Numerical Scientific Computing Mini Project

Project Report

Group 18gr842

Aalborg University
Numerical Scientific Computing

AALBORG UNIVERSITY
STUDENT REPORT

Numerical Scientific Computing
Aalborg University
http://www.aau.dk

Title:
Numerical Scientific Computing Mini Project

Theme:
Numerical Scientific Computing

Project Period:
Spring Semester 2018

Project Group:
Group 18gr842

Participants:
Shagen Djanian
Niclas Hjorth Stjernholm
Marike Koch van den Broek

Supervisor:

Number of Pages: 20

Date of Completion:
May 9, 2018

Abstract:

The video example of the project is found on YouTube, following the link: https://youtu.be/79HsSZoeQIg.

The video example of the simulation is found on YouTube, following the link: https://youtu.be/1-KY_z2-Zf0.

Contents

Chapter 1

Introduction

The focus of this mini project is to implement and optimise a piece of python code. In this case the code snippet consists of two functions used in process of image processing of images of the iris. The functions are a noice remover function and a histogram equalisation function. In the following sections the functionalities of the functions are elaborated and the setup and approach for the optimisation is outlined.

1.1 The Functions

The images which are processed by the two functions are normalised images of the iris as can be seen in Figure 1.1. First the image is handled by the noise remover function and then the output is handled by the Equalise histogram function. The following sections describe the functionalities of the functions.

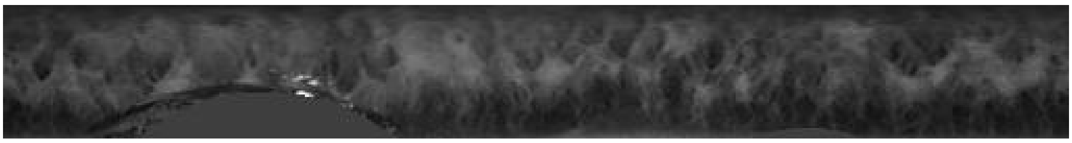

Figure 1.1: The normalised image of the iris before applying functions.

1.1.1 Noise Remover

The purpose of the noise remover function is to remove noise occurring in the form of eyelashes covering parts of the iris. Usually the pixels showing the lashes will be among the darkest pixels. Since every image of the iris is different and how dark the iris is also varies a threshold has to be identified adaptively. After the threshold has ben found it is applied to all pixels in the image. If the pixels are lower than the threshold the pixels have to be eliminated and reconstructed from neighbouring pixels.

1.1.2 Equalise Histogram

This function has to identify the span of the "main" pixel values in the histogram. Once the "main" part of the histogram has been identified this is stretched to cover the whole span of the range of the pixel values. This is done in order to increase the contrast in the image.

Chapter 2

Methods

This chapter describes the aim of the miniproject as well as the methods used

2.1 Approach

To implement the functionality from section 1.1 Python will be be used. The system was be designed according to the following plan:

1. The system specifications will be specified
2. The code will be compartmentalised into modules
3. During implementation unit testing will be used
4. Once it passes the unit testing it will be optimised

The code will be differentiated into two parts; the naïve and the optimised. The naïve implementation will be written using only functionality available in Python. The optimised version will contain the same functionality as the naïve but will be implemented using optimised libraries like numpy.

To track the progress git will be used for version control and feature/bug tracking. Comments in the code will be used as documentation. Once the system works and had been optimised, it will be rewritten for parallel processing to see if it can be optimised further.

2.2 Test setup

All tests were conducted on the same laptop in order to be as consistent as possible with the conditions under which the testing was conducted. The laptop used was a Macbook Pro from early 2013, running High Sierra. The Macbook has 8 GB ram and has a 2.6 GHz Intel Core i5 processor, which is a dual core processor. The scripts were run using the Python version 3.6.3.

Chapter 3

Implementations

This chapter describes the process of implementing the three different versions of the functions.

3.1 Naïve implementation

The naïve implementation was first implemented. This was done using only standard python operations, however, Numpy arrays were used as well as the numpy.zeros and numpy.empty for creating the arrays.

In the naïve version of the code a function is needed which creates histogram information based on an image and parameters given as arguments.

The final code of the naïve implementation can be seen in Appendix B

3.2 Optimised Implementation

In order to optimise the naïve implementation the execution time for each line was investigated using line_profiler. The result of running the line_profiler on the naïve implementation can be seen in Appendix C. As can be seen in Appendix C the line that takes the absolute longest to execute per hit are the lines which calls the function which creates the histograms. Therefore, this was first sought to be improved. This was done by utilising the Numpy function for generating histograms.

