CSE 401: Numerical Analysis
Course Project

Bhavesh Shrimali

NetID: bshrima2

December 7, 2016

Problem 5.22

The problem is to be solved using the fixed-point iteration scheme. Given problem

$$
\begin{aligned}
x_1 &= -\frac{\cos x_1}{81} + \frac{x_2^2}{9} + \frac{x_3}{3} \\
x_2 &= \frac{\sin x_1}{3} + \frac{\cos x_3}{3} \\
x_3 &= -\frac{\cos x_1}{9} + \frac{x_2}{3} + \frac{\sin x_3}{6}
\end{aligned}
$$

Let $\mathbf{x}$ denote the unknown vector and $\mathbf{G(x)}$ denote the corresponding fixed point function on the RHS. Using this we have

$$\mathbf{x} = \mathbf{G}(\mathbf{x})$$

where

$$\mathbf{x} = \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} \quad ; \quad \mathbf{G}(\mathbf{x}) = \begin{Bmatrix} -\frac{\cos x_1}{81} + \frac{x_2^2}{9} + \frac{\sin x_3}{3} \\ \frac{\sin x_1}{3} + \frac{\cos x_3}{3} \\ -\frac{\cos x_1}{9} + \frac{x_2}{3} + \frac{\sin x_3}{6} \end{Bmatrix}$$

(a): Fixed-Point Iteration

The solution is defined as reached, when the absolute error in the fixed point iteration reaches the level of machine-precision given by the numpy library for the float data-type. The corresponding fixed point solution, obtained from a starting vector of $\mathbf{x}_0 = [0, 0, 0]^T$ is given by

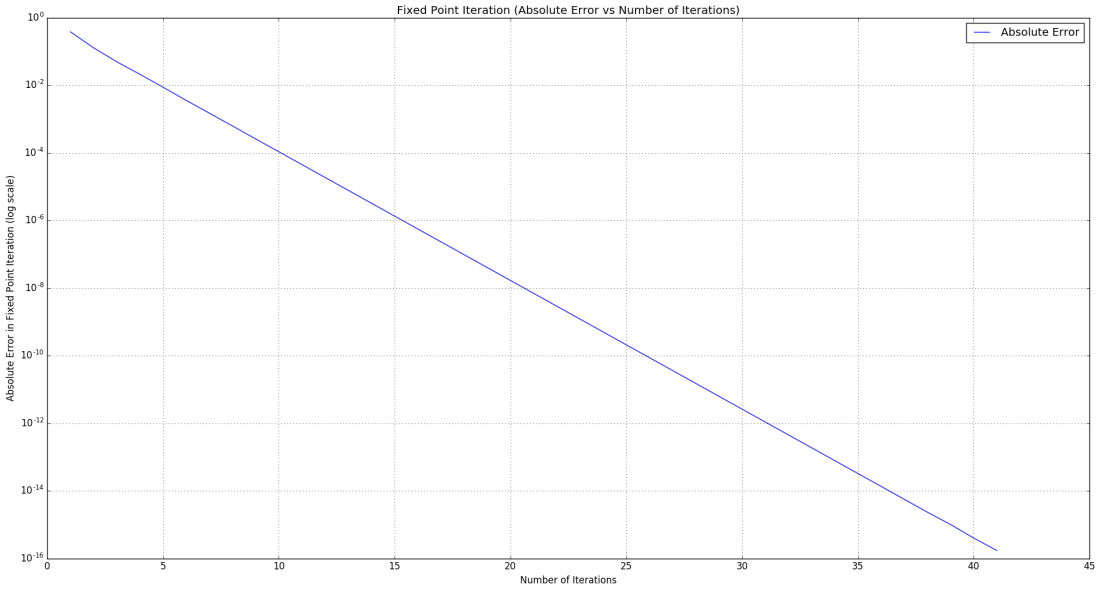
$$\mathbf{x}_{\text{sol}} = \begin{Bmatrix} 0 \\ 1/3 \\ 0 \end{Bmatrix}$$

(b): Convergence rate

The fixed-point iteration takes 41 iterations to converge to the solution. The value of C, in the linear convergence rate, at the fixed point solution is given by the (absolute value of) spectral radius of the Jacobian matrix of the fixed point function:

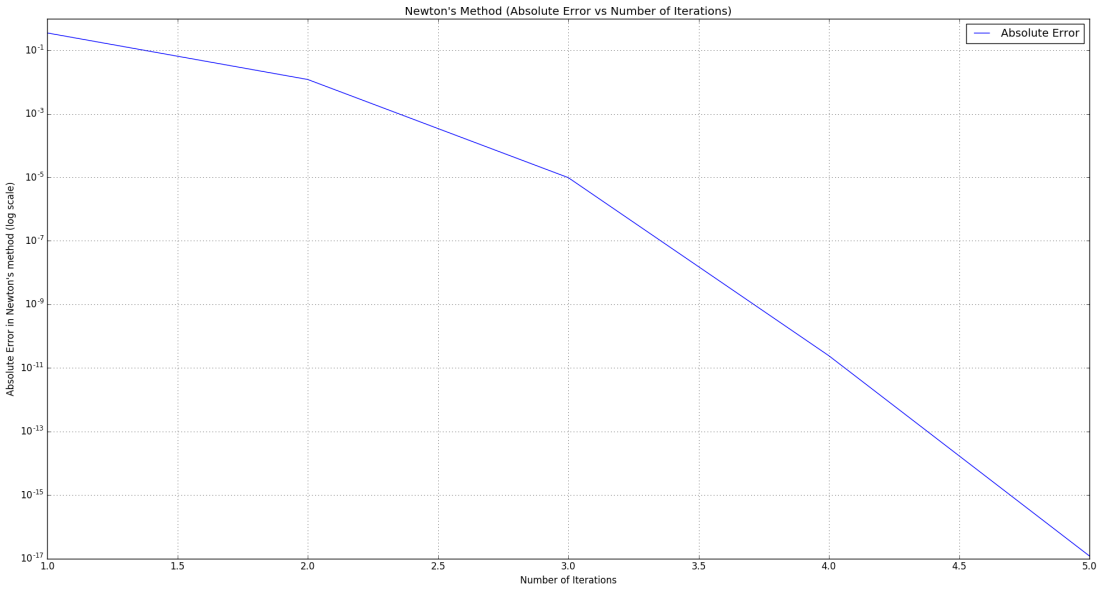
$$|\rho(\mathbf{G}(\mathbf{x}_{\text{sol}}))| = 0.4161$$

The semi-log curve for the absolute error vs the number of iterations is given as shown:



(c): Newton's Method

When solved with the Newton's method, the problem takes 5 iterations (quadratic convergence) to converge to the solution. The corresponding plot of the absolute error vs the number of iterations is given as shown:



Problem 5.24:

The system of equations to be solved:

$$\begin{aligned} w_1 + w_2 &= 2 \\ w_1 x_1 + w_2 x_2 &= 0 \\ w_1 x_1^2 + w_2 x_2^2 &= \frac{2}{3} \\ w_1 x_1^3 + w_2 x_2^3 &= 0 \end{aligned} \tag{1}$$

where w_1, w_2 denote the weights and x_1 and x_2 denote the corresponding gauss points. I use *scipy.optimize.root* to determine the solution of the above nonlinear system of equations. The guesses used for the problem are as follows:

$$\begin{Bmatrix} w_1 \\ w_2 \\ x_1 \\ x_2 \end{Bmatrix}_{\text{trial}} = \begin{Bmatrix} 1.0 \\ 1.0 \\ 0.577 \\ -0.577 \end{Bmatrix}$$

which verifies the analytically the existing solution:

$$\begin{Bmatrix} w_1 \\ w_2 \\ x_1 \\ x_2 \end{Bmatrix}_{\text{computed}} = \begin{Bmatrix} 1.0 \\ 1.0 \\ 0.577 \\ -0.577 \end{Bmatrix} \tag{2}$$

Another guess

$$\begin{Bmatrix} w_1 \\ w_2 \\ x_1 \\ x_2 \end{Bmatrix}_{\text{trial}} = \begin{Bmatrix} 1.0 \\ 1.0 \\ 0.0 \\ 0.0 \end{Bmatrix}$$

yields the following solution

$$\begin{Bmatrix} w_1 \\ w_2 \\ x_1 \\ x_2 \end{Bmatrix}_{\text{computed}} = \begin{Bmatrix} 1.000001 \\ 0.999999 \\ 0.577 \\ -0.577 \end{Bmatrix}$$

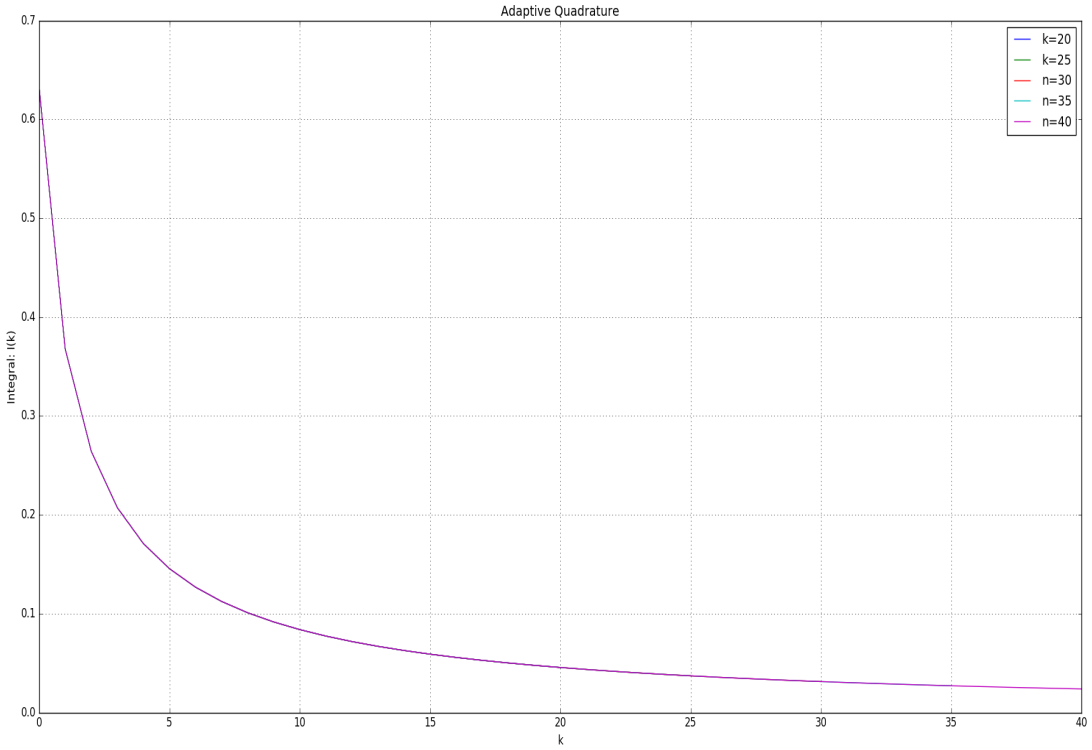
Several other perturbations result in very slight variation in the solution. Hence there is a single solution to the above system of nonlinear equations. It is, however, important to note that all the guesses which are considered here satisfy the first equation in (**??**), atleast. A starting guess of $[0, 0, 0, 0]'$ doesn't result in convergence and hence is avoided.

