

**KAUNAS UNIVERSITY OF TECHNOLOGY**

**Faculty of Informatics**

**Department of Information Systems**

**Image Processing – Application of data Concurrency Tools**

December 02, 2021

Student: Leo Onyema Sochi Muoghara

Teacher: Karolis Ryselis

Kaunas, 2021

### Table of Contents

[1. Task analysis and solution 3](#_Toc89017156)

[1.1. Problem Description 3](#_Toc89017157)

[1.2. Problem Solution 3](#_Toc89017158)

[1.3. Tool Implimentation 3](#_Toc89017159)

[2. Testing and instructions 6](#_Toc89017160)

[2.1. Program Results 7](#_Toc89017161)

[2.2. Program Run Instruction 7](#_Toc89017162)

[3. Performance analysis 7](#_Toc89017163)

[Conclusions 7](#_Toc89017164)

[References 7](#_Toc89017165)

[Appendices 9](#_Toc89017166)

[Project Initialization and functions 9](#_Toc89017167)

[Display the Function 9](#_Toc89017168)

[Scanning Method of 𝑓(𝑥) and 𝑔(𝑥) 9](#_Toc89017169)

[Bisection Method 11](#_Toc89017170)

[Chord Method 11](#_Toc89017171)

[Newton Method 12](#_Toc89017172)

[Chord Method of 𝑓(𝑥) and 𝑔(𝑥) 12](#_Toc89017173)

[Newton’s Method of 𝑓(𝑥) and 𝑔(𝑥) 13](#_Toc89017174)

[Tasks numbers: 11-15 14](#_Toc89017175)

# Task analysis and solution

## Problem Description

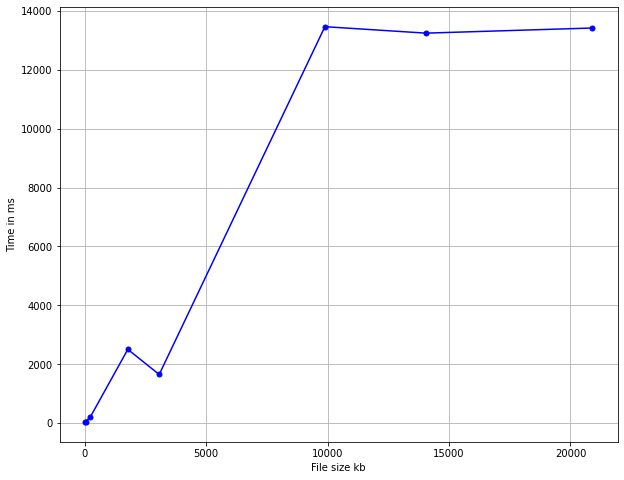
A control test was done to transform 8 digital images by applying the Gaussian Blur. The control test was performed in a single thread on the CPU with no concurrent tool running.

The following results were obtained.