3.3 Multiprocessing

The whole flow of the image processing is sequential and it doesn't make sense to parallelise as the processes are depended on each other. But since the images are independent from of each other, it is Single Program Multiple Data (SPMD) situation. In other words it does not what order the images are processed and the same image processing is applied to all images. There is potential for improvement in performance by utilising more than one CPU core for processing. This is implemented by using the Python package Multiprocessing. Here a pool of workers is created that can work on the data. When a worker has finished it's task it takes the next task in line, and so on. The tasks are scheduled automatically by the package. The function map_async() is used as the work does not need to be done synchronously. map_async().get is also used to make the program wait until the processing is done and return everything at once. The data is then saved into a pandas dataframe for later usage.

Chapter 4

Testing

This section describes the more structured testing conducted during this mini project. The testing work is split into two parts. The first part contains descriptions of the unit testing conducted during development, while the second part describes the benchmarking of the implemented code.

4.1 Unit Teasting

Unit testing was used to ensure the coded worked as intended during development. The concept of unit testing is to build tests cases for the smallest part of the code that can be run to check if the code produces the expected results. Each test case is independent from each other. When the unit test is run a log is generated with error messages, if any were found. The unit test was run before working on the code and after finishing work. For this the framework unittest in python was used. Here a separate module is created with a class that contains the test cases. Two separate cases, setUp and tearDown, are also created to be run before and after every test. This ensures that the data being used in the tests is not changed from test to test. Three test cases were made:

- Comparing the panda dataframe produced by the parallel and the sequential implementation
- Comparing the histogram equalisation from the naive implementation and optimised implementation
- Comparing the noise removal and reconstruction from the naive implementation and optimised implementation

In all three Numpy was used to compare the values with Numpy.isclose. Comparing floats directly using the == operator can return false because of floating point errors. Since the things being compared were images which are stored as matrices, Numpy.isclose returns a boolean matrix. Numpy.all is then used to check if all the entries are true. If they are all true the test case is accepted.

4.2 Benchmarking

In order to benchmark the implemented code some tests were conducted. The three different versions of the code were compared in regard to execution time. During development line_profiler was used for identifying the most time demanding lines in order to know what lines to improve. However, this section describes the comparison of the final versions of the code. For the testing described in this section a script that contains the database of iris images and necessary functions was created. This script can be seen in **??**.
A comparison of the naïve implementation and the optimised implementation was done comparing the run times of the processing of 10 images. Ideally a comparison of the processing of the entire available database would be conducted. However, during development it was observed that the processing of a single image by the naïve implementation takes more than a minute, and since there are more than 1200 images in the database it would take more than 20 hours to let the naïve implementation process the whole database. Therefore, the decision was made to only run on a limited set of images and compare these times.
The times for this test were both measured using line_profiler as well as using the timeit module. The precision of the two method differ and the obtained result also in some case differs a little as can be seen in Table **??**.

Implementation	line_profiler	timeit
Naïve	1145.8 s	1145.8354333179996 s
Optimised	0.871651 s	0.8716400859993882 s

Table 4.1: Execution times of the processing of 10 iris images.

Furthermore, it should be noted what a large difference in execution time there is between the runs of the two implementations. The second, optimised implementation, which utilises Numpy functionalities, is more than 1,300 times faster.
For the comparison between the optimised code running normally in a sequential way or running the processing of images in parallel using multi processing the code was run on the entire available database. Since the line_profiler was not working for the multiprocessing the scripts were only timed using the timeit module.

Implementation	timeit
Sequential	108.69415000799927 s
Multi Processing	56.29580797100061 s

Table 4.2: Execution times of the processing of the entire database of iris images.

As can be seen in Table **??** the execution time is decreased significantly by utilising multiprocessing. This shows how beneficial multiprocessing can be. For this, rather small, example of processing of a database the time used for implementing the multiprocessing might not worth the time saved in the execution, however, one can imagine that a decrease in execution time by almost half of the time for processing a very large database could be a huge gain.

