Assignment-1

Git: <https://github.com/iamankan/MA421G.git>

Branch: assignments

P4:

1. Newton’s Method

Code:

|  |
| --- |
| import numpy as np  # implementing Newton's iteration to minimize f(x)  eps\_const = 1e-6  # Equation: 2x1\*\*2 + x2\*\*2 - 2x1\*x2 + 2x1\*\*3 + x1\*\*4  def calc\_df(x1: float, x2: float) -> np.ndarray:  df = np.array([[(4\*x1)-(2\*x2)+(6\*pow(x1,2))+(4\*pow(x1,3))],[(2\*x2)-(2\*x1)]])  return df  def calc\_hessian(x1: float, x2: float) -> np.ndarray:  hess = np.array([[(12\*pow(x1,2)+(12\*x1)+4),-2],[-2,2]])  return hess  def calc\_prereq(x0: np.ndarray) -> tuple:  del\_f = calc\_df(x0[0][0], x0[1][0])  H\_f = calc\_hessian(x0[0][0], x0[1][0])  return del\_f, H\_f  def newton(x0: np.ndarray, del\_f: np.ndarray, H\_f: np.ndarray, eps: float = 1e-6, max\_iter: int=100):  x = [x0]  for k in range(max\_iter):  norm\_2 = np.linalg.norm(x=del\_f, ord=2)  if norm\_2 < eps:  return x, k, x[-1]  d = np.linalg.solve(H\_f, del\_f)  x.append(x[k] - d)  del\_f = calc\_df(x[k+1][0][0], x[k+1][1][0])  H\_f = calc\_hessian(x[k+1][0][0], x[k+1][1][0])  print(f'k:{k}')  return x, k, x[-1]  def newton\_pipeline(x0: np.ndarray, eps: float = 1e-6, max\_iter: int=100):  del\_f, H\_f = calc\_prereq(x0)  return newton(x0, del\_f, H\_f)  x0=[np.array([[1],[1]]), np.array([[1],[-1]]), np.array([[2],[-2]])]  newton\_ans = {}  cnt = 0  for xi in x0:  newton\_output = newton\_pipeline(x0=xi)  newton\_ans[cnt] = {  'x0': xi,  'newton': newton\_output,  'iteration': newton\_output[1],  'optimal value': newton\_output[2]  }  cnt = cnt + 1  print(newton\_ans)  # Checking quadratic convergence  lhs= {}  rhs= {}  for idx in newton\_ans:  len1 = newton\_ans[idx]['iteration']  opt\_x = newton\_ans[idx]['optimal value']  newton\_all = newton\_ans[idx]['newton'][0]  lhs[idx] = {  'x0': newton\_ans[idx]['x0'],  'error': []  }  rhs[idx] = {  'x0': newton\_ans[idx]['x0'],  'error': []  }  for i in range(len1-1):  lhs[idx]['error'].append(np.linalg.norm(newton\_all[i+1] - opt\_x, ord=2))  rhs[idx]['error'].append(np.linalg.norm(newton\_all[i] - opt\_x, ord=2)\*\*2)  import matplotlib.pyplot as plt  for idx in lhs:  plt.title(f'{lhs[idx]["x0"]}')  lhs\_err = lhs[idx]['error']  rhs\_err = rhs[idx]['error']  plt.plot(range(len(lhs\_err)), lhs\_err, label='present error')  plt.plot(range(len(rhs\_err)), rhs\_err, label='previous error')  plt.legend()  plt.show() |

Output:

