpy.random.randn(w,h)
```

c.) **Matlab:** Note that in our case `Af=imblur(f)` and since convolution is self-adjoint, i.e. $A^\top = A$, we have $\mathtt{A}^\top\mathtt{g=imblur(g)}$. Then the call looks like this (similarly for calling `gmres`)

$\mathtt{fa} = \mathtt{pcg}(@(\mathtt{x})\mathtt{ATA}(\mathtt{x}, \mathtt{alpha}), \mathtt{ATg})$.

When calling the functions `pcg`, `gmres` and even `lsqr`, one should pay attention to the dimension of the input variables. To be concrete, these solvers are solving a linear equation, say $Hx = y$, and expecting both x and y of vector form. As a result, one has to reshape the images from matrices to vectors before passing them to the solver. And one may also have to reshape the images from vectors to matrices before blurring them by functions.

Python: You can use `scipy.sparse.linalg.gmres` to call `gmres` in Python. The implementation of `gmres` requires function `scipy.sparse.linalg.LinearOperator`. You can first use `lambda` function to create the function handle for `ATA`, then define the Linear Operator as follow:

```
A = scipy.sparse.linalg.LinearOperator((M,N),ATA)
```

where `M` and `N` are the size of the M-by-N matrix of the linear system. Notice that `gmres` in Python requires the right hand side of the linear system to be in the vector form. Hence, you need to vectorize `ATg` before putting into `gmres`. You are able to call the `gmres` as follow:

```
gmresOutput = scipy.sparse.linalg.gmres(A, ATg)
```

d.) **Matlab:** You can define a function handle

$\mathtt{AaugHandle} = @(\mathtt{f}, \mathtt{transposeFlag})\mathtt{Aaug}(\mathtt{f}, \mathtt{A}, , \mathtt{alpha}, \mathtt{transposeFlag})$,

where `Aaug` implements the application of the matrix in the augmented equation above. You need to define a transpose and non-transpose. You can do this as follows:

```
function z=Aaug(f,A,alpha,transposeFlag)

switch transposeFlag
    case 'notransp'
        % implementation of the augmented matrix multiplication
    case 'transp'
        % implementation of the transposed augmented matrix multiplication
    otherwise
        error('input transposeFlag has to be ''transp'' or ''notransp''')
end

end
```

Python: You can use the function `scipy.sparse.linalg.lsqr` to implement `lsqr`. You need to define a transpose and non-transpose as in Matlab. You can do this as follows:

```
def M_f(f):
    % implementation of the augmented matrix multiplication
    return M_f
```

```
def MT_b(b):
    % implementation of the transposed augmented matrix multiplication
    return MT_b
```

The implementation of `lsqr` also requires function `scipy.sparse.linalg.LinearOperator`. You can define the linear operator as follow:

```
A = scipy.sparse.linalg.LinearOperator((M,N),matvec = M_f, rmatvec = MT_b)
```

where `M` and `N` are the size of the M-by-N matrix of the linear system. We need to concatenate the vectorized image `g` with a zero-vector which has the same size as `g` as the input `b` of `lsqr`, you can do this as follow:

```
import numpy as np
size = g.size
b = np.vstack([np.reshape(g,(size,1)),np.zeros((size,1))])
lsqrOutput = scripy.sparse.linalg.lsqr(M,b)
```

2. Choose a regularisation parameter α

a.) **Matlab:** You can use the function `fzero` to find the zero of the discrepancy function.

Python: You can use function `scipy.optimize.root` or `scipy.optimize.brentq` to find the zero of the discrepancy function.

b.) **Matlab:** You can use the `loglog` function to display the L-Curve.

Python: The function `matplotlib.pyplot.loglog` can be used for plotting the L-Curve.

3. Using a regularisation term based on the spatial derivative

a.) **Matlab:** See matlab functions `spdiags` for constructing the gradient operator using an explicit sparse matrix. Alternatively, you can use the function `diff` which is the most appropriate choice for the latter, but also multiplying by frequency in the Fourier domain is possible.

Python: You can construct the gradient operator via an explicit sparse matrix by using the `scipy.sparse.spdiags` function. Also you can use `numpy.diff`.

b.) **Matlab and Python:** You can check the correctness of constructing transpose operator $D^\top$ by simply examining the identity $< Du, v >=< u, D^\top v >$ for random u and v.

c.) **Matlab and Python:** You can use the methods suggested in Question 2 to choose an optimal value for α.

4. Construct an *anisotropic* derivative filter

- **Matlab:** Again you can use the function `spdiags` to construct the filter matrix γ.

  Python: You can use `scipy.sparse.spdiags` to construct γ.

5. Iterative deblurring

- **Matlab and Python:** In each iteration you can compute your minimiser with a solver from Exercise 1 (`pcg`, `gmres`, `lsqr`).