Problem 1. *Write a program to solve the discrete Poisson equation on the unit square using preconditioned conjugate gradient. Set up a test problem and compare the number of iterations and efficiency of using (i) no preconditioning, (ii) SSOR preconditioning, (iii) multigrid preconditioning. Run your tests for different grid sizes. How does the number of iterations scale with the number of unknowns as the grid is refined? For multigrid preconditioning, compare the efficiency of with multigrid as a standalone solver.*

I used my PCG algorithm to solve the Poisson equation

$$\Delta u = -\exp\big(-(x-0.25)^2-(y-0.6)^2\big) = f$$

on $(0,1)\times(0,1)$ with Dirichlet boundary conditions.

With the stopping criterion $\|r\|_\infty \le 10^{-7}\cdot\|f\|_\infty$, we have that the pre-conditioned conjugate gradient descent method (PCG) vastly outperforms the standard conjugate gradient descent method (CG), and within pre-conditioning methods multigrid outperforms SSOR. We find that CG with no pre-conditioning doubles in iteration count as the grid scales up, i.e., as the number of unknowns quadruples. On the other hand, PCG with SSOR only scales one and half times in iteration count as the number of unkowns quadruples, and PCG with MG (with GS RB-BR smoothing) remains relatively grid-independent in iteration count. Notably, PCG with MG outperforms standalone MG (both with GS RB-BR smoothing). The results are tabulated as follows:

Table 1: Iteration count comparison

method	$h=2^{-4}$	2^{-5}	2^{-6}	2^{-7}	2^{-8}	ratio
CG	42	88	181	369	747	2
PCG - SSOR	16	24	36	54	81	1.5
PCG - MG	6	6	7	7	7	1
MG - GSRBBR	11	11	11	11	11	1

Table 2: CG with no pre-conditioning

grid spacing h	iteration count	run time (seconds)
2^{-4}	42	0.007836
2^{-5}	88	0.013646
2^{-6}	181	0.055342
2^{-7}	369	0.287279
2^{-8}	747	2.06416

Table 3: PCG with SSOR

grid spacing h	iteration count	run time (seconds)
2^{-4}	16	0.074605
2^{-5}	24	0.433971
2^{-6}	36	2.63342
2^{-7}	54	15.9162
2^{-8}	81	97.6399

Table 4: PCG with MG (with GS RB-BR smoothing)

grid spacing h	iteration count	run time (seconds)
2^{-4}	6	0.060207
2^{-5}	6	0.25899
2^{-6}	7	1.21256
2^{-7}	7	4.97453
2^{-8}	7	20.1019

Table 5: Multigrid V-cycle Stand-alone with GS-RBBR Smoothing

grid spacing h	iteration count	run time (seconds)
2^{-4}	11	0.093634
2^{-5}	11	0.41561
2^{-6}	11	1.6579
2^{-7}	11	6.5331
2^{-8}	11	26.6621