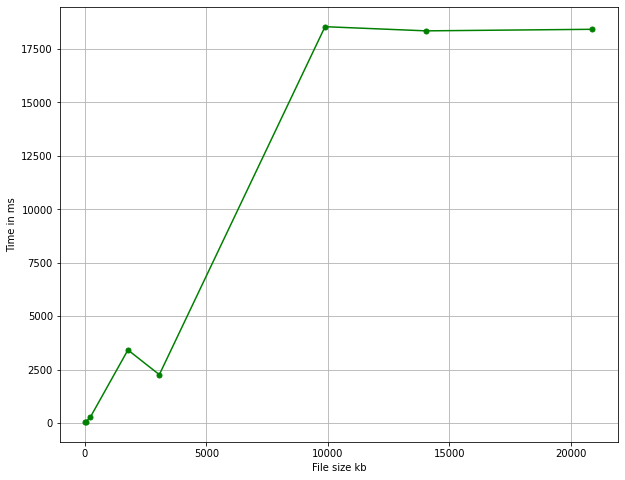
Chart, line chart

Description automatically generated

**Fig 1 Time Taken in ms to Apply Gaussian Blur of Kernel size 3x3 Images of different file Sizes in kb**



**Fig 2 Time Taken in ms to Apply Gausssian Blur of Kernel size 5x5 Images of different file Sizes in kb**



**Fig 3 Time Taken in ms to Apply Gausssian Blur of Kernel size 7x7 Images of different file Sizes in kb**

**Table 1 Result of a Single Thread applying Gaussian Blur of Varied sizes**

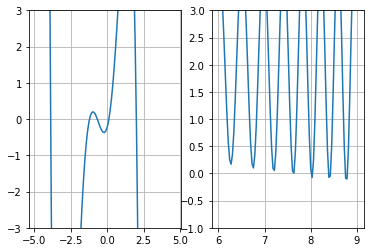
|  |  |  |  |
| --- | --- | --- | --- |
| File Size kb | Kernel 3x3 (ms) | 5x5 (ms) | 7x7 (ms) |
| 10 | 25 | 35 | 50 |
| 30 | 33 | 46 | 62 |
| 215 | 137 | 189 | 263 |
| 1,770 | 1775 | 2503 | 3422 |
| 3,063 | 1175 | 1650 | 2266 |
| 9,887 | 9625 | 13462 | 18544 |
| 14,045 | 9448 | 13243 | 18349 |
| 20,900 | 9631 | 13417 | 18422 |

We can see that the trend in all three graphs is somewhat consistent. In the **Table.1**, we see that the time taken increases as the kernel size increases because the number of operation increases.

The goal of this project is to utilize CUDA (c++) concurrent tool to reduce the time required to perform these transformations.

## Problem Solution

The second step is to display functions 𝑓(𝑥) and 𝑔(𝑥) so that the roots of the functions can be seen graphically. The code used can be seen **in Appendices.**



**Figure.1 Left Graph is the function 𝑓(𝑥) and Right graph is the function 𝑔(𝑥)**

## Tool Implimentation

The third step is to find all root isolation intervals of the functions 𝑓(𝑥) and 𝑔(𝑥) using the scanning method with a constant size step (chosen step is **0.1**). For 𝑓(𝑥), then select the interval was obtained in **section 1.1**, for 𝑔(𝑥) the interval provided in the **Task**was used. All the results were displayed graphically and by text seen in **Figure.2, Figure.3, Table.1, and Table.2**.

Chart, line chart

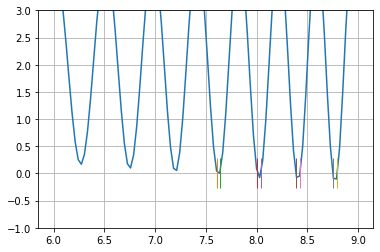
Description automatically generated

**Figure.2 Graph of the function 𝑓(𝑥) with root indicators**

The coloured circles show all the roots.

**Table.1 Results obtained for 𝑓(𝑥) from the Scanning Method**

|  |
| --- |
| Root of f(x) = -0.00000000002832870050  Root = -3.88395555239622192190  xFrom = -3.88395555239840462036; xTo = -3.88395555239403922343  Root of f(x) = 0.00000000003631026035  Root = -1.21661564647220155067  xFrom = -1.21661564655369236476; xTo = -1.21661564639071095861  Root of f(x) = 0.00000000000714270310  Root = -0.68074114703340526944  xFrom = -0.68074114719638667559; xTo = -0.68074114687042375227  Root of f(x)= -0.00000000002789332654  Root = 0.11394900159211759916  xFrom = 0.11394900138257002542; xTo = 0.11394900180166517289  Root of f(x)= -0.00000000007237746513  Root = 2.00069667764764735907  xFrom = 2.00069667763309544384; xTo = 2.00069667766219927429  Total iteration: = 170 |



**Figure.3 Graph of the function 𝑔(𝑥) with root indicators**

The coloured lines show all the roots.

**Table.2 Results obtained for 𝑔(𝑥) from the Scanning Method**

|  |
| --- |
| Root of f(x)= 0.00000000009679101964  Root = 7.61235680685316928873  xFrom= 7.61235680675829140540; xTo= 7.61235680694804806024  Root of f(x)= 0.00000000002802469368  Root = 7.63540009117798046390  xFrom= 7.63540009114072759644; xTo= 7.63540009121523333135  Root of f(x)= -0.00000000007545919445  Root = 8.00772865871196160015  xFrom= 8.00772865866655791933; xTo= 8.00772865875736350461  Root of f(x)= -0.00000000007087175291  Root = 8.04293704922422847403  xFrom= 8.04293704903796502492; xTo= 8.04293704941049369950  Root of f(x)= 0.00000000001057376409  Root = 8.38683266674163263588  xFrom= 8.38683266673115568324; xTo= 8.38683266675210958851  Root of f(x)= 0.00000000001029265562  Root = 8.42846829760930660314  xFrom= 8.42846829755342774604; xTo= 8.42846829766518723659  Root of f(x)= 0.00000000002814681821  Root = 8.75051605630718043471  xFrom= 8.75051605629728612712; xTo= 8.75051605631707651867  Root of f(x)= 0.00000000001843325492  Root = 8.79614506732782786003  xFrom= 8.79614506729988931966; xTo= 8.79614506735576817675  Total iteration: = 248 |