Appendix A

Appendix A

The script used for running the tests across several images.

```
# -*- coding: utf-8 -*-
"""
Created on Tue May  1 13:24:40 2018

@author: Shaggy
"""
import load_images_2_python
import NSC_miniproject_2 as noiserm
import pywt
import cv2
import matplotlib.pyplot as plt
import timeit
import multiprocessing as mp
import os

#%%
#@profile
def iris_proc (image):
    img_without_noise = noiserm.noiseremover(image,0.1,10)
  #  print("noise removed")
    equalised_img = noiserm.equalisehistogram(img_without_noise,10)
  #  print("histogram equalised")
    wavelet_results= pywt.wavedec2(equalised_img,'haar',level=3)
    featureVec = wavelet_results[0].reshape(wavelet_results[0].size)
    pid = os.getpid ()
    print("Feature Extracted by: process  id:   {:7d}".format(pid))
    return featureVec

#%%
featureVector = []
dataFrame = load_images_2_python.dataFrame

#dataFrame = dataFrame.iloc[range(224),:]
#dataFrame = dataFrame.drop(dataFrame.index[224]) 3-polar.jpg 0005left
discardList = ['0005left_3-polar.jpg','0014right_2-polar.jpg'] #
dataFrame = dataFrame[~dataFrame['full_path'].isin(discardList)]
```

```python
print("The following images have been dropped because the iris
    localisation was not good enough: ", discardList)

#%%
#@profile
def sequential():

    start_time = timeit.default_timer()
    i = 1
    for image in dataFrame.image:

        featureVec = iris_proc(image)
        print("feature extracted")
        featureVector.append(featureVec)

        #print(len(featureVector))
        print("Image ",i, "out of ",dataFrame.image.size,"done")
        i = i + 1

    print ("FINISHED Sequencial")
    print(timeit.default_timer() - start_time) # 250.1482454747301
    dataFrame['featureVector'] = featureVector
    dataFrame.to_pickle('pythonDatabase_seq')

#@profile
def parallel():

    __spec__ = "ModuleSpec(name='builtins', loader=<class
        '_frozen_importlib.BuiltinImporter'>)"
    n_cores = mp.cpu_count()
    pool = mp.Pool(processes=n_cores//2)
    start_time = timeit.default_timer()
    featureVector = pool.map_async(iris_proc,dataFrame.image).get()
    pool.close()
    pool.join()
    print("FINISHED parallel")
    print(timeit.default_timer() - start_time) #94.28003701802831
    dataFrame['featureVector'] = featureVector
    dataFrame.to_pickle('pythonDatabase_para')


if __name__ == '__main__':
    parallel()
    #sequential()
    pass
```