|  |
| --- |
| {0: {'x0': array([[1],  [1]]),  'newton': ([array([[1],  [1]]),  array([[0.53846154],  [0.53846154]]),  array([[0.25028589],  [0.25028589]]),  array([[0.08710246],  [0.08710246]]),  array([[0.01620003],  [0.01620003]]),  array([[0.00073202],  [0.00073202]]),  array([[1.60210289e-06],  [1.60210289e-06]]),  array([[7.70014343e-12],  [7.70014343e-12]])],  7,  array([[7.70014343e-12],  [7.70014343e-12]])),  'iteration': 7,  'optimal value': array([[7.70014343e-12],  [7.70014343e-12]])},  1: {'x0': array([[ 1],  [-1]]),  'newton': ([array([[ 1],  [-1]]),  array([[0.53846154],  [0.53846154]]),  array([[0.25028589],  [0.25028589]]),  array([[0.08710246],  [0.08710246]]),  array([[0.01620003],  [0.01620003]]),  array([[0.00073202],  [0.00073202]]),  array([[1.60210289e-06],  [1.60210289e-06]]),  array([[7.70014343e-12],  [7.70014343e-12]])],  7,  array([[7.70014343e-12],  [7.70014343e-12]])),  'iteration': 7,  'optimal value': array([[7.70014343e-12],  [7.70014343e-12]])},  2: {'x0': array([[ 2],  [-2]]),  'newton': ([array([[ 2],  [-2]]),  array([[1.18918919],  [1.18918919]]),  array([[0.66000444],  [0.66000444]]),  array([[0.32439061],  [0.32439061]]),  array([[0.12640158],  [0.12640158]]),  array([[0.03020608],  [0.03020608]]),  array([[0.00239946],  [0.00239946]]),  array([[1.70809427e-05],  [1.70809427e-05]]),  array([[8.7520605e-10],  [8.7520605e-10]])],  8,  array([[8.7520605e-10],  [8.7520605e-10]])),  'iteration': 8,  'optimal value': array([[8.7520605e-10],  [8.7520605e-10]])}} |

Quadratic Convergence property:

Chart, line chart

Description automatically generated

Chart, line chart

Description automatically generated

Chart, line chart

Description automatically generated

1. Gradient Descent

Code:

|  |
| --- |
| import numpy as np  # Equation: 2x1\*\*2 + x2\*\*2 - 2x1\*x2 + 2x1\*\*3 + x1\*\*4  def f(x1, x2):  return (2\*x1\*\*2) + (x2\*\*2) - (2\*x1\*x2) + (2\*x1\*\*3) + (x1\*\*4)  def calc\_df(x1: float, x2: float) -> np.ndarray:  df = np.array([[(4\*x1)-(2\*x2)+(6\*pow(x1,2))+(4\*pow(x1,3))],[(2\*x2)-(2\*x1)]])  return df  def calc\_hessian(x1: float, x2: float) -> np.ndarray:  hess = np.array([[(12\*pow(x1,2)+(12\*x1)+4),-2],[-2,2]])  return hess  # checking alpha  def calc\_alpha\_cond(hess: np.ndarray) -> float:  norm\_2 = np.linalg.norm(hess, ord=2)  return 2/norm\_2  def alpha\_check(x0: np.ndarray, alpha: float) -> tuple:  x0=np.array([[2],[-2]])  hess = calc\_hessian(x0[0][0], x0[1][0])  alpha\_cond = calc\_alpha\_cond(hess)  return alpha < alpha\_cond, alpha\_cond  def GD(x0: np.ndarray, dfx: np.ndarray, alpha: float = 1, eps: float = 10e-6, max\_iter: int = 100) -> tuple:  xlist = [x0]  alpha\_cond, alphanorm = alpha\_check(x0, alpha)  if alpha\_cond == True:  for k in range(max\_iter):  print(f'x{k}= {xlist[k]}, f(x{k})= {f(xlist[k][0][0], xlist[k][1][0])}, 2-norm= {np.linalg.norm(x=dfx, ord=2)}')  if np.linalg.norm(x=dfx, ord=2) < eps:  return (xlist, k, xlist[-1], 'Early stop.', alphanorm)  xlist.append(xlist[k]-(alpha\*dfx))  dfx = calc\_df(xlist[k+1][0][0], xlist[k+1][1][0])  return (xlist, k, xlist[-1], 'All iterations.', alphanorm)  else:  print(f'alpha= {alpha} cannot guarantee convergence. So, quiting!')  return (xlist, 0, xlist[-1], 'Bad alpha!', alphanorm)  x0set = [np.array([[1],[1]]), np.array([[1],[-1]]), np.array([[2],[-2]])]  alphaset = [1, 1e-1, 1e-2,1e-3]  gdout={}  cnt = 0  for xi in x0set:  for alpha in alphaset:  dfx = calc\_df(xi[0][0], xi[1][0])  print(alpha)  # print(dfx)  gdans = GD(xi, dfx, alpha=alpha)  gdout[cnt]={  'x0': xi,  'alpha': alpha,  'dfx': dfx,  'gdoutput': gdans,  'optimal': {  'x\_k+1': gdans[2],  'iters': gdans[1],  'status': gdans[3],  'alpha\_condition': gdans[4]  }  }  cnt=cnt+1    for gdidx in gdout:  gd\_dict = gdout[gdidx]  print(f'x0: {gd\_dict["x0"]}\n alpha: {gd\_dict["alpha"]}\n optimal solutions: {gd\_dict["optimal"]}')  print('=================================')  for xi in x0set:  print(f'x0: {xi}')  for alpha in alphaset:  print(f'{alpha}: {alpha\_check(x0= xi, alpha=alpha)}') |