xFrom and xTo are the root isolation interval.

**Scanning Method**

This method works by first selecting the initial guess or isolation interval of the root, i.e., 0.1. Risk of missing roots because the wrong step of discretization was chosen. At the end of each isolation interval, the function value has opposite values. This is known as the ***sign rule***. After, one must narrow down the root value range continuously while the **acceptance tolerance (e.g., 10-10)**of the root value isn’t exceeded. Once the root and its’ intervals are found, one must start from the upper limit of the root isolation interval which was found and use the initial isolation interval and repeat the process. The code used can be found in the **Appendices**.

**Bisection Method**

This method was used concurrently with the scanning method to obtain the roots and its’ interval. When using the bisection method, we must first choose the range of the interval, xn and xn1. By applying the formula below, we obtain the mid of the interval.

Once xmid is obtained, we check if the sign of **𝑓(𝑥mid) is equal to 𝑓(𝑥n).** If they are equal, we assign the value of xmid to xn thereby narrowing the interval; if they are not equal then xmid is assigned to xn1. This process is repeated while the acceptance tolerance isn’t exceeded. The code used can be found in the **Appendices**.

# Testing and instructions

**Task**

Body of temperature 𝑇0 is brought to the environment of temperature 𝑇𝐴. Body temperature does not impact the temperature of the environment. Body temperature 𝑇(𝑡) depends on environment temperature as

𝑇(𝑡) = (𝑇0 − 𝑇𝐴)𝑒𝑘𝑡 + 𝑇𝐴

Here 𝑘 – proportional coefficient, 𝑡 – time. What is the proportionality coefficient k, if it is known that after time period 𝑡1 body temperature is 𝑇1? Where:

**𝑇0, K** = 473; **𝑇𝐴, K** = 373; **𝑡1, s** = 20; **𝑇1, K** = 378

Inputting the values into the formula, we get:

378 = (473 − 373)𝑒𝑘20 + 373

Equating the equation to zero, we get:

0 = (473 − 373)𝑒𝑘20 + 373 - 378

This task was solved numerically using the Scanning and Bisection method, and the results are shown below.

Chart, line chart

Description automatically generated

**Figure.10 This image shows the calculated results of 𝑔(𝑥) obtained from WoldramAlpha**

**Table.10 Results obtained for 𝑔(𝑥) from the Newton’s Method**

|  |
| --- |
| Root of T(t) = 0.00000000002467004379  k = -0.14978661367745288446  xFrom= -0.14978661368985113356; xTo= -0.14978661366505466312  Total iteration: = 35 |

The code used can be found in the **Appendices**.

## Program Results

## Program Run Instruction

# Performance analysis

# Conclusions

This task was solved successfully and can be backed by the results obtain in comparison to the results gotten from WolframAlpha. Some discrepancies can be seen In the result’s level of accuracy across the different methods used which may be due to the acceptance tolerance, as well as the methodology used to find the results.

We can note that:

* Newton’s Method is the fastest at **33** iterations for f(x) and **35** iterations for g(x),
* followed by Bisection method at **170** iterations for f(x) and **248** iterations for g(x),
* and then Chord Method at **294** iterations for f(x) and **285** iterations for g(x).

# References

* Task of Project I, P170B115 Numerical Methods and Algorithms (2021). *KTU Moodle*. Available at *https://moodle.ktu.edu/* (Accessed: 2 September 2021).
* Solution of a single linear equation, NUMERICAL METHODS AND ALGORITHMS (NMA) P170B115. *KTU Moodle*. Available at *https://moodle.ktu.edu/* (Accessed: 2 September 2021).