Problem 8.6:

The problem specifies to use an adaptive quadrature routine to compute the integral. For this, the library routine *scipy.integrate.quad* is employed to compute the integral. The integral values obtained from the different methods are plotted below to compare and contrast:

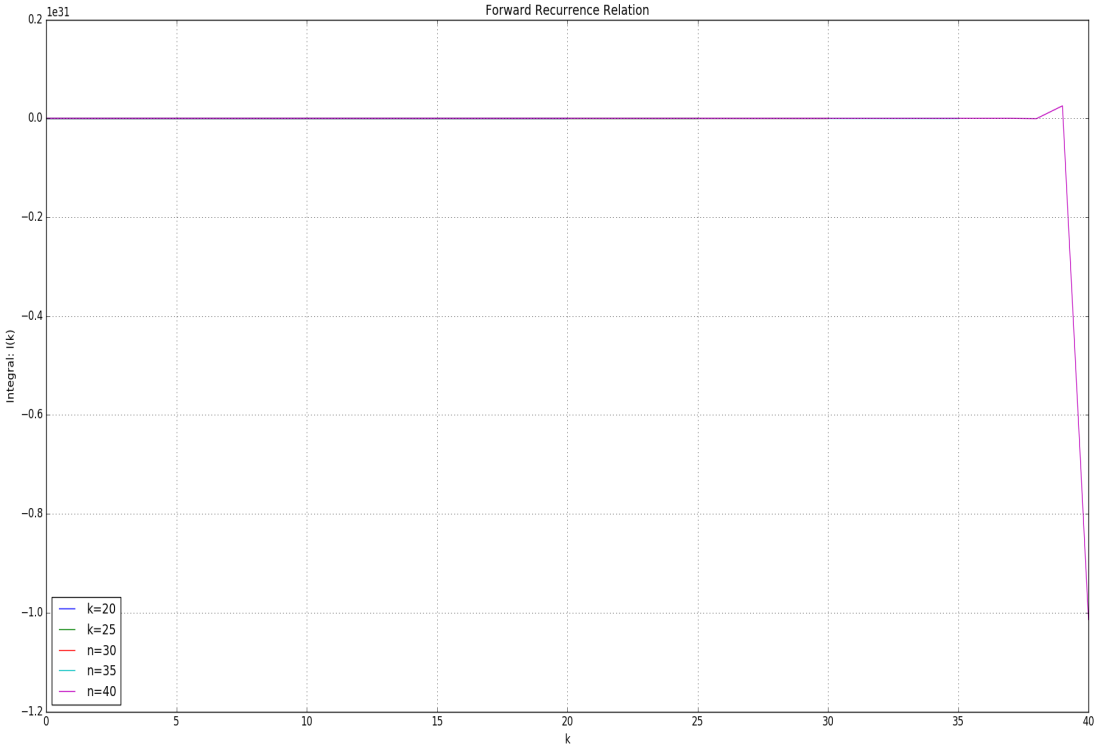
(a): Adaptive Quadrature

The plot below shows the computed integral values as a function of the parameter k



(b): Forward Recurrence Relation

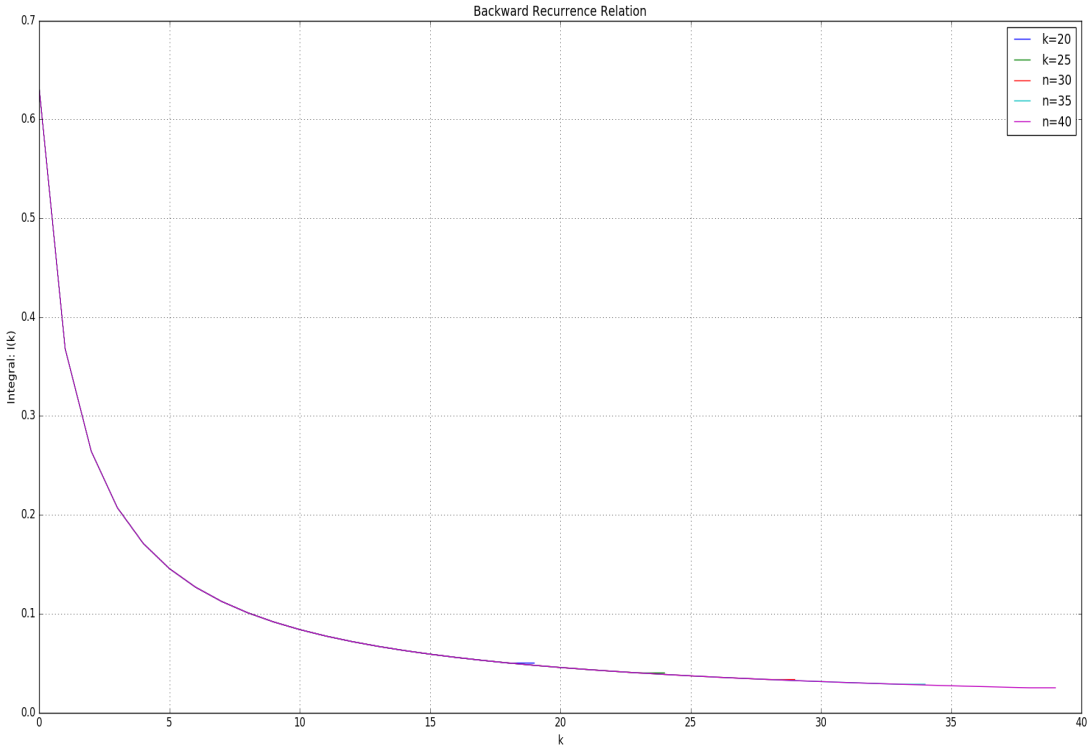
The plot below shows the computed integral values as a function of the parameter k using the forward recurrence relation.



It is plain from the above plot that the forward recurrence relation is increasingly unstable for larger values of the parameter k.

(c): Backward Recurrence Relation

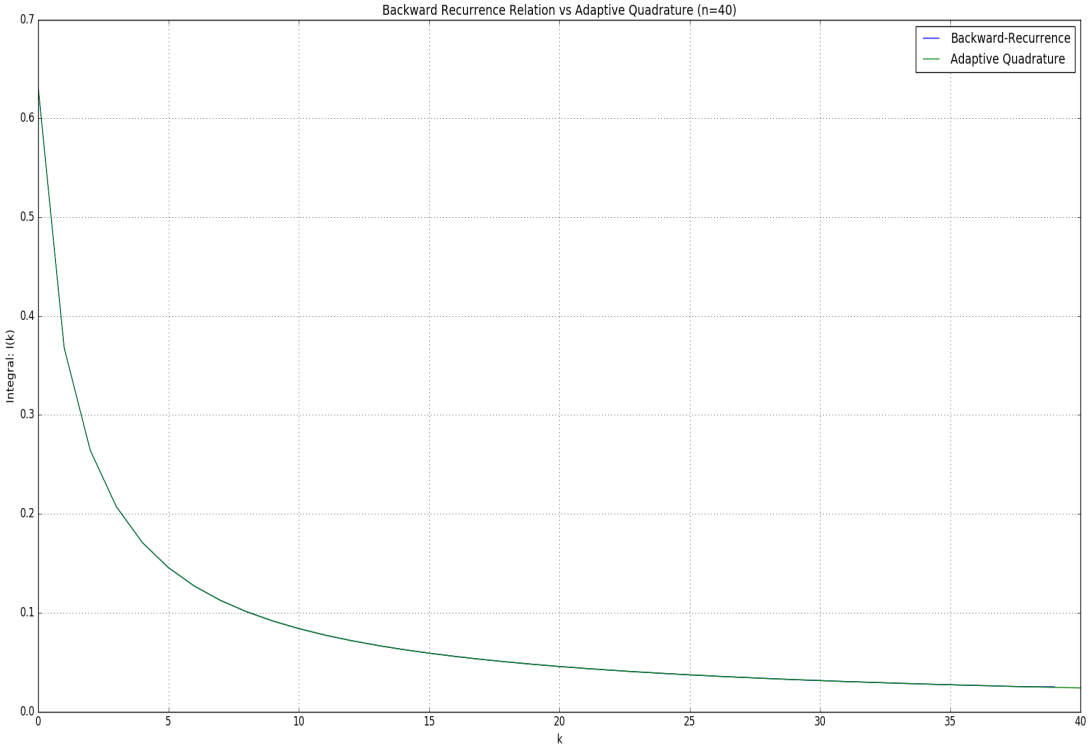
The plot below shows the computed integral values as a function of the parameter k using the backward recurrence relation.



The backward recurrence relation, unlike the forward recurrence relation, is much more stable for larger values of the parameter k.

(d): Comparison between Adaptive Quadrature and Backward Recurrence

The plot below shows a comparative plot between the backward recurrence formula and the adaptive quadrature rule for $k = 40$. We choose the maximum value of the parameter, among the set of values, in order to inspect the stability of the adaptive quadrature with respect to the backward recurrence formula and vice-versa.



- As a conclusive remark the forward recurrence relation is unstable for higher values of k, and the adaptive quadrature library routine and the backward recurrence formula are stable.

(e): Comparison of Run-time

- Since the backward and forward recurrence formula involve only substitution, they are ought to be faster than the adaptive quadrature routine, (*quad*). However *quad* is more accurate as compared to both of these.
- The Backward recurrence formula assumes the value is equal to zero for a sufficiently large n and then proceeds in a backward manner to calculate. Hence there is no discretization error (which is incurred in case of the forward recurrence formula).

Problem 8.10: Gamma Function

The given function to be integrated is:

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t}\, dt \quad ; \qquad x > 0$$

(a): Composite Quadrature Rule - Truncated interval

For Composite-Quadrature rule, we choose the library routine *scipy.integrate.simps* to compute the integral. The limit of integration imposed on the integration parameter t is $(0, 10^6)$ instead of the infinite integral, thereby using a truncated limit for the integral. The limit of integration which is used in this case, is determined after succesive trials with different upper limits, thereby making consideration for accuracy and runtime.

(b): Adaptive Quadrature Rule - Truncated interval

In this case we use the truncated interval to compute the integral and corresponding values are later plotted. It is observed that for the chosen truncation limit, the integral is approximated well by the truncated limits as opposed to the infinite integration limit.

(c): Adaptive Quadrature - Infinite Interval

The corresponding infinite integral is plotted below

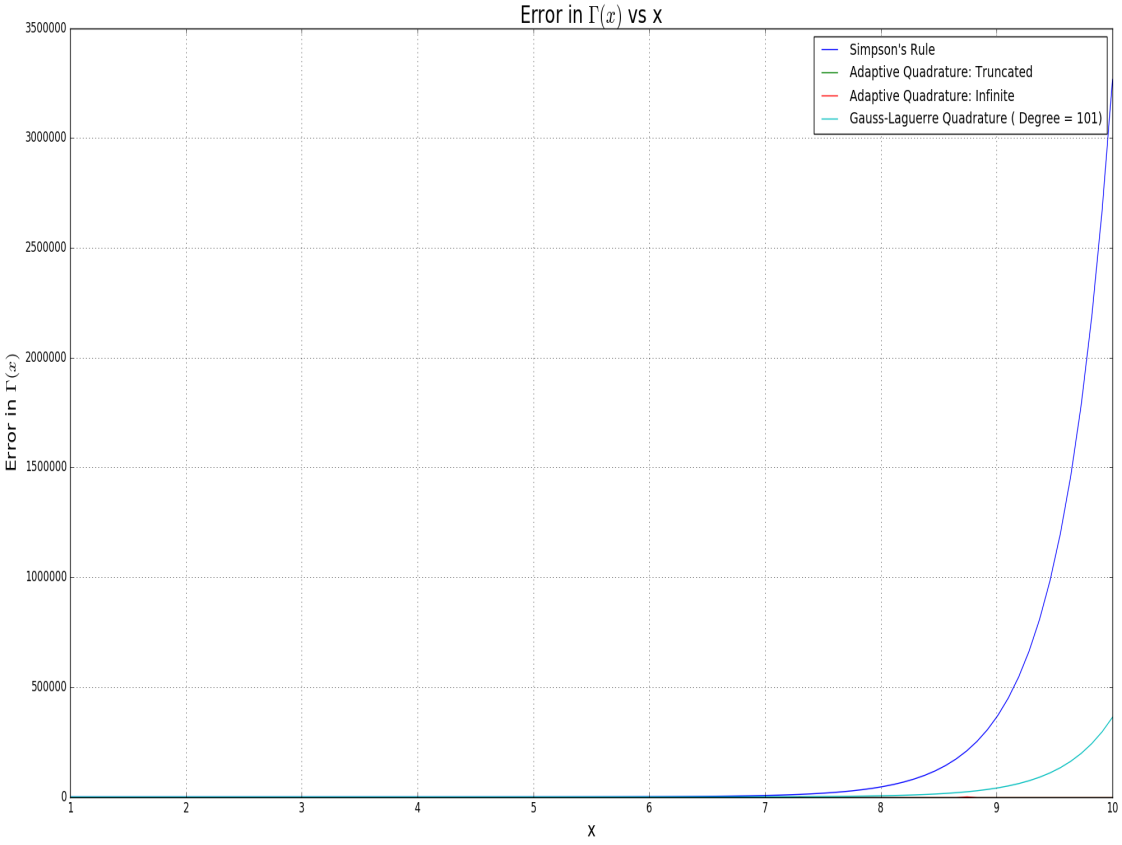
(d): Gauss-Laguerre Quadrature - Degree (101)

The Gauss-Laguerre Quadrature is better than the Simpson's rule for integration but is worse, in accuracy, than the Adaptive Quadrature. The corresponding plot (with all the methods) is attached below:

Comments:

The following observations can be made:

- The Simpson's rule incurs the largest error among all the methods.
- The Adaptive Quadrature is very accurate even when evaluating the infinite integral.
- The Gauss-Laguerre Quadrature, for the chosen degree to calculate the infinite integral, is more accurate than the Simpson's rule (Composite Quadrature) but is less accurate than the adaptive quadrature.