Appendix B

Appendix B

The code of the final naïve implementation.

```
import numpy as np
import matplotlib.pyplot as plt
import copy


def customhist(image,numberofbins,ran):
    increment=ran[1]/numberofbins
    binsE=np.zeros(numberofbins+1)
    hist=np.zeros(numberofbins)
    for i in range(0,numberofbins+1):
         binsE[i]=i*increment
    imdim=image.shape
    for ii in range(0,imdim[0]):
        for iii in range(0,imdim[1]):
            for pp in range(0,numberofbins):
                if image[ii][iii]>=binsE[pp] and image[ii][iii]<binsE[pp+1]:
                    hist[pp]=hist[pp]+1
    return hist, binsE

#@profile
def equalisehistogram(reconstructIris,LimitValue):

    numberOfBins=256
    histE, bin_edgesE=customhist(reconstructIris,numberOfBins,(0,256))
    imageDim=reconstructIris.shape
    Equalised=np.zeros(imageDim)
    lowVal = 255.0
    higVal = 0.0

    for i in range(0,numberOfBins):
        if histE[i]>LimitValue:
            if bin_edgesE[i]<lowVal:
                lowVal=bin_edgesE[i]
            if bin_edgesE[i]>higVal:
                higVal=bin_edgesE[i]
```

```

    for p in range(0,imageDim[0]):
         for c in range(0,imageDim[1]):
              temp=(reconstructIris[p][c]-lowVal)*(255/(higVal-lowVal))
              Equalised[p][c]=temp
              if temp<0:
                  Equalised[p][c]=0
              if temp>255:
                  Equalised[p][c]=255
    return Equalised







#@profile
def noiseremover(sourceimage,HistoFrac,RecognitionValue):

    numberOfBins=256
    hist, bin_edges=customhist(sourceimage,numberOfBins,(0,256))
    #plt.hist(sourceimage.ravel(),256)
    #plt.show()

    lowVal = 255.0
    higVal = 0.0

    for i in range(0,numberOfBins):
        if hist[i]>RecognitionValue:
            if bin_edges[i]<lowVal:
                lowVal=bin_edges[i]
            if bin_edges[i]>higVal:
                higVal=bin_edges[i]

    ThresVal=lowVal+HistoFrac*(higVal-lowVal);
    reconstructIris=copy.deepcopy(sourceimage)
    imageDim=sourceimage.shape
    ref=np.empty(imageDim, dtype=bool)
    xCord=[]
    yCord=[]
    for h in range(0,imageDim[0]):
        for s in range(0,imageDim[1]):
            if sourceimage[h][s]<=ThresVal:
                ref[h][s]=True
                xCord.append(h)
                yCord.append(s)
            else:
                ref[h][s]=False
    processMap=copy.deepcopy(ref)
    NumberofEliminations=len(xCord)
    numberofUneliminatedNeighbors=0
    pixelVal=0
```

```
    SumVal=0
    UnprocessedPixels=NumberofEliminations


    while UnprocessedPixels>0:
        for ii in range(0,NumberofEliminations):
            if processMap[xCord[ii]][yCord[ii]]==True:#if the current pixel still has not been reconstructed then do reconstruction
                if xCord[ii]-1>=0:#make sure the neighbor is within the image boundary
                    if processMap[xCord[ii]-1][yCord[ii]] == False and sourceimage[xCord[ii]-1][yCord[ii]] is not None: #make sure the neighbor pixel is not none and does not need reconstruction.
                        SumVal=SumVal+reconstructIris[xCord[ii]-1][yCord[ii]] #contribute to current pixel reconstruction
                        numberofUneliminatedNeighbors = numberofUneliminatedNeighbors+1
                if xCord[ii]+1<imageDim[0]:#make sure the neighbor is within the image boundary
                    if processMap[xCord[ii]+1][yCord[ii]] == False and sourceimage[xCord[ii]+1][yCord[ii]] is not None: #make sure the neighbor pixel is not none and does not need reconstruction.
                        SumVal=SumVal+reconstructIris[xCord[ii]+1][yCord[ii]]
                        numberofUneliminatedNeighbors = numberofUneliminatedNeighbors+1
                if yCord[ii]-1>=0: #make sure the neighbor is within the image boundary
                    if processMap[xCord[ii]][yCord[ii]-1] == False and sourceimage[xCord[ii]][yCord[ii]-1] is not None: #make sure the neighbor pixel is not none and does not need reconstruction.
                        SumVal=SumVal+reconstructIris[xCord[ii]][yCord[ii]-1]
                        numberofUneliminatedNeighbors = numberofUneliminatedNeighbors+1
                if yCord[ii]+1<imageDim[1]:#make sure the neighbor is within the image boundary
                    if processMap[xCord[ii]][yCord[ii]+1] == False and sourceimage[xCord[ii]][yCord[ii]+1] is not None: #make sure the neighbor pixel is not none and does not need reconstruction.
                        SumVal=SumVal+reconstructIris[xCord[ii]][yCord[ii]+1]
                        numberofUneliminatedNeighbors = numberofUneliminatedNeighbors+1
                #the numbers in the if statement below represents the number of included neighbors
                if numberofUneliminatedNeighbors==4 or numberofUneliminatedNeighbors==3 or numberofUneliminatedNeighbors==2:
```

```
                    pixelVal=SumVal/numberofUneliminatedNeighbors #calculate pixel value based on average of existing neighbor pixels
                    reconstructIris[xCord[ii]][yCord[ii]]=pixelVal
                    processMap[xCord[ii]][yCord[ii]]=False
                    UnprocessedPixels=UnprocessedPixels-1 #decrease the counter of pixels still to be processed
                SumVal=0
                numberofUneliminatedNeighbors=0
    return reconstructIris
```

Appendix C

Appendix C

```
import numpy as np
import matplotlib.pyplot as plt
import copy


def customhist(image,numberofbins,ran):
    increment=ran[1]/numberofbins
    binsE=np.zeros(numberofbins+1)
    hist=np.zeros(numberofbins)
    for i in range(0,numberofbins+1):
         binsE[i]=i*increment
    imdim=image.shape
    for ii in range(0,imdim[0]):
        for iii in range(0,imdim[1]):
            for pp in range(0,numberofbins):
                if image[ii][iii]>=binsE[pp] and
                    image[ii][iii]<binsE[pp+1]:
                    hist[pp]=hist[pp]+1
    return hist, binsE

#@profile
def equalisehistogram(reconstructIris,LimitValue):

    numberOfBins=256
    histE, bin_edgesE=customhist(reconstructIris,numberOfBins,(0,256))
    imageDim=reconstructIris.shape
    Equalised=np.zeros(imageDim)
    lowVal = 255.0
    higVal = 0.0

    for i in range(0,numberOfBins):
        if histE[i]>LimitValue:
            if bin_edgesE[i]<lowVal:
                lowVal=bin_edgesE[i]
            if bin_edgesE[i]>higVal:
                higVal=bin_edgesE[i]