# Appendices

## Project Initialization and functions

|  |
| --- |
| import numpy as np;  import matplotlib.pyplot as plt  # given function fx  def fx(x):    return -0.3\*x\*\*5 - 1.10\*x\*\*4 + 1.14\*x\*\*3 + 3.84\*x\*\*2 + 1.48\*x - 0.22  # given function gx  def gx(x):    return 2-np.log(x)\*np.sin(x\*\*2) |

## Display the Function

|  |
| --- |
| # Question 1.2  x1 = np.linspace(-4.9, 4.6, 100)  y1 = fx(x1)  x2 = np.linspace(6, 9, 100)  y2 = gx(x2)  #plot for fx  plt.subplot(1, 2, 1)  plt.plot(x1,y1)  plt.ylim(-3,3)  plt.grid(b=True);  #plot for gx  plt.subplot(1, 2, 2)  plt.plot(x2,y2)  plt.ylim(-1,3)  plt.grid(b=True); |

## Scanning Method of 𝑓(𝑥) and 𝑔(𝑥)

|  |
| --- |
| # Question 1.3a for fx  # Rneg and Rpos are the intial range of the scan  # h is the intial scanning interval  # toX is the local upper limti of the range  # x1 is the root  Rneg, Rpos, h = -4.9, 4.6, 0.1  fromX = Rneg  toX = h+Rneg  itera = 0  x = np.linspace(-4.9, 4.6, 100)  y = fx(x)  plt.plot(x,y)  plt.grid(b=True);  while (toX < Rpos):    if (np.sign(fx(fromX)) != np.sign(fx(toX))):      x1,toX,iter = bisection(fx, fromX, toX + h)      plt.ylim(-3,  3)      plt.plot(x1, fx(x1), markersize=10, marker='o')      fromX=toX      itera = itera + iter    else:      toX = toX + h  print('Total iteration: = {0}'.format(itera), '\n')  print('x = {0:.20f} f(x) = {1:.20f}'.format(x1, fx(x1)))  # Question 1.3b for gx  # Rneg and Rpos are the intial range of the scan  # h is the intial scanning interval  # toX is the local upper limti of the range  # x1 is the root  Rneg, Rpos, h = 6, 9, 0.01  fromX = Rneg  toX = h+Rneg  itera = 0  x = np.linspace(6, 9, 100)  y = gx(x)  plt.plot(x,y)  plt.grid(b=True);  while (toX < Rpos):    if (np.sign(gx(fromX)) != np.sign(gx(toX))):      x1,toX,iter = bisection(gx, fromX, toX + h)      plt.plot(x1, gx(x1), markersize=30, marker='|')      plt.ylim(-1,  3)      fromX=toX      itera = itera + iter    else:      toX = toX + h  print('Total iteration: = {0}'.format(itera), '\n') |

## Bisection Method

|  |
| --- |
| 1. # bisection method 2. # at is the accepted tolerence 3. def bisection(func, xFrom, xTo): 4. xmid = (xFrom + xTo) / 2 5. iter = 0 6. at = 1e-10 7. while (np.abs(func(xmid)) > at): 8. iter += 1 9. if (np.sign(func(xmid)) == np.sign(func(xFrom))): 10. xFrom = xmid 11. else: 12. xTo = xmid 13. xmid = (xFrom + xTo) / 2 15. print("Root of f(x)= {0:.20f}".format(func(xmid))) 16. print("Root = {0:.20f}".format(xmid)) 17. print('xFrom= {0:.20f}; xTo= {1:.20f}'.format(xFrom, xTo),'\n')   24     return xmid,xTo,iter |

## Chord Method

|  |
| --- |
| # Chord method  # at is the accepted tolerence  def chord(func, xFrom, xTo):    k = np.abs((func(xFrom)/func(xTo)))    xmid = ((xFrom+(k\*xTo))/(1+k))    iter = 0    at = 1e-10    while (np.abs(func(xmid)) > at):          iter += 1          if (np.sign(func(xmid)) == np.sign(func(xFrom))):              xFrom = xmid          else:              xTo = xmid          k = np.abs((func(xFrom)/func(xTo)))          xmid = ((xFrom+(k\*xTo))/(1+k))    print("Root of f(x) = {0:.20f}".format(func(xmid)))    print("Root = {0:.20f}".format(xmid))    print('xFrom = {0:.20f};  = {1:.20f}'.format(xFrom, xTo),'\n')    return xmid,xTo,iter |