Error in $\Gamma(x)$ vs x
Simpson's Rule
Adaptive Quadrature: Truncated
Adaptive Quadrature: Infinite
Gauss-Laguerre Quadrature (Degree = 101)
Error in $\Gamma(x)$
X
0
500000
1000000
1500000
2000000
2500000
3000000
3500000
1
2
3
4
5
6
7
8
9
10

Problem 8.17: Integral Equation:

The integral equation to be solved:

$$\int_0^1 \left(s^2+t^2\right)^{1/2} u(t)\, dt = \frac{\left(s^2+1\right)^{3/2} - s^3}{3}$$

The corresponding true solution is given by:

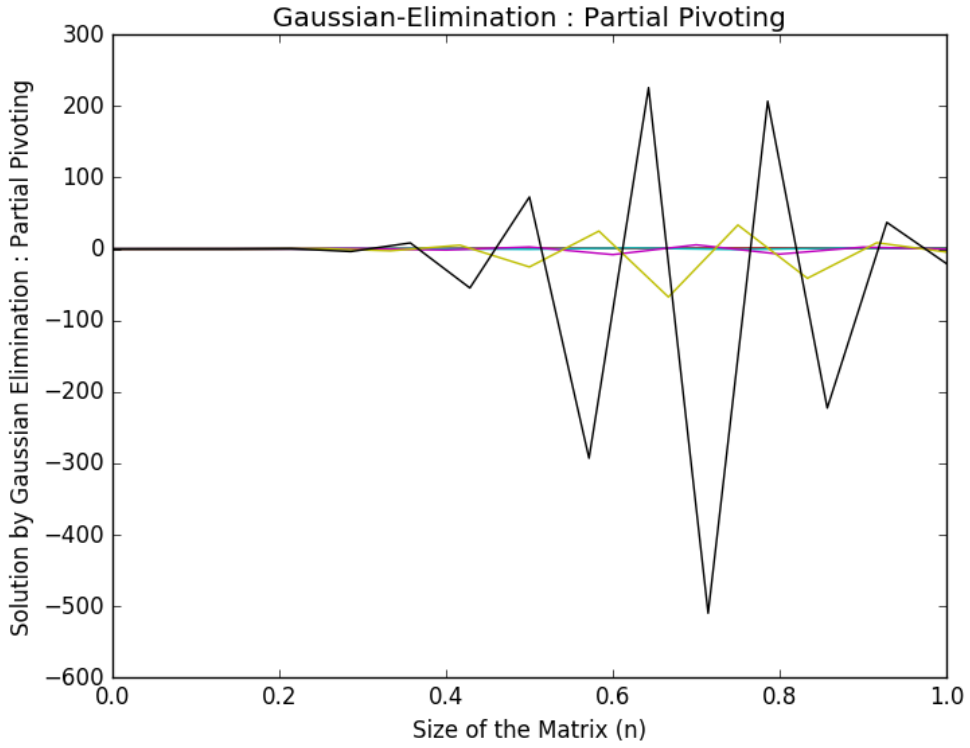
$$u(t) = t$$

Note:

It is noted that all the plots below are given for $n = 3, 5, 7, 9, 11, 15$

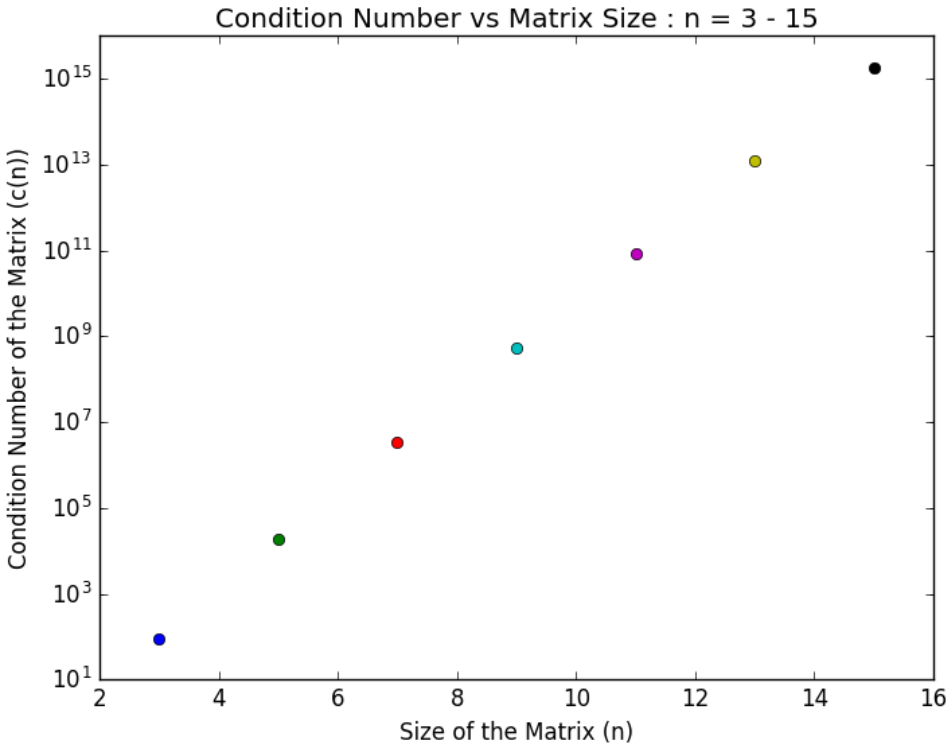
(a): Gaussian-Elimination with Partial Pivoting

Now we use the Simpson's rule (Composite Quadrature) to formulate the linear system which is then solved using the process of Gaussian-elimination with partial pivoting. The corresponding results obtained are as follows:



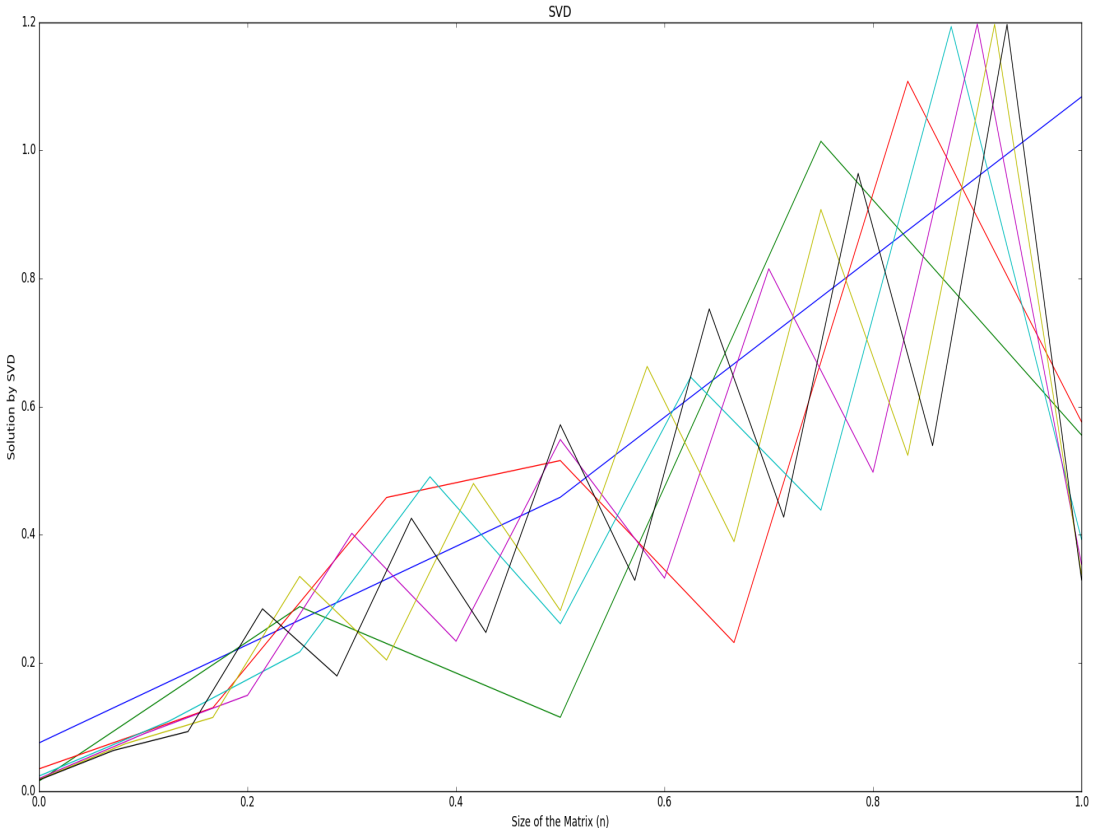
(b): Condition Number

The condition number of the matrix (**A**) as a function of (n) is plotted below. It can be seen that the condition number of the matrix **A**, increases with (**n**). Increasing the size of the matrix (n) makes the matrix increasingly ill-conditioned.



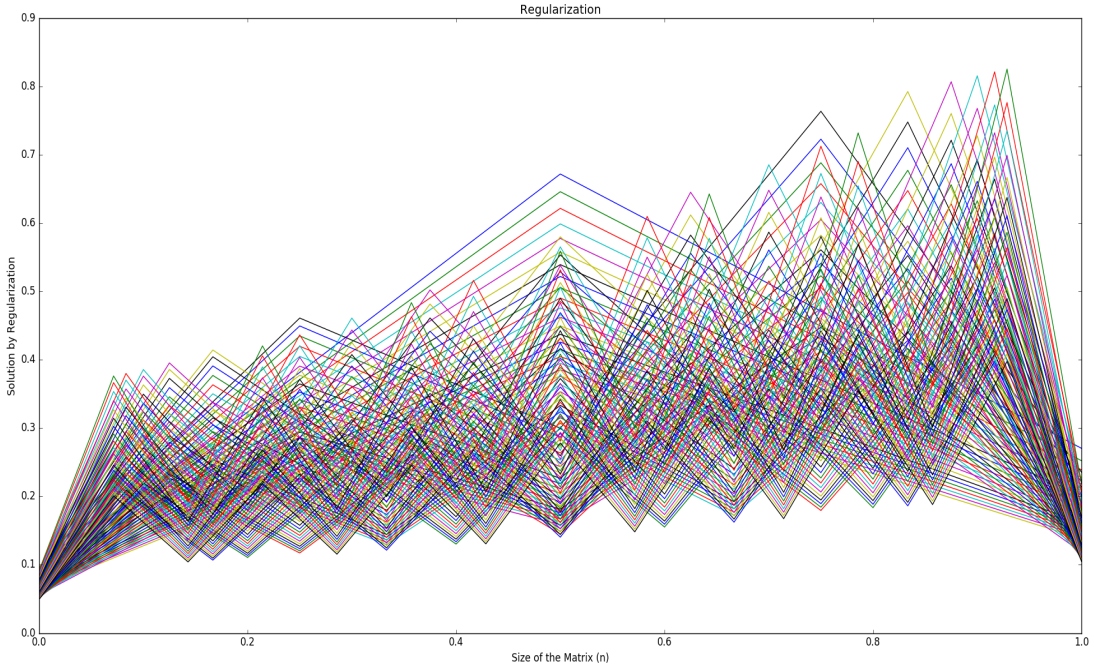
(c): SVD

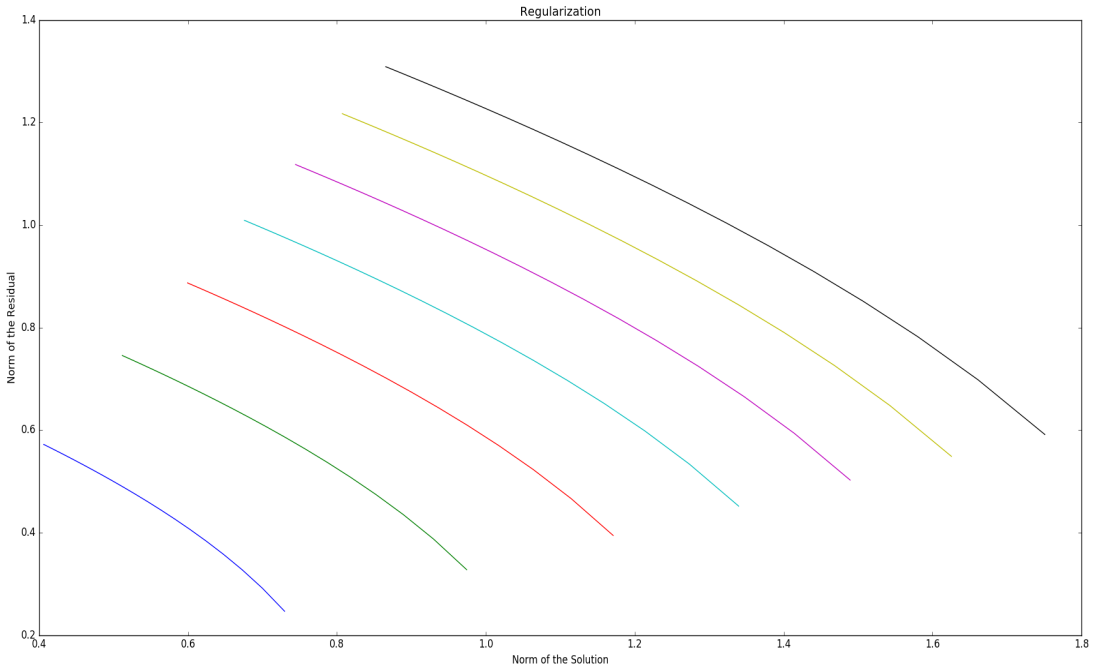
Here, in order to ignore the small singular values, through SVD, the criteria that is employed, to ignore a particular Singular values, is if the ratio of the maximum and the minimum singular values of the matrix. The corresponding plot is attached below:



(d): Regularization

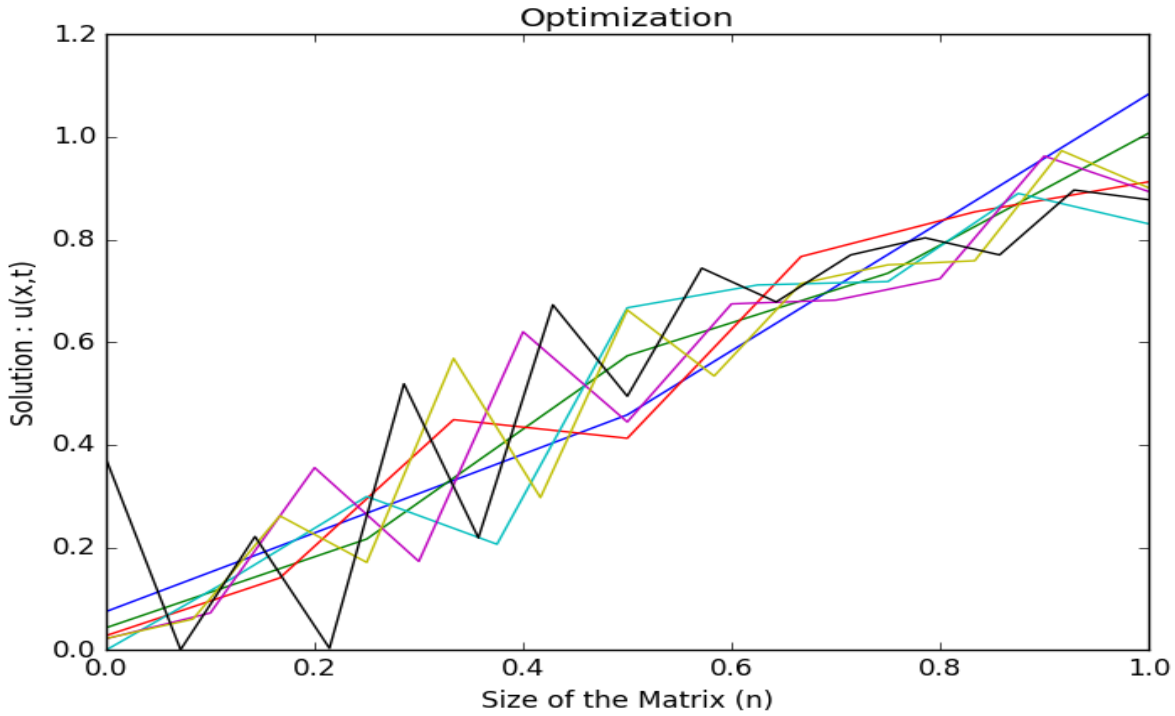
The optimal point is the point on the graph where both the norm of the solution and the norm of the residual are equal to each other $(0.9, 0.9)$. The corresponding plots are attached below:

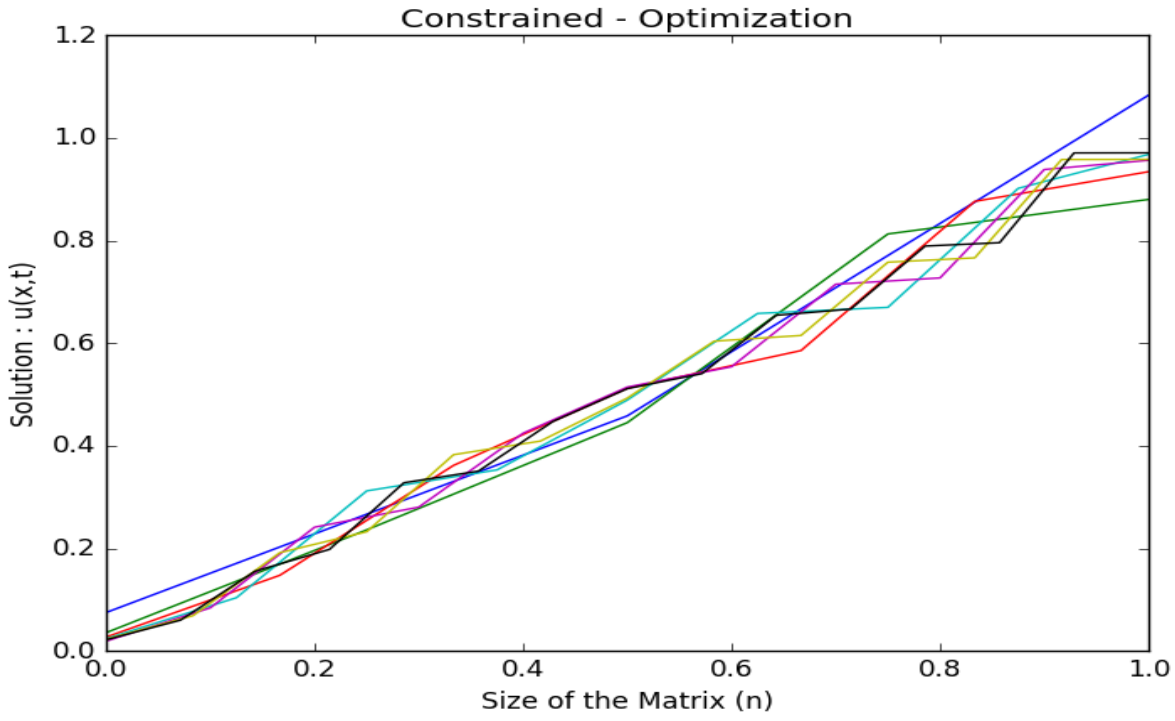




(e) and (f): Constrained and Unconstrained Optimization

The constrained optimization is the best among all the methods as can be observed by the plots below:





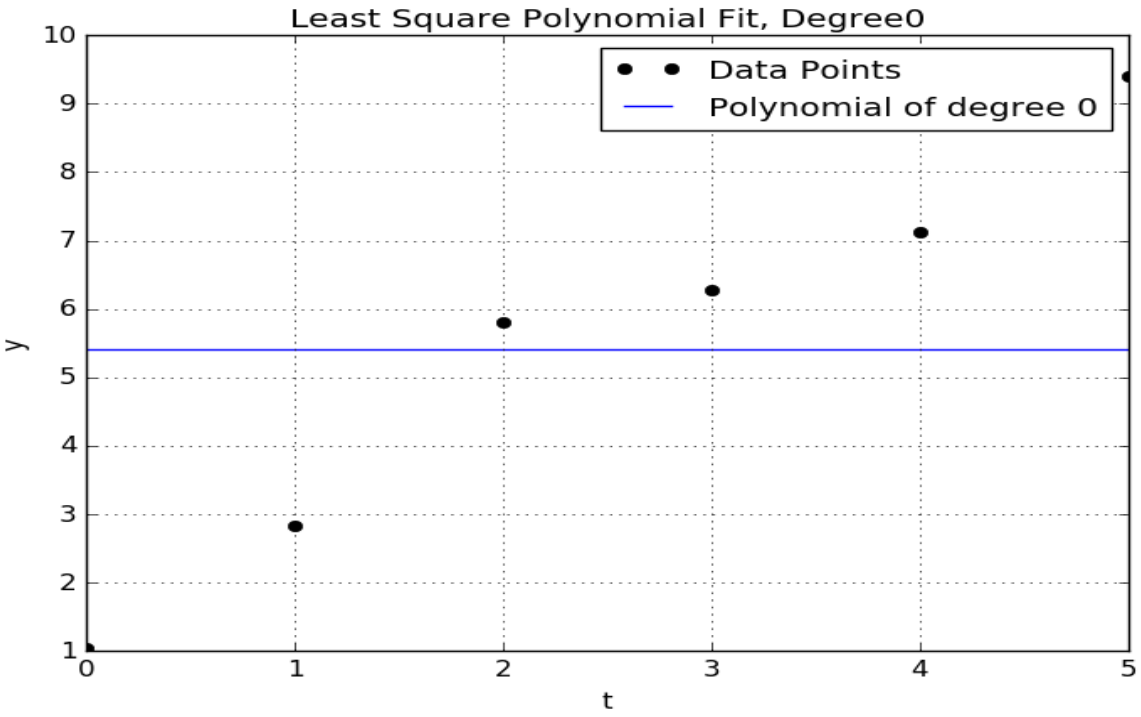
Notes:

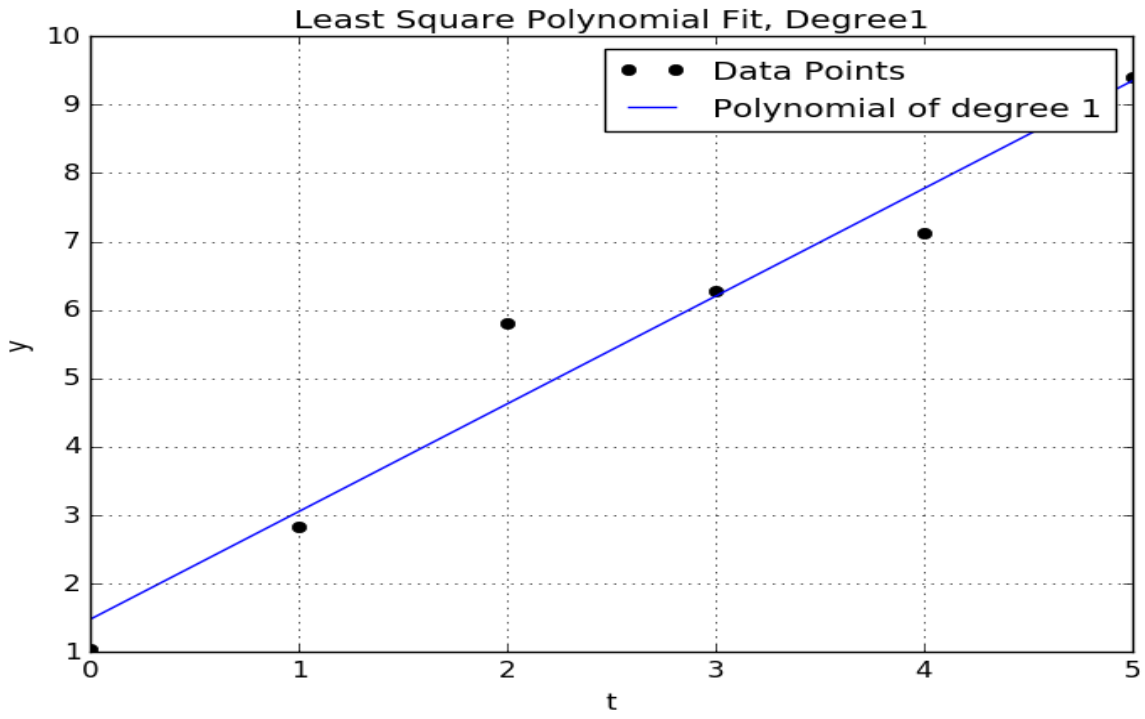
- For regularization the error is damped but it is very large and hence any particular improvement towards the true solution cannot be inferred from the graph.
- The Constrained Optimization follows the trendline for the true solution, the closest among all the methods. It is followed by the unconstrained optimization.
- The Gaussian Elimination with partial pivoting does not perform satisfactorily because the matrix becomes increasingly ill-conditioned. This is verified by the plot of the conditioning number with the matrix size (n).
- The solution given by the SVD fluctuate about the true solution (trendline) but the mean values are close to the actual solution. The negative and the positve errors cancel each other in one oscillation.