```

```
    for p in range(0,imageDim[0]):
         for c in range(0,imageDim[1]):
              temp=(reconstructIris[p][c]-lowVal)*(255/(higVal-lowVal))
              Equalised[p][c]=temp
              if temp<0:
                  Equalised[p][c]=0
              if temp>255:
                  Equalised[p][c]=255
    return Equalised







#@profile
def noiseremover(sourceimage,HistoFrac,RecognitionValue):

    numberOfBins=256
    hist, bin_edges=customhist(sourceimage,numberOfBins,(0,256))
    #plt.hist(sourceimage.ravel(),256)
    #plt.show()

    lowVal = 255.0
    higVal = 0.0

    for i in range(0,numberOfBins):
        if hist[i]>RecognitionValue:
            if bin_edges[i]<lowVal:
                lowVal=bin_edges[i]
            if bin_edges[i]>higVal:
                higVal=bin_edges[i]

    ThresVal=lowVal+HistoFrac*(higVal-lowVal);
    reconstructIris=copy.deepcopy(sourceimage)
    imageDim=sourceimage.shape
    ref=np.empty(imageDim, dtype=bool)
    xCord=[]
    yCord=[]
    for h in range(0,imageDim[0]):
        for s in range(0,imageDim[1]):
            if sourceimage[h][s]<=ThresVal:
                ref[h][s]=True
                xCord.append(h)
                yCord.append(s)
            else:
                ref[h][s]=False
    processMap=copy.deepcopy(ref)
    NumberofEliminations=len(xCord)
    numberofUneliminatedNeighbors=0
    pixelVal=0
    SumVal=0
```

```
    UnprocessedPixels=NumberofEliminations


    while UnprocessedPixels>0:
        for ii in range(0,NumberofEliminations):
            if processMap[xCord[ii]][yCord[ii]]==True:#if the current pixel still has not been reconstructed then do reconstruction
                if xCord[ii]-1>=0:#make sure the neighbor is within the image boundary
                    if processMap[xCord[ii]-1][yCord[ii]] == False and sourceimage[xCord[ii]-1][yCord[ii]] is not None: #make sure the neighbor pixel is not none and does not need reconstruction.
                        SumVal=SumVal+reconstructIris[xCord[ii]-1][yCord[ii]] #contribute to current pixel reconstruction
                        numberofUneliminatedNeighbors = numberofUneliminatedNeighbors+1
                if xCord[ii]+1<imageDim[0]:#make sure the neighbor is within the image boundary
                    if processMap[xCord[ii]+1][yCord[ii]] == False and sourceimage[xCord[ii]+1][yCord[ii]] is not None: #make sure the neighbor pixel is not none and does not need reconstruction.
                        SumVal=SumVal+reconstructIris[xCord[ii]+1][yCord[ii]]
                        numberofUneliminatedNeighbors = numberofUneliminatedNeighbors+1
                if yCord[ii]-1>=0: #make sure the neighbor is within the image boundary
                    if processMap[xCord[ii]][yCord[ii]-1] == False and sourceimage[xCord[ii]][yCord[ii]-1] is not None: #make sure the neighbor pixel is not none and does not need reconstruction.
                        SumVal=SumVal+reconstructIris[xCord[ii]][yCord[ii]-1]
                        numberofUneliminatedNeighbors = numberofUneliminatedNeighbors+1
                if yCord[ii]+1<imageDim[1]:#make sure the neighbor is within the image boundary
                    if processMap[xCord[ii]][yCord[ii]+1] == False and sourceimage[xCord[ii]][yCord[ii]+1] is not None: #make sure the neighbor pixel is not none and does not need reconstruction.
                        SumVal=SumVal+reconstructIris[xCord[ii]][yCord[ii]+1]
                        numberofUneliminatedNeighbors = numberofUneliminatedNeighbors+1
                #the numbers in the if statement below represents the number of included neighbors
                if numberofUneliminatedNeighbors==4 or numberofUneliminatedNeighbors==3 or numberofUneliminatedNeighbors==2:
                    pixelVal=SumVal/numberofUneliminatedNeighbors #calculate pixel value based on average of existing neighbor pixels
```

```
                    reconstructIris[xCord[ii]][yCord[ii]]=pixelVal
                    processMap[xCord[ii]][yCord[ii]]=False
                    UnprocessedPixels=UnprocessedPixels-1 #decrease
                        the counter of pixels still to be processed
                SumVal=0
                numberofUneliminatedNeighbors=0
    return reconstructIris
```