The code used for problem 1 includes all the functions from last homework, which I don't recopy here, and the new functions:

```
#MAT228A HW 5
#Number 1

from argparse import ArgumentParser
import numpy as np
from numpy import sin, pi, exp
from conjugate_gradient_descent import conjugate_gradient_descent as cgd
from make_matrix_rhs_circleproblem import make_matrix_rhs_circleproblem as mmrc
from compute_residual import get_Laplacian, apply_Laplacian
import tabulate
from time import clock

def PARSE_ARGS():
  parser = ArgumentParser()
  parser.add_argument("-p", "--power", type=int, default=7, dest="power")
  parser.add_argument("-s", "--shampoo", type=int, default=1, dest="shampoo")
  parser.add_argument("-m", "--maxits", type=int, default=2000, dest="max_iterations")
  parser.add_argument("-t", "--tol", type=float, default=1e-7, dest="tol")
  return parser.parse_args()

def RHS(h):
  n = int(1/h)-1
  u = np.zeros((n+2,n+2))

  for i in range(1,n+1):
```

```
    for j in range(1,n+1):
      # u[i][j] = sin(pi*i*h)*sin(pi*j*h)
      u[i][j] = -exp(-(i*h-0.25)**2 - (j*h-0.6)**2)
  return u

def main():
  ARGS = PARSE_ARGS()

  #make Laplacians for V cycle
  Laplacians = []
  for i in range(2,10):
    Laplacians.append(get_Laplacian(2**(-i)))

  grid_spacings = [2**(-4), 2**(-5), 2**(-6), 2**(-7), 2**(-8)]
  itcounts = []
  times = []

  for h in grid_spacings:
    print "h = ", h
    #compute grid point number n and power
    n = int(1/h) -1
    power = -np.log2(h)

    #compute RHS to f=Au
    A = Laplacians[int(-2-np.log2(h))]
    f = RHS(h)

    #use CGD or PCG algorithm
    toc = clock()
    [u,iterations] = cgd(power, ARGS.shampoo, ARGS.tol, ARGS.max_iterations, A, Laplacians,
    tic = clock()
    times.append(tic-toc)
    itcounts.append(iterations)

  test_table = [[grid_spacings[i], itcounts[i], times[i]] for i in range(np.size(grid_spacin
  if ARGS.shampoo == 1:
    print "CG with no pre-conditioning"
  if ARGS.shampoo == 2:
    print "PCG with SSOR"
  if ARGS.shampoo == 3:
    print "PCG with MG"
  print tabulate.tabulate(test_table, headers = ["grid spacing h", "iteration count", "run t

def MG_preconditioner(r, h, Laplacians):
  n = int(1/h)-1
  z = np.zeros((n+2,n+2))
```

```
  z = V_cycle(z,r,h,1,1,Laplacians)
  # print z
  return z

def SSOR_preconditioner(r, h):
  n = int(1/h)-1
  z = np.zeros((n+2,n+2))

  w = 2/(1+sin(pi*h))

  for i in range(1,n+1):
    for j in range(1,n+1):
      z[i][j] = (w/4)*(z[i-1][j] + z[i][j-1] + z[i+1][j] + z[i][j+1] - (h**2)*r[i][j]) + (1-

  for i in range(n,0,-1):
    for j in range(n,0,-1):
      z[i][j] = (w/4)*(z[i-1][j] + z[i][j-1] + z[i+1][j] + z[i][j+1] - (h**2)*r[i][j]) + (1-

  return z

def conjugate_gradient_descent(power, shampoo, tol, max_iterations, A, Laplacians, f):
  #set grid size h and grid points N
  h = 2**(-power)
  n = int(1/h) -1

  #initial guess
  u = np.zeros((n+2,n+2))

  #initialize residual
  r = f - apply_Laplacian(u,h,A)

  tic = clock()
  #apply pre-conditioner/ shampoo
  if shampoo == 1:
    z = copy.deepcopy(r)
  if shampoo == 2:
    z = SSOR_preconditioner(r, h)
  if shampoo == 3:
    z = MG_preconditioner(r,h, Laplacians)
  if shampoo == 4:
    z = exact_solve_preconditioner(r,h,A)

  #initial search direction (residual/negative gradient)
  p = copy.deepcopy(z)
```

```
  #loop until solved
  t = tqdm(xrange(max_iterations))
  for k in t:
    w = apply_Laplacian(p, h, A)

    alpha_num = np.dot(z.flatten(),r.flatten())
    alpha_denom = np.dot(p.flatten(),w.flatten())
    alpha =  alpha_num/alpha_denom
    u = u+ alpha*p

    r_new = r - alpha*w

    if np.amax(np.abs(r_new)) <= tol*np.amax(np.abs(f)):
      print "iteration count =", k+1
      print "||r|| =", np.amax(np.abs(r_new))
      toc = clock()
      print "runtime = ", toc-tic
      break;

    #apply pre-conditioner
    if shampoo == 1:
      z_new = copy.deepcopy(r_new)
    if shampoo == 2:
      z_new = SSOR_preconditioner(r_new, h)
    if shampoo == 3:
      z_new = MG_preconditioner(r_new, h, Laplacians)
    if shampoo == 4:
      z_new = exact_solve_preconditioner(r_new,h,A)

    beta = np.dot(z_new.flatten(), r_new.flatten()) / np.dot(z.flatten(),r.flatten())

    p = z_new + beta*p

    r = copy.deepcopy(r_new)
    normr = np.amax(np.abs(r))
    t.set_description("||res||=%.10f"%normr)
    z = copy.deepcopy(z_new)

  if k == max_iterations-1:
    print "max iterations exceeded"

  return u, k+1


if __name__ == "__main__":
  main()
```

Problem 2. *Use the provided code that gives a matrix and right hand side for a discretized Poisson equation on a domain which is the intersection of the interior of the unit square and exterior of a circle centered at (0.3, 0.4) with radius 0.15. The boundary conditions are zero on the square and 1 on the circle. Use your preconditioned conjugate gradient code to solve this problem. Explore the performance of no preconditioning and multigrid preconditioning for different grid sizes. Comment on your results. Note that the MG code is based on an MG solver for a different domain, and so it cannot be used as a solver for this problem. Is it an effective preconditioner?*

Translating the provided code into python and then solving the Poisson equation with the given RHS and domain, we found that the PCG code works wonders on speeding up the CG. The CG (with no-preconditioning) performs slightly worse than in the regular domain problem, going up in iteration count by an average factor of 2.2 as the number of unknowns quadruples. The pre-conditioned CG performs much better than standard CG, but slightly worse than in the previous problem. For PCG with SSOR, the iteration count goes up by an average factor of 1.55 as the number of unknowns quadruples (compared to 1.5 in the last problem), and for PCG with MG (with GS RB-BR smoothing) the iteration count goes up by an average factor of 1.45 as the number of unknowns quadruples (compared to 1 in the last problem). This isn't quite as optimal as the performance in the last problem, but it is still a significant reduction (compared to unconditioned CG) in the number of iterations for fine grids, so MG is an effective preconditioner. Although the run times of my code may make it seem as if the MG pre-conditioning isn't worth the effort, with a more optimized code the vastly smaller iteration count would translate into much more competitive speeds. These results are tabulated as follows:

Table 6: Iteration count comparison

method	$h = 2^{-4}$	2^{-5}	2^{-6}	2^{-7}	2^{-8}	avg. ratio
CG	43	95	185	518	1151	2.2
PCG - SSOR	19	31	45	75	125	1.55
PCG - MG	16	23	31	49	74	1.45

Table 7: CG with no pre-conditioning

grid spacing h	iteration count	run time (seconds)
2^{-4}	43	0.007709
2^{-5}	95	0.022307
2^{-6}	185	0.051272
2^{-7}	518	0.389691
2^{-8}	1151	3.34225

Table 8: PCG with SSOR

grid spacing h	iteration count	run time (seconds)
2^{-4}	19	0.090087
2^{-5}	31	0.565604
2^{-6}	45	3.29774
2^{-7}	75	22.5772
2^{-8}	125	152.952

Table 9: PCG with MG (with GS RB-BR smoothing)

grid spacing h	iteration count	run time (seconds)
2^{-4}	16	0.163576
2^{-5}	23	0.972086
2^{-6}	31	5.44926
2^{-7}	49	34.8369
2^{-8}	74	216.899

The code used for problem 2 is as follows:

```
#MAT228A HW 5
#Number 2
#Unique Domain Poisson Solver Using CG

from argparse import ArgumentParser
import numpy as np
from time import clock
import tabulate

from conjugate_gradient_descent import conjugate_gradient_descent as cgd
from make_matrix_rhs_circleproblem import make_matrix_rhs_circleproblem as mmrc
from compute_residual import get_Laplacian


def PARSE_ARGS():
  parser = ArgumentParser()
  parser.add_argument("-p", "--power", type=int, default=7, dest="power")
  parser.add_argument("-s", "--shampoo", type=int, default=1, dest="shampoo")
  parser.add_argument("-m", "--maxits", type=int, default=2000, dest="max_iterations")
  parser.add_argument("-t", "--tol", type=float, default=1e-7, dest="tol")
  return parser.parse_args()

def main():
  ARGS = PARSE_ARGS()

  #make Laplacians for V cycle
```

```
  Laplacians = []
  for i in range(2,10):
    Laplacians.append(get_Laplacian(2**(-i)))

  grid_spacings = [2**(-4), 2**(-5), 2**(-6), 2**(-7), 2**(-8)]
  itcounts = []
  times = []

  for h in grid_spacings:
    print "h = ", h
    #compute grid point number n and power
    n = int(1/h) -1
    power = -np.log2(h)

    #compute RHS to f=Au on special domain
    [A, f, phi] = mmrc(h)
    f = f.reshape(n, n)
    f = np.pad(f, ((1,1),(1,1)), mode='constant')

    #use CGD or PCG algorithm
    toc = clock()
    [u,iterations] = cgd(power, ARGS.shampoo, ARGS.tol, ARGS.max_iterations, A, Laplacians,
    tic = clock()
    times.append(tic-toc)
    itcounts.append(iterations)

  test_table = [[grid_spacings[i], itcounts[i], times[i]] for i in range(np.size(grid_spacin
  if ARGS.shampoo == 1:
    print "CG with no pre-conditioning"
  if ARGS.shampoo == 2:
    print "PCG with SSOR"
  if ARGS.shampoo == 3:
    print "PCG with MG"
  print tabulate.tabulate(test_table, headers = ["grid spacing h", "iteration count", "run t

def make_matrix_rhs_circleproblem(h):

  #initialize A as the standard Laplacian (scaled)
  n = int(1/h)-1
  # print "n = ", n
  A = (h**2)*get_Laplacian(h)

  #initialize f to be zeros
  f = np.zeros(n*n)

  #assume boundary value constant
```

```
  Ub = 1

  #form arrays of grid point locations
  x = [i*h for i in range(1,n+1)]
  x,y = np.meshgrid(x,x)

  #parameters for the embedded circle, center coordinates and radius
  xc = 0.3
  yc = 0.4
  rad = 0.15

  #compute the signed distance function
  phi = sqrt( (x-xc)**2 + (y-yc)**2 ) - rad
  IJ = [[-1, 0], [1, 0], [0, -1], [0,1]]

  for j in range(1,n-1):
    for i in range(1,n-1):

      #skip the interior points
      if phi[i][j] < 0:
        continue

      for k in range(4):
        if phi[i+IJ[k][0]][j+IJ[k][1]]  < 0:
          #approximate distance to the boundary, scaled by h
          alpha = phi[i][j]/(phi[i][j] - phi[ i+IJ[k][0] ][ j+IJ[k][1] ])

          #compute the distance to the boundary
          kr = np.ravel_multi_index([i,j], (n,n) )
          kc = np.ravel_multi_index([i+IJ[k][0], j+IJ[k][1]], (n,n))


          #adjust RHS
          f[kr] = f[kr] - Ub/alpha

          #adjust diagonal element
          A[kr,kr] = A[kr,kr] + 1 - 1/alpha

          #adjust off-diagonal and enforce symmetry
          A[kr,kc] = 0
          A[kc,kr] = 0

  return A, f, phi

if __name__ == "__main__":
  main()
```