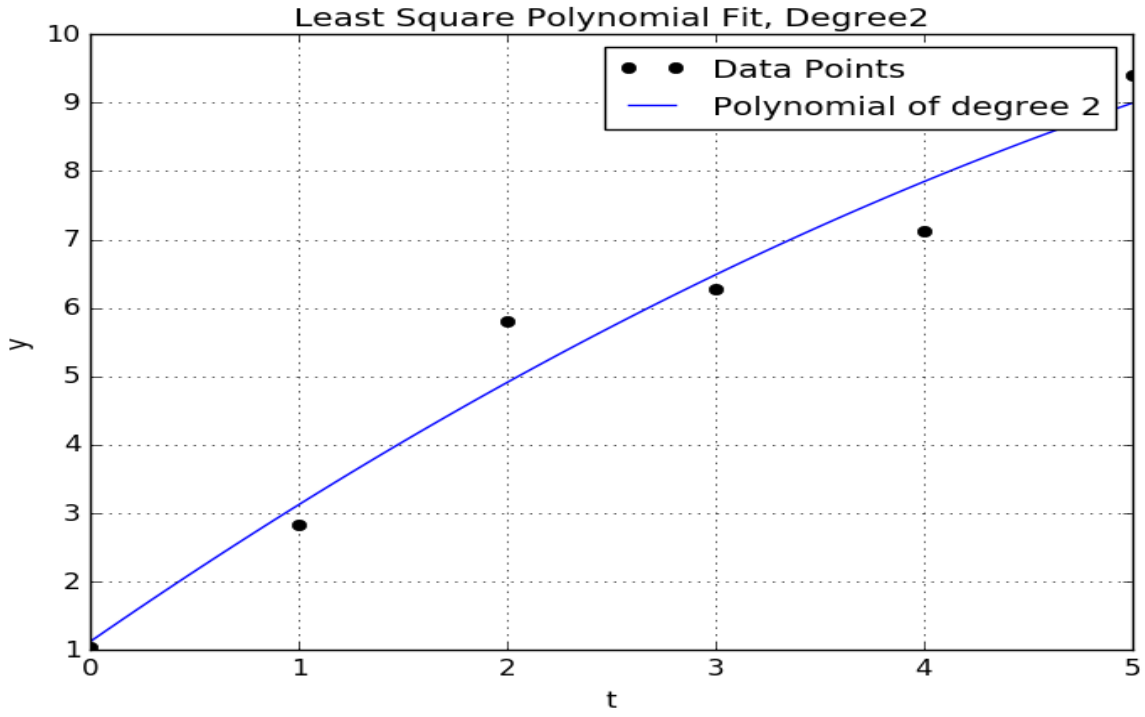
Problem 8.18: Poly-fits and Splines

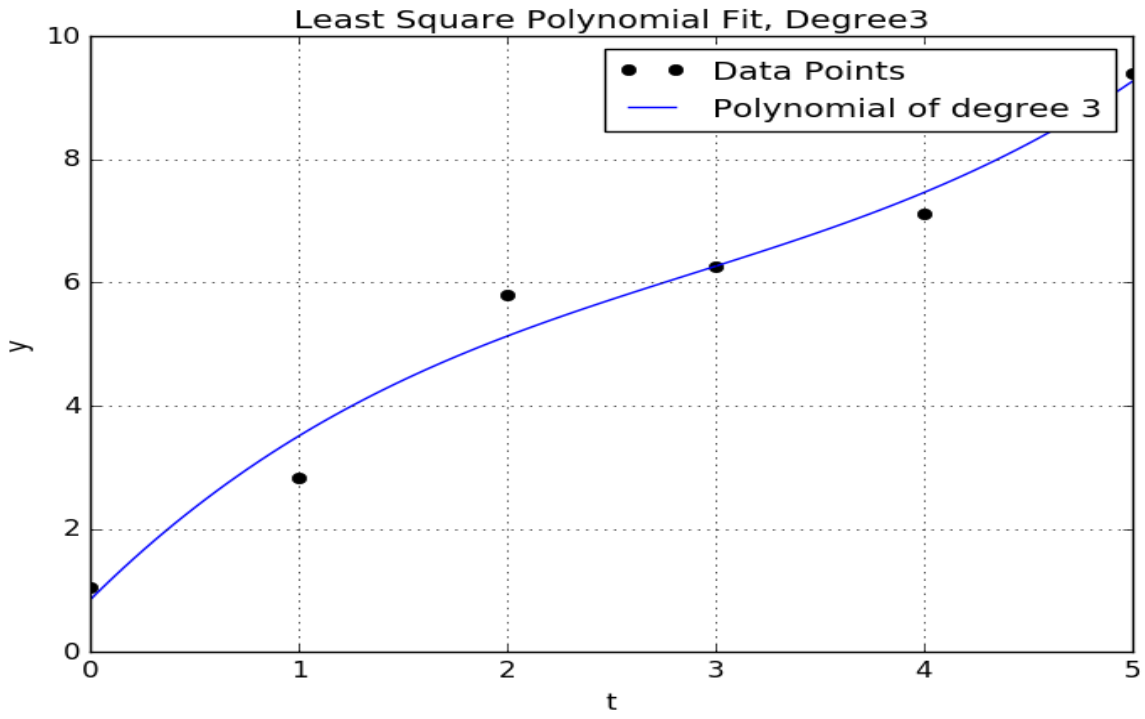
The given data points are interpolated through polynomial fits and splines of different types and the corresponding plots are attached below:

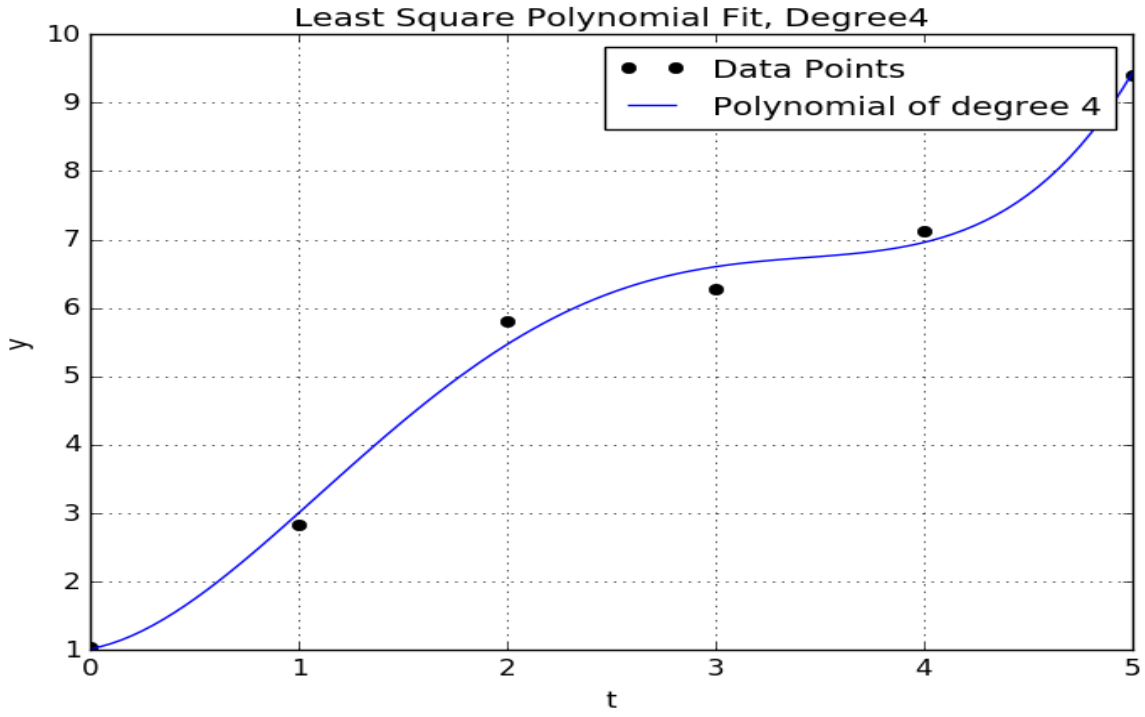
(a): Least Squares Polynomial Fits

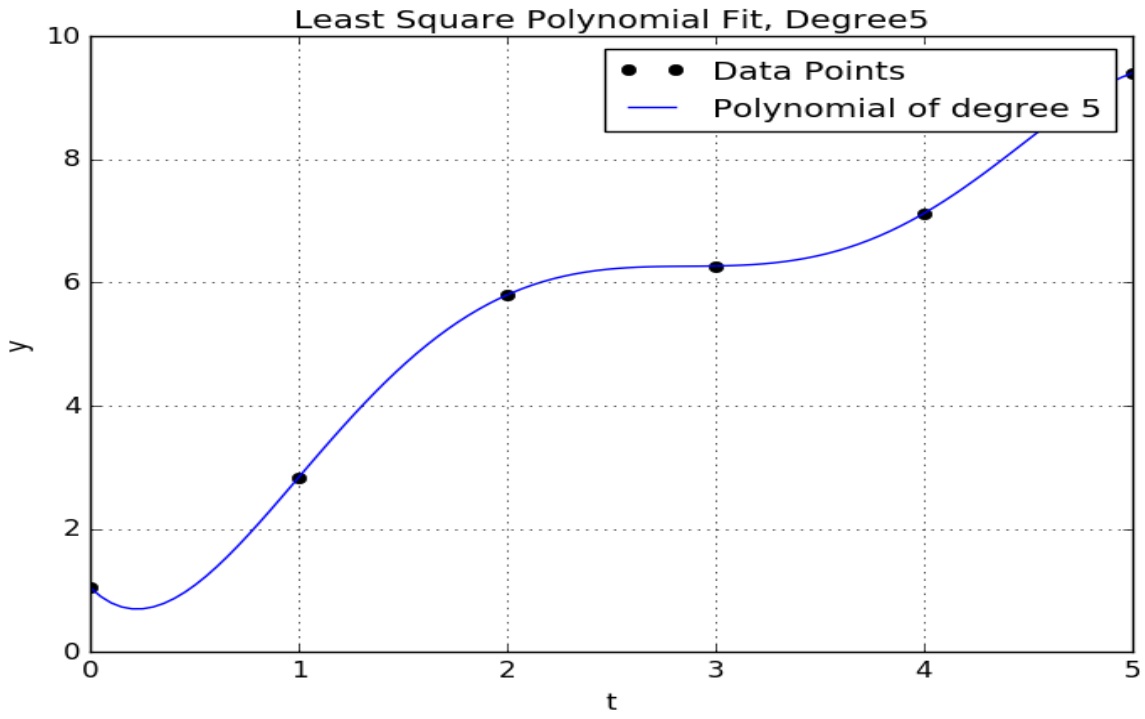



Least Square Polynomial Fit, Degree1
Data Points
Polynomial of degree 1
y
t
0
1
2
3
4
5
1
2
3
4
5
6
7
8
9
10