Output:

|  |
| --- |
| x0: [[1]  [1]]  alpha: 1  optimal solutions: {'x\_k+1': array([[1],  [1]]), 'iters': 0, 'status': 'Bad alpha!', 'alpha\_condition': 0.026297099631110685}  =================================  x0: [[1]  [1]]  alpha: 0.1  optimal solutions: {'x\_k+1': array([[1],  [1]]), 'iters': 0, 'status': 'Bad alpha!', 'alpha\_condition': 0.026297099631110685}  =================================  x0: [[1]  [1]]  alpha: 0.01  optimal solutions: {'x\_k+1': array([[0.16005716],  [0.33833177]]), 'iters': 99, 'status': 'All iterations.', 'alpha\_condition': 0.026297099631110685}  =================================  x0: [[1]  [1]]  alpha: 0.001  optimal solutions: {'x\_k+1': array([[0.49542474],  [0.94016202]]), 'iters': 99, 'status': 'All iterations.', 'alpha\_condition': 0.026297099631110685}  =================================  x0: [[ 1]  [-1]]  alpha: 1  optimal solutions: {'x\_k+1': array([[ 1],  [-1]]), 'iters': 0, 'status': 'Bad alpha!', 'alpha\_condition': 0.026297099631110685}  =================================  x0: [[ 1]  [-1]]  alpha: 0.1  optimal solutions: {'x\_k+1': array([[ 1],  [-1]]), 'iters': 0, 'status': 'Bad alpha!', 'alpha\_condition': 0.026297099631110685}  =================================  x0: [[ 1]  [-1]]  alpha: 0.01  optimal solutions: {'x\_k+1': array([[-0.15741484],  [-0.22310316]]), 'iters': 99, 'status': 'All iterations.', 'alpha\_condition': 0.026297099631110685}  =================================  x0: [[ 1]  [-1]]  alpha: 0.001  optimal solutions: {'x\_k+1': array([[ 0.29728532],  [-0.71878223]]), 'iters': 99, 'status': 'All iterations.', 'alpha\_condition': 0.026297099631110685}  =================================  x0: [[ 2]  [-2]]  alpha: 1  optimal solutions: {'x\_k+1': array([[ 2],  [-2]]), 'iters': 0, 'status': 'Bad alpha!', 'alpha\_condition': 0.026297099631110685}  =================================  x0: [[ 2]  [-2]]  alpha: 0.1  optimal solutions: {'x\_k+1': array([[ 2],  [-2]]), 'iters': 0, 'status': 'Bad alpha!', 'alpha\_condition': 0.026297099631110685}  =================================  x0: [[ 2]  [-2]]  alpha: 0.01  optimal solutions: {'x\_k+1': array([[-0.59545138],  [-0.63088539]]), 'iters': 99, 'status': 'All iterations.', 'alpha\_condition': 0.026297099631110685}  =================================  x0: [[ 2]  [-2]]  alpha: 0.001  optimal solutions: {'x\_k+1': array([[ 0.3410173 ],  [-1.48868318]]), 'iters': 99, 'status': 'All iterations.', 'alpha\_condition': 0.026297099631110685}  ================================= |

Status means whether it used all the iterations or stopped early.

Alpha condition is checking whether alpha is < 2/norm(Hessian,2).