## Newton Method

|  |
| --- |
| def Dff(f,a,h=0.01):    return (f(a + h) - f(a - h))/(2\*h)  def newton(func, dFunc, x):    iter = 0    while (np.abs(func(x)) > 1e-10):      x = x - (1 /dFunc(func,x)\*func(x))      iter = iter + 1    return x,iter |

## Chord Method of 𝑓(𝑥) and 𝑔(𝑥)

|  |
| --- |
| # Question 1.4a (chord Method) for fx  Rneg, Rpos, h = -5.9, 4.6, 0.1  fromX = Rneg  toX = h+Rneg  itera = 0  x = np.linspace(-5.9, 4.6, 100)  y = fx(x)  plt.plot(x,y)  plt.grid(b=True);  while (toX < Rpos):    if (np.sign(fx(fromX)) != np.sign(fx(toX))):      x1,toX,iter = chord(fx, fromX, toX + h)  plt.ylim(-1,  3)      plt.plot(x1, fx(x1), markersize=10, marker='o')      fromX=toX      itera = itera + iter    else:      toX = toX + h  print('Total iteration: = {0}'.format(itera), '\n')  print('x = {0:.20f} f(x) = {1:.20f}'.format(x1, fx(x1)))  # Question 1.4b (chord Method) for gx  Rneg, Rpos, h = 6, 9, 0.01  fromX = Rneg  toX = h+Rneg  itera = 0  x = np.linspace(6, 9, 100)  y = gx(x)  plt.plot(x,y)  plt.grid(b=True);  while (toX < Rpos):    if (np.sign(gx(fromX)) != np.sign(gx(toX))):      x1,toX,iter = chord(gx, fromX, toX + h)  plt.ylim(-1,  3)      plt.plot(x1, gx(x1), markersize=30, marker='|')      fromX=toX      #toX = toX + h      itera = itera + iter    else:      toX = toX + h  print('Total iteration: = {0}'.format(itera), '\n') |

## Newton’s Method of 𝑓(𝑥) and 𝑔(𝑥)

|  |
| --- |
| #Newton Method for f(x)  xStart,xEnd, h = -5.9, 4.6, 0.01  itera = 0  x = np.linspace(-5.9, 4.6, 100)  y = fx(x)  plt.plot(x,y)  plt.grid(b=True);  while (xStart + h < xEnd):    if (np.sign(fx(xStart)) != np.sign(fx(xStart+h))):      x1,itera = newton(fx,Dff, xStart)      print('Root Values: {0:.20f}'.format(x1), '\n')      plt.ylim(-3,  3)      plt.plot(x1, fx(x1), markersize=10, marker='o')      itera = itera + iter      xStart = xStart + h  print('Total iteration: = {0}'.format(itera), '\n')  #Newton Method for g(x)  xStart,xEnd, h = 6, 9, 0.01  itera = 0  x = np.linspace(6, 9, 100)  y = gx(x)  plt.plot(x,y)  plt.grid(b=True);  while (xStart + h < xEnd):    if (np.sign(gx(xStart)) != np.sign(gx(xStart+h))):      x1,itera = newton(gx,Dff, xStart)      print('Root Values: {0:.20f}'.format(gx(x1)), '\n')      plt.ylim(-1,  3)      plt.plot(x1, gx(x1), markersize=30, marker='|')      itera = itera + iter      xStart = xStart + h  print('Total iteration: = {0}'.format(itera), '\n') |

## Tasks numbers: 11-15

|  |
| --- |
| # given function T(t)  def Tt(k):    x= k\*20    return (473-373)\*np.exp(x)+373-378  Rneg, Rpos, h = -1, 1, 0.001  fromX = Rneg  toX = h+Rneg  itera = 0  x = np.linspace(-0.3, 0.3, 100)  y = Tt(x)  plt.plot(x,y)  plt.grid(b=True);  while (toX < Rpos):    if (np.sign(Tt(fromX)) != np.sign(Tt(toX))):      x1,toX,iter = bisection(Tt,  fromX, toX + h)      plt.ylim(-10,  10)      plt.plot(x1, Tt(x1), markersize=10, marker='o')      fromX=toX      #toX = toX + h      itera = itera + iter    else:      toX = toX + h  print('Total iteration: = {0}'.format(itera), '\n') |