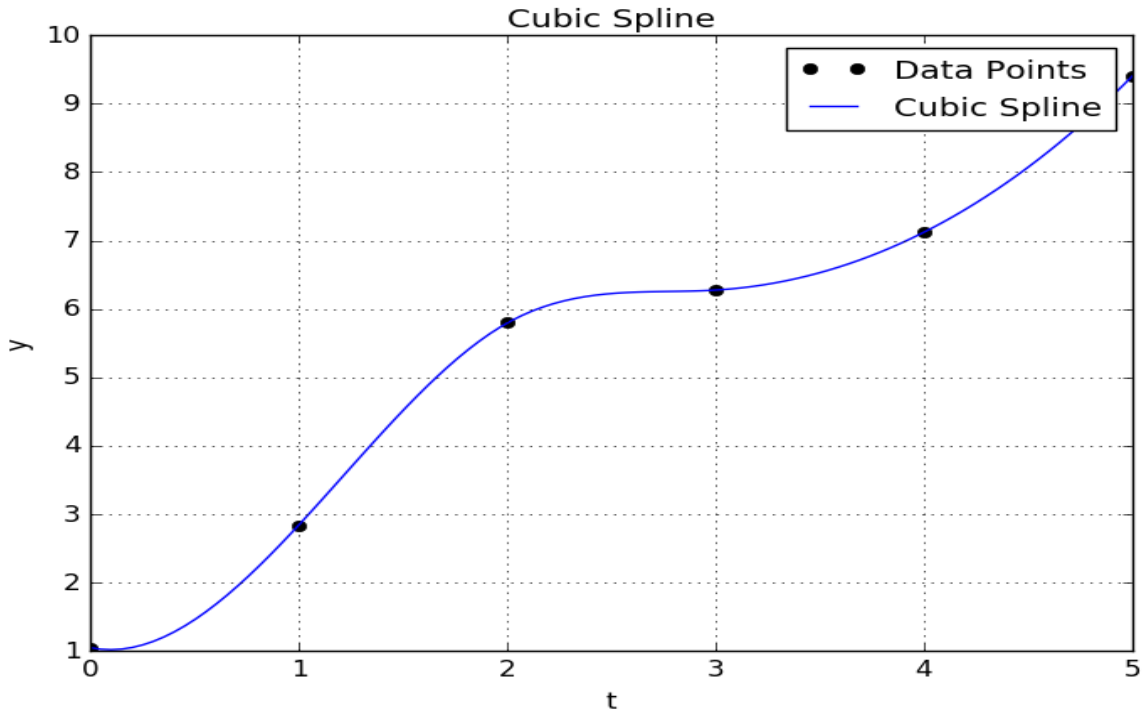

Least Square Polynomial Fit, Degree2
Data Points
Polynomial of degree 2
y
t
0
1
2
3
4
5
1
2
3
4
5
6
7
8
9
10


Least Square Polynomial Fit, Degree3
Data Points
Polynomial of degree 3
y
t
0
1
2
3
4
5
0
2
4
6
8
10

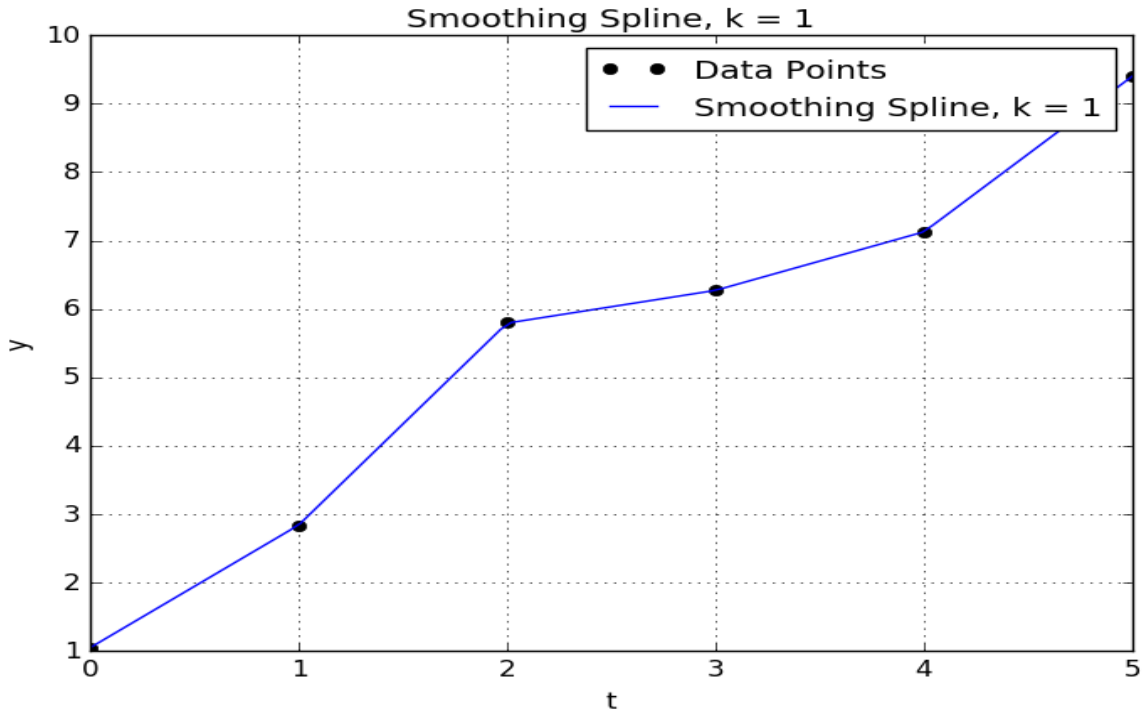

Least Square Polynomial Fit, Degree4
Data Points
Polynomial of degree 4
y
t
0
1
2
3
4
5
1
2
3
4
5
6
7
8
9
10

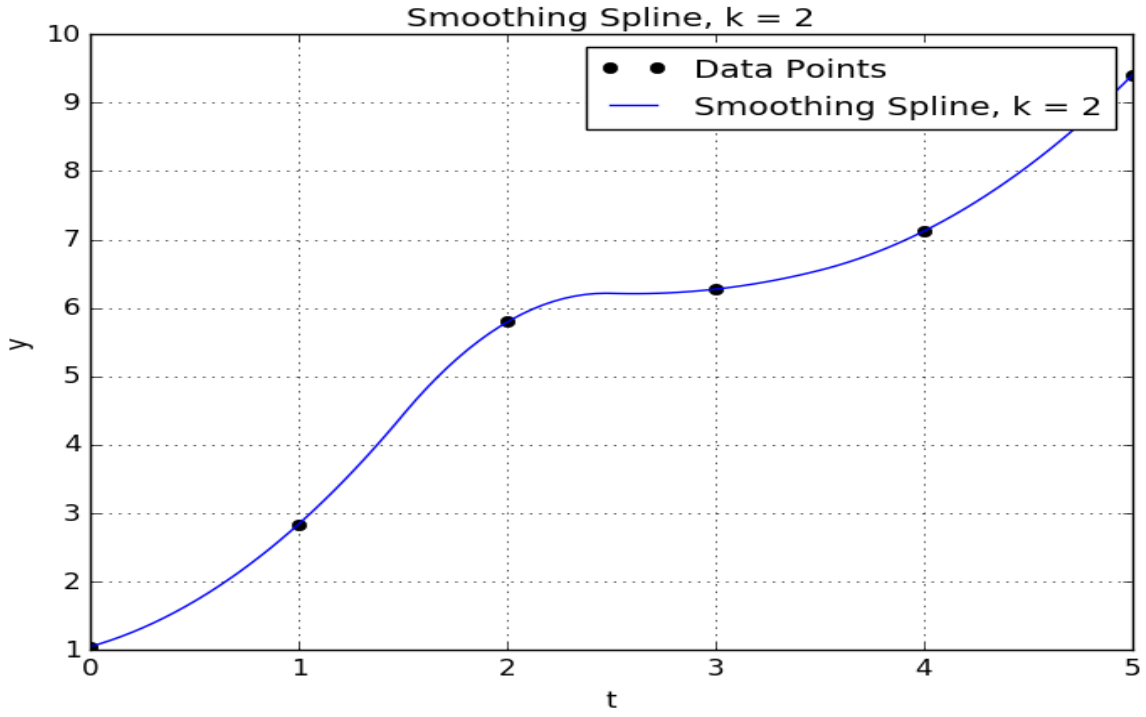


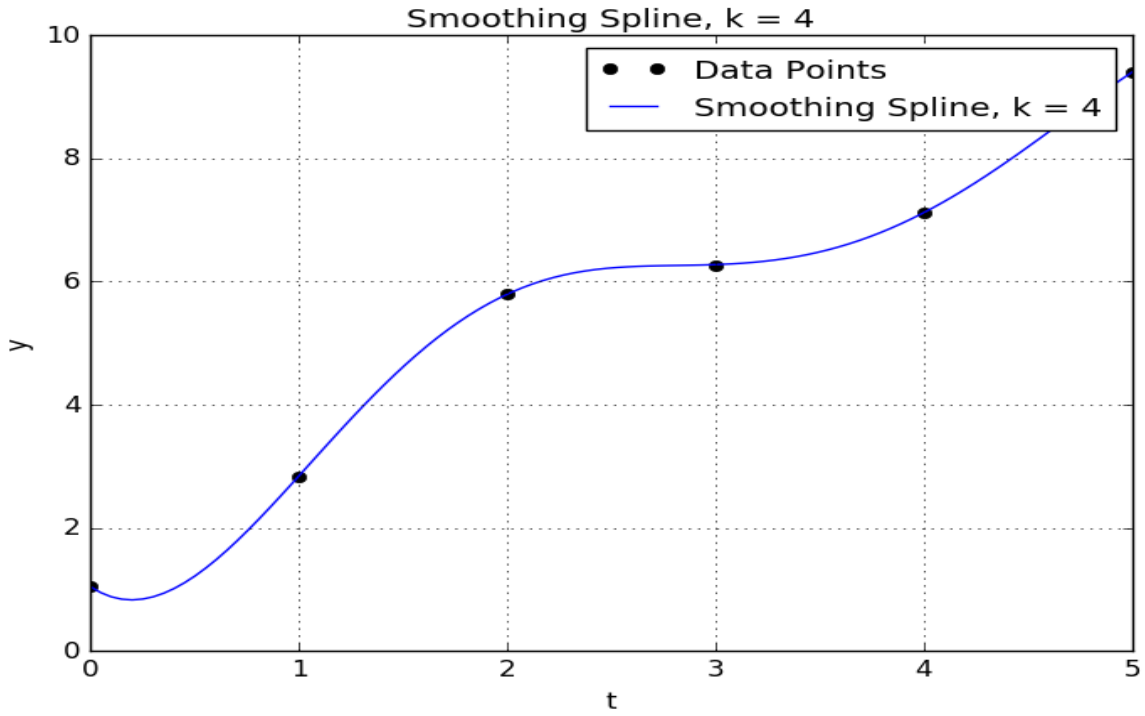
(b): Cubic Spline

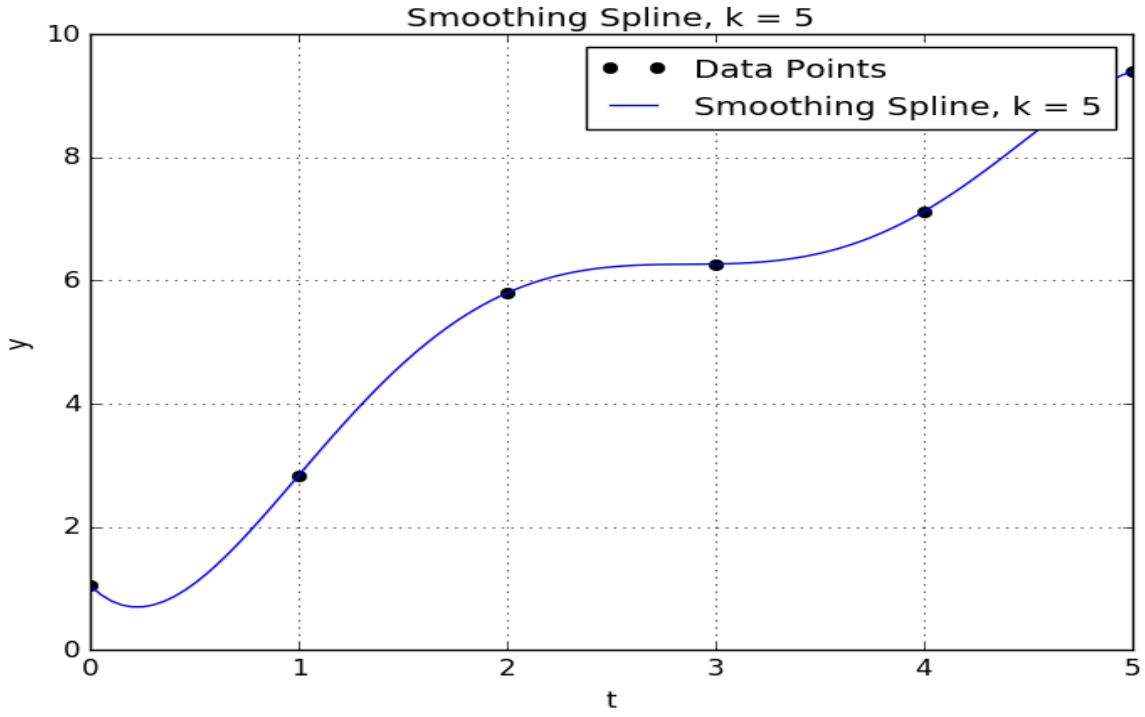


(c): Smoothening Spline of different degrees

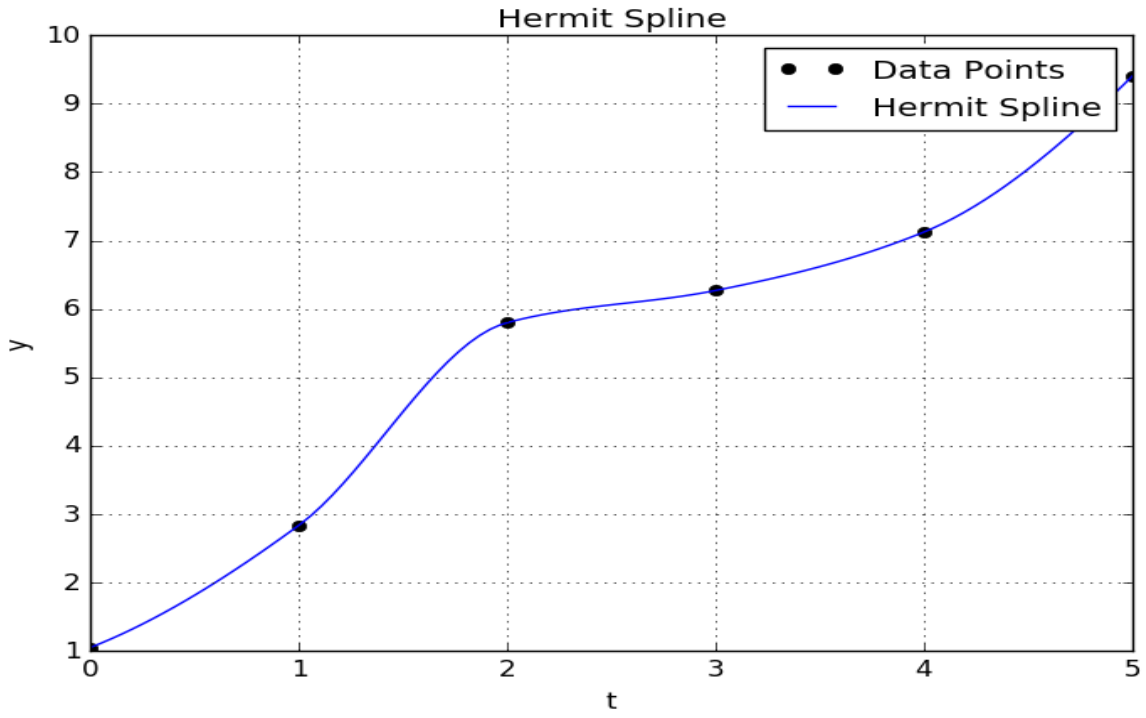





Smoothing Spline, k = 4
Data Points
Smoothing Spline, k = 4
y
t
0
1
2
3
4
5
0
2
4
6
8
10


Smoothing Spline, k = 5
Data Points
Smoothing Spline, k = 5
y
t
0
1
2
3
4
5
0
2
4
6
8
10

(d) Hermite Spline



Notes:

The Splines, in general are better in interpolating the data points smoothly, and also preserving the monotonicity of the data. Hermite Spline follows the trend of the data-set more closely as compared to the cubic-spline. Least Squares Polynomial Interpolants may/maynot pass through the data points always. Also several problems can be incurred (derivatives etc.) when the data set is perturbed.

Appendix: Python Codes

5.22

The Python code used to solve the problem 5.22 is appended below:

```
# -*- coding: utf-8 -*-
"""
Created on Sun Nov  6 14:50:36 2016

@author: bhavesh
"""

import numpy as np
from scipy.optimize import fsolve
import math as ma
import matplotlib.pyplot as plt

#ans = np.array([[11],[1],[1],[1]])
def g(x):
    return np.array([-1/81*ma.cos(x[0]) + 1/9 *(x[1])**2  ...
    + 1/3*(ma.sin(x[2])),(1/3*(ma.sin(x[0])+ma.cos(x[2]))),-1/9*ma.cos(x[0])...
    +1/3*(x[1])+1/6*(ma.sin(x[2]))])

# True solution as given by fsolve:
true_sol = fsolve(g,np.array([0,0.33,0]))

# Initial Starting guess for the problem: 0
err = 1
epsilon = np.finfo(float).eps
citer = 1
xin = np.array([1,1,1])
error_f = ([])
total_it = ([])
while err > epsilon:
    if citer ==1:
        xn = g(xin)
    else:
        xn = g(xn)
    err = np.linalg.norm(xn-g(xn))
    error_f.append(err)
    total_it.append(citer)
    citer = citer+1

error_f = np.array(error_f)
total_it = np.array(total_it)
logerr = np.log(error_f)
logit = np.log(total_it)

# Calculating the best fit polynomial for calculating the relation...
# between the log error and number of iterations:
```

```
fit = np.polyfit(logit,logerr,1)
print("best fit line slope = {} \n\n".format(fit[0]))
xsol = xn
# Computing the Spectral radius of the corresponding Jacobian Matrix:
def G(x):
    return np.array([[1/81*ma.sin(x[0]),2/9*x[1],1/3*ma.cos(x[2])],...
    [1/3*ma.cos(x[0]),0,-1/3*ma.sin(x[2])],[1/9*ma.sin(x[0]),...
    1/3,1/6*ma.cos(x[2])]])

#print("\n\nxsol: {}\n\n".format(G(xsol)))
eigvl,eigvec = (np.linalg.eig(G(xsol)))
rho = np.max(np.abs(eigvl))

print("\nC = {}\n\n".format(rho))

# Plotting the error for observation:
plt.figure(1)
plt.semilogy(total_it,error_f);
plt.xlabel('Number of Iterations')
plt.ylabel('Absolute Error in Fixed Point Iteration (log scale)')
plt.title('Fixed Point Iteration (Absolute Error vs Number of Iterations)')
plt.legend(['Absolute Error'],loc = 0,prop={'size':14})
plt.grid(True)


# Solving using the Newton Raphson method:
niter = 1
err_newton = 1
total_itn = ([])
error_n=([])
# Definition of the function to solve for:
def f(x):
    return np.array([-1/81*ma.cos(x[0]) + 1/9 *(x[1])**2 + 1/3*(ma.sin(x[2]))..
    -x[0],(1/3*(ma.sin(x[0])+ma.cos(x[2])))-x[1],-1/9*ma.cos(x[0])+1/3*(x[1])..
    +1/6*(ma.sin(x[2]))-x[2]])


# Definition of the Jacobian:
def J(x):
    return np.array([[1/81*ma.sin(x[0])-1.0,2/9*x[1],1/3*ma.cos(x[2])],...
    [1/3*ma.cos(x[0]),-1.0,-1/3*ma.sin(x[2])],[1/9*ma.sin(x[0]),1/3,...
    1/6*ma.cos(x[2])-1.0]])

# Newton Loop for the iteration:
while err_newton > epsilon :
    if niter ==1:
        xk = np.array([0,0,0])
    else:
        sk = np.linalg.solve(-1.0*J(xk),f(xk))
        xk = xk+sk
```

```
    err_newton = np.linalg.norm(f(xk))
    error_n.append(err_newton)
    total_itn.append(niter)
    niter= niter+1

error_n=np.array(error_n)
total_it=np.array(total_it)

# Plotting the error for observation:
plt.figure(2)
plt.semilogy(total_itn,error_n);
plt.xlabel('Number of Iterations')
plt.ylabel('Absolute Error in Newton\'s method (log scale)')
plt.title('Newton\'s Method (Absolute Error vs Number of Iterations)')
plt.legend(['Absolute Error'],loc = 0,prop={'size':14})
plt.grid(True)
```

5.24

The code used to solve the problem 5.24 is appended below:

```
# -*- coding: utf-8 -*-
"""
Created on Sun Nov 27 17:03:41 2016

@author: bhavesh
"""

import numpy as np
from scipy.optimize import fsolve,root

def f(x):
    w1,w2,x1,x2 = x
    out = [w1+w2-2.0,w1*x1+w2*x2-0.0,w1*x1**2+w2*x2**2-(2.0/3.0),w1*x1**3+w2*x2**3-0
    return out

# Choosing a guess:
guess1 = [1,1,1.0/np.sqrt(3),-1.0/np.sqrt(3)]
guess2 = [1,1,0.0,0.0]
guess3 = [1,1,-1,1]
#guess4 = [0,0,0,0]


# Solving the nonlinear-system:
sol_nl=root(f,guess4)
sol_nl=sol_nl.x
true_sol = np.array([1.0,1.0,-1.0/np.sqrt(3),1.0/np.sqrt(3)])
# Output the solution:
print("Solution:\n\nw1 = {}\nw2 = {}\nx1 = {}\nx2 = {}".format(sol_nl[0],sol_nl[1],so
print('\n\nw1+w2 = {};\nw1*x1+w2*x2 = {};\nw1*x1^2+w2*x2^2 = {}\nw1*x1^3+w2*x2^3 = {}
print("\n\nTrue Solution: {}".format(true_sol))
```

8.6

The code used to solve the problem 8.6 is appended below:

```
# -*- coding: utf-8 -*-
"""
Created on Sun Nov 27 18:49:53 2016

@author: bhavesh
"""

import numpy as np
from scipy.integrate import quad
import matplotlib.pyplot as plt

def f(x,k):
    return np.e**(-1)*x**k*np.e**x

# Comparison of Forward Backward and Adaptive Quadrature:
num = np.array([20,25,30,35,40])
I_sum = ([])
Iquad = ([])
I_ch = ([])

for n in num:
    I = ([])
    for k in range(n+1):
        I.append(quad(f,0,1,(k)))
    I = np.array(I)
    I = I[:,0]
    Iquad.append(I)

# Forward Recurrence Relation:
    If = np.zeros(n+1)
    If[0] = 1-np.e**(-1)
    for k in range(1,n+1):
        If[k] = 1-k*If[k-1]
    I_ch.append(If)

# Backward Recurrence Relation:
    Ik = np.zeros(n+1)
    for m in range(n+1):
        Ik[n-1-m] = (1-Ik[n-m])*1.0/(n-m)
    I_sum.append(Ik)

# Dividing the values into specific arrays:

# For Adaptive Quadrature:
I1 = Iquad[0]
n1 = np.linspace(0,len(I1)-1,len(I1))
```

```
I2 = Iquad[1]
n2 = np.linspace(0,len(I2)-1,len(I2))
I3 = Iquad[2]
n3 = np.linspace(0,len(I3)-1,len(I3))
I4 = Iquad[3]
n4 = np.linspace(0,len(I4)-1,len(I4))
I5 = Iquad[4]
n5 = np.linspace(0,len(I5)-1,len(I5))

#For Forward Recurrence Relation:
I1r = I_ch[0]
n1r = np.linspace(0,len(I1r)-1,len(I1r))
I2r = I_ch[1]
n2r = np.linspace(0,len(I2r)-1,len(I2r))
I3r = I_ch[2]
n3r = np.linspace(0,len(I3r)-1,len(I3r))
I4r = I_ch[3]
n4r = np.linspace(0,len(I4r)-1,len(I4r))
I5r = I_ch[4]
n5r = np.linspace(0,len(I5r)-1,len(I5r ))

# Backward Recurrence Relation:
I1b = I_sum[0]
n1b = np.linspace(0,len(I1b)-1,len(I1b))
I2b = I_sum[1]
n2b = np.linspace(0,len(I2b)-1,len(I2b))
I3b = I_sum[2]
n3b = np.linspace(0,len(I3b)-1,len(I3b))
I4b = I_sum[3]
n4b = np.linspace(0,len(I4b)-1,len(I4b))
I5b = I_sum[4]
n5b = np.linspace(0,len(I5b)-1,len(I5b))


wtest = [2 for i in range(5)]
# Adaptive Quadrature:
plt.figure(0)
plt.plot(n1,I1,n2,I2,n3,I3,n4,I4,n5,I5);
plt.xlabel('k')
plt.ylabel('Integral: I(k)')
plt.title('Adaptive Quadrature')
plt.legend(['k=20','k=25','n=30','n=35','n=40'],loc = 0,prop={'size':14})
plt.grid(True)

# Forward Recurrence Relation:
plt.figure(1)
plt.plot(n1r,I1r,n2r,I2r,n3r,I3r,n4r,I4r,n5r,I5r);
plt.xlabel('k')
plt.ylabel('Integral: I(k)')
plt.title('Forward Recurrence Relation')
```

```
plt.legend(['k=20','k=25','n=30','n=35','n=40'],loc = 0,prop={'size':14})
plt.grid(True)

# Backward Recurrence Relation:
plt.figure(2)
plt.plot(n1b,I1b,n2b,I2b,n3b,I3b,n4b,I4b,n5b,I5b);
plt.xlabel('k')
plt.ylabel('Integral: I(k)')
plt.title('Backward Recurrence Relation')
plt.legend(['k=20','k=25','n=30','n=35','n=40'],loc = 0,prop={'size':14})
plt.grid(True)

# Comparison between the bcakward recurrence and adaptive quadrature:

plt.figure(3)
plt.plot(n5b,I5b,n5,I5);
plt.xlabel('k')
plt.ylabel('Integral: I(k)')
plt.title('Backward Recurrence Relation vs Adaptive Quadrature (n=40)')
plt.legend(['Backward-Recurrence','Adaptive Quadrature'],loc = 0,prop={'size':14})
plt.grid(True)
```

8.10

The code used to solve the problem 8.10 is appended below:

```
# -*- coding: utf-8 -*-
"""
Created on Sun Dec  4 00:29:48 2016

@author: bhavesh
"""

import numpy as np
import scipy as sp
import math as ma
import matplotlib.pyplot as plt
from scipy.integrate import quad
from scipy.integrate import simps

xs = 1.0
deg = 101
xf = 10
dx = 0.1
nx = int(xf/dx) + 1
xnum = np.linspace(xs,xf,nx)
Gam = ([])
lim = 1e5
dt = 0.1
nt = int(lim/dt) + 1
t_tr = np.linspace(0,lim,nt)
```

```
arr = ([])
arr_quad = ([])
arr_quad_tr = ([])
arr_gauss = ([])

# Computing the weights and sample points of the Gauss-Laguerre Quadrature:
sam,wgt = np.polynomial.laguerre.laggauss(deg)


def f(t,x):
    return t**(x-1.0)*np.e**(-1.0*t)


for x in xnum:
#     Composite Quadrature: Simpson's Rule
    arr.append(f(t_tr,x))
    def g(t):
        return f(t,x)
    arr_quad.append(quad(g,0.0,np.inf)[0])
    arr_quad_tr.append(quad(g,0.0,1e2)[0])
    arr_gauss.append(f(sam,x).dot(wgt))

#     Gauss-Laguerre Polynomial:

arr_quad=np.array(arr_quad)
arr_quad_tr=np.array(arr_quad_tr)

# Number of rows and columns:
r = len(arr)
c = len(arr[0])
r_gauss = len(arr_gauss)
#c_gauss = len(arr_gauss[0])

b = np.zeros([r,c])
a = np.zeros(r)
#b_gauss = np.zeros([r_gauss,c_gauss])
err_comp = np.zeros(r)
err_quad = np.zeros(r)
err_quadtr = np.zeros(r)
err_gauss = np.zeros(r_gauss)

for i in range(len(arr)):
    b[i,:] = arr[i]
    a[i] = simps(b[i,:])
    val = ma.gamma(xnum[i])

    err_comp[i] = np.abs(a[i]-val)
    err_quad[i] = np.abs(arr_quad[i]-val)
    err_quadtr[i] = np.abs(arr_quad_tr[i]-val)
    err_gauss[i] = np.abs(arr_gauss[i]-val)
```

```
plt.figure(0)
plt.plot(xnum,err_comp,xnum,err_quad,xnum,err_quadtr,xnum,err_gauss)
plt.xlabel('x',fontsize=18)
plt.ylabel('Error in $\Gamma(x)$',fontsize=18)
plt.legend(['Simpson\'s Rule','Adaptive Quadrature: Truncated','Adaptive Quadrature:
plt.title('Error in $\Gamma(x)$ vs x',fontsize = 22)
plt.grid(True)
```

8.17

The code used to solve the problem 8.17 is appended below:

```
# -*- coding: utf-8 -*-
"""
Created on Sun Dec  4 19:01:01 2016

@author: bhavesh
"""

import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize

N = np.linspace(3,15,7)
for n in N:
    h = 1/(n-1)
    t = np.linspace(0,1,n)
    s = t
    S,T = np.meshgrid(s,t)

    w1 = np.ones(int(n))
    w1 [range(1,int(n)-1,2)]= 4
    w1 [range(2,int(n)-2,2)]= 2
    w1 = w1*h/3
    A = (S**2 + T**2)**0.5
    for j in range(len(A[1,:])):
        A [:,j] = A [:,j] * w1[j]

    plt.figure(1)
    plt.semilogy(n,np.linalg.cond(A),'o')
    plt.xlabel('Size of the Matrix (n)')
    plt.ylabel('Condition Number of the Matrix (c(n))')
    plt.title('Condition Number vs Matrix Size : n = 3 - 15')

#      Gaussian-Elimination with partial pivoting :

    f = ((s**(2)+1)**(3/2) - s**3)/3
    xgauss = (np.linalg.solve(A,f))
    plt.figure(2)
    plt.plot(t,xgauss)
```

```
plt.title('Gaussian-Elimination : Partial Pivoting')
plt.xlabel('Size of the Matrix (n)')
plt.ylabel('Solution by Gaussian Elimination : Partial Pivoting')

plt.figure(3)
U,S,V = np.linalg.svd(A)
l = 0
x2 = np.zeros(int(n))
Areq = 1000

for l in range(len(S)):
    if S[1]/S[l] <= Areq:
        x2 = x2 + (np.transpose(U[:,l]).dot(f)/S[l])*V[l,:]

plt.figure(4)
plt.plot(t,x2)
plt.title('SVD')
plt.xlabel('Size of the Matrix (n)')
plt.ylabel('Solution by SVD')

ns = np.zeros(20)
nr = np.zeros(20)
j = 0

for mu in np.linspace(0.1,1,20):
    A2 = np.concatenate((A,(mu**0.5)*np.identity(int(n))),axis = 0)
    y2 = np.concatenate((f,np.zeros(int(n))),axis = 0)
    x3 = np.linalg.lstsq(A2, y2)[0]
    ns[j] = np.linalg.norm(x3)
    nr[j] = np.linalg.norm(y2-A2.dot(x3))
    j +=1
    plt.figure(5)
    plt.plot(t,x3)
    plt.title('Regularization')
    plt.xlabel('Size of the Matrix (n)')
    plt.ylabel('Solution by Regularization')
plt.figure(6)
plt.plot(ns,nr)
plt.title('Regularization')
plt.xlabel('Norm of the Solution')
plt.ylabel('Norm of the Residual')

xo = np.ones(int(n))

def Res(q):
    return(np.linalg.norm(f-A.dot(q)))
bnds = tuple((0,None) for x in xo)
x4 = minimize(Res,xo, method='SLSQP', bounds=bnds).x

plt.figure(7)
```

```
plt.plot(t,x4)
plt.title('Optimization')
plt.xlabel('Size of the Matrix (n)')
plt.ylabel('Solution : u(x,t)')

xo = np.ones(int(n))
cons= ({'type': 'ineq', 'fun': lambda x: x[1] - x[0]},
       {'type': 'ineq', 'fun': lambda x: x[2] - x[1]},
       {'type': 'ineq', 'fun': lambda x: x[3] - x[2]},
       {'type': 'ineq', 'fun': lambda x: x[4] - x[3]},
       {'type': 'ineq', 'fun': lambda x: x[5] - x[4]},
       {'type': 'ineq', 'fun': lambda x: x[6] - x[5]},
       {'type': 'ineq', 'fun': lambda x: x[7] - x[6]},
       {'type': 'ineq', 'fun': lambda x: x[8] - x[7]},
       {'type': 'ineq', 'fun': lambda x: x[9] - x[8]},
       {'type': 'ineq', 'fun': lambda x: x[10] - x[9]},
       {'type': 'ineq', 'fun': lambda x: x[11] - x[10]},
       {'type': 'ineq', 'fun': lambda x: x[12] - x[11]},
       {'type': 'ineq', 'fun': lambda x: x[13] - x[12]},
       {'type': 'ineq', 'fun': lambda x: x[14] - x[13]})
consS = cons[0:int(n-1)]
x5 = minimize(Res,xo, method='SLSQP', bounds=bnds,constraints= consS).x

plt.figure(8)
plt.plot(t,x5)
plt.title('Constrained - Optimization')
plt.xlabel('Size of the Matrix (n)')
plt.ylabel('Solution : u(x,t)')
```

8.18

The code used to solve the problem 8.18 is appended below:

```
# -*- coding: utf-8 -*-
"""
Created on Sun Dec  4 19:01:01 2016

@author: bhavesh
"""

import numpy as np
from scipy.interpolate import UnivariateSpline
import matplotlib.pyplot as plt
#input data
t = np.array([0.0,1.0,2.0,3.0,4.0,5.0])
y = np.array([1.0,2.7,5.8,6.6,7.5,9.9])
#Perturbing y
y = (np.random.randint(low=-1, high=2, size=len(y))*5/100 +1)*y
#Print Polynomial fitting
print('Polynomil')
for n in range(6):
```

```
    print('n =', n)
    x = np.polyfit(t,y,n)
    x = np.flipud(x)
    print ('x =', x)
    T = np.linspace(0,5.0,100)
    x_val = np.zeros(len(T))
    for z in range(len(T)):
        for l in range(n+1):
            x_val[z] += x[l]*T[z]**l
    dx = np.zeros(len(x)-1)
    for i in range (len(x)-1):
        dx[i]=x[i+1]*(i+1)
    print('dx =', dx)
    dx_val = np.zeros(len(t))
    for z in range(len(t)):
        for l in range(n):
            dx_val[z] += dx[l]*t[z]**l
    print('dx_val =', dx_val)
    plt.figure(n)
    plt.plot(t,y,'ok', label = 'Data Points')
    plt.plot(T,x_val, label = 'Polynomial of degree %s'%n)
    plt.title('Least Square Polynomial Fit, Degree%s'%n)
    plt.xlabel('t')
    plt.ylabel('y')
    plt.grid()
    plt.legend()
#Cubic spline
print('Cubic Spline')
SP_coeffs = UnivariateSpline(t, y,s=0.0001)
plt.figure()
plt.plot(t,y,'ok', label = 'Data Points')
plt.plot(T,SP_coeffs(T),label = 'Cubic Spline')
plt.title('Cubic Spline')
plt.xlabel('t')
plt.ylabel('y')
plt.grid()
plt.legend()
for z in range(len(t)):
    dx_val[z] = SP_coeffs.derivatives(t[z])[1]
print('dx_val =', dx_val)
#smoothing spline
print('Smoothing Spline')
K = np.array([1,2,4,5])
for k1 in K:
    print('k =', k1)
    SP_coeffs = interpolate.UnivariateSpline(t,y,k=k1,s=0.0001)
    for z in range(len(t)):
        dx_val[z] = SP_coeffs.derivatives(t[z])[1]
    print('dx_val =', dx_val)
    plt.figure()
```

```
    plt.plot(t,y,'ok', label = 'Data Points')
    plt.plot(T, SP_coeffs(T), label = 'Smoothing Spline , k = %s '%k1)
    plt.title('Smoothing Spline , k = %s '%k1)
    plt.xlabel('t')
    plt.ylabel('y')
    plt.grid()
    plt.legend()
#Hermit spline
print('Hermit Spline')
Her_coeffs = interpolate.PchipInterpolator(t,y)
Her_der = Her_coeffs.derivative(nu=1)
print('dx_val =', Her_der(t))
plt.figure()
plt.plot(t,y,'ok', label = 'Data Points')
plt.plot(T, Her_coeffs(T), label = 'Hermit Spline')
plt.title('Hermit Spline')
plt.xlabel('t')
plt.ylabel('y')
plt.grid()
plt.legend()
```