

┌ **Problem 1.** Write a multigrid V-cycle code to solve the Poisson equation in two dimensions on the unit square with Dirichlet boundary conditions. Use full weighting for restriction, bilinear interpolation for prolongation, and red-black Gauss-Seidel for smoothing. Use this code to solve

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

on the unit square $(0, 1) \times (0, 1)$ with homogeneous Dirichlet boundary conditions for different grid spacings. How many steps of pre and postsmoothing did you use? What tolerance did you use? How many cycles did it take to converge? Compare the amount of work needed to reach convergence with your solvers from Homework 3 taking into account how much work is involved in a V-cycle.

└

For the given problem, I ran my multigrid V-cycle code with the stopping criterion max-norm of the residual relative to the max-norm of the RHS function, with a tolerance of 10^{-10} :

$$\|r^k\|_\infty \leq \text{tol} \cdot \|f\|_\infty.$$

This is a good stopping criterion since the error is bounded by the norm of the residual times the matrix norm of the inverse discrete Laplacian:

$$\|e^k\| \leq \|A^{-1}\| \|r^k\|.$$

Using my analysis from 2, I found that (1,1), (2,1), and (2,2) were top contenders for optimal pre-smoothing, post-smoothing step numbers, and the fastest results were obtained for 2 pre-smoothing steps and 1 post-smoothing step. The results show that iteration count for the multigrid algorithm is grid-independent. The choice of $(\nu_1, \nu_2) = (1, 1)$ performs slightly better than (2, 1) and (2, 2) on all grid sizes, so it is the optimal choice. The results are tabulated as follows:

For $(\nu_1, \nu_2) = (1, 1)$,

grid spacing h	iteration count	run time (seconds)
2^{-5}	12	0.296257
2^{-6}	12	1.23618
2^{-7}	12	4.7184
2^{-8}	12	18.9813
2^{-9}	12	77.6296

For $(\nu_1, \nu_2) = (2, 1)$,

grid spacing h	iteration count	run time (seconds)
2^{-5}	10	0.303806
2^{-6}	10	1.24774
2^{-7}	10	4.95247
2^{-8}	10	19.4882
2^{-9}	10	78.2237

For $(\nu_1, \nu_2) = (2, 2)$,

grid spacing h	iteration count	run time (seconds)
2^{-5}	9	0.331599
2^{-6}	9	1.37659
2^{-7}	9	5.52204
2^{-8}	9	21.5597
2^{-9}	9	88.6846

Compared to the solvers from homework 3, i.e., the Jacobi, GS-lex, and SOR iterative methods, this program takes significantly fewer total iterations, and considerably less work overall. Each iteration of those methods is the same amount of work as a single smoothing operation in the multigrid V-cycle, call this amount of work W . In each V-cycle iteration of the multigrid algorithm, we do ν smoothing operations on the finest mesh. We also perform a single residual computation, restriction operation, interpolation operation, and correction (matrix addition) each of which is comparable work to a smoothing operation, so the total work on the finest mesh is thus $(\nu + 4)W$. As we discussed in class, every coarser mesh level requires a factor of 4 less work, so the total work of all the smooths in the limit of many mesh levels is

$$\sum_{\ell=1}^L (\nu + 3)W \left(\frac{1}{4}\right)^{\ell-1} \approx (\nu + 3)W \left(\frac{1}{1 - \frac{1}{4}}\right) = \frac{4}{3}(\nu + 3)W.$$

So for a 64×64 mesh ($h = 2^{-6}$) and tolerance of 10^{-6} , multigrid clearly outperforms the best solver from homework 3, SOR. Note that MG with (1,1) as the pre-smooth, post-smooth steps takes slightly less work than the (2,1) and the same work as (2,2).

	iterations	work per iteration	total work
SOR	144	W	$144 W$
MG (1,1)	8	$\frac{4}{3}(2 + 4)W = 8W$	$64W$
MG (2,1)	7	$\frac{4}{3}(3 + 4)W = \frac{28}{3}W$	$65W$
MG (2,2)	6	$\frac{4}{3}(4 + 4)W = \frac{32}{3}W$	$64 W$

And for a 264×264 mesh ($h = 2^{-8}$) and tolerance of 10^{-7} , multigrid blows SOR out of the water. Note that MG with (1,1) pre-smooth, post-smooth steps takes slightly less work than the (2,1) and (2,2) counterparts, so the choice of (1,1) as the optimal smoothing count is justified.

	iterations	work per iteration	total work
SOR	658	W	$658 W$
MG (1,1)	9	$8W$	$72W$
MG (2,1)	8	$\frac{28}{3}W$	$75 W$
MG (2,2)	7	$\frac{32}{3}W$	$74.7 W$

```
#Multigrid V-Cycle Solver
```

```
#Multigrid V-cycle with  
#full-weighting, bilinear interpol, GS-RB smoothing  
#for Poission equation with Dirichlet BCs
```

```
def V_cycle(u, f, h, v1, v2):  
    #presmooth v1 times  
    u = GS_RB_smoother(u,f, h, v1)  
  
    #compute residual  
    res = compute_residual(u, f, h)  
  
    #restrict residual  
    res2 = full_weighting_restriction(res, h)  
  
    #solve for coarse grid error, check grid level to decide whether to solve or be recursive  
    if h == 2**(-2):  
        error = trivial_direct_solve(res2, 2*h)  
    else:  
        error = np.zeros((int(1/(2*h)+1), int(1/(2*h)+1)), dtype=float)  
        error = V_cycle(error, res2, 2*h, v1, v2)  
  
    #interpolate error  
    error2 = bilinear_interpolation(error, 2*h)  
  
    #correct (add error back in)  
    u = u+error2  
  
    #post-smooth v2 times  
    u = GS_RB_smoother(u, f, h, v2)  
  
    return u
```

#GS-RB Smoother

```
def GS_RB_smoother(u, f, h, steps):

    #set number of grid points in each row/column
    n = int(1/h - 1)

    #separate red, black indices into two lists
    reds = []
    blacks = []
    for i in range(1,n+1):
        for j in range(1,n+1):
            if (i+j)%2==0:
                reds.append((i,j))
            else:
                blacks.append((i,j))

    #begin iterative scheme
    for k in range(steps):

        #loop red
        for (i,j) in reds:
            # print "red", i,j
            u[i][j] = (1/4)*(u[i-1][j]+u[i][j-1]+u[i+1][j] + u[i][j+1] - (h**2)*f[i][j])

        #loop black
        for (i,j) in blacks:
            # print "black", i,j
            u[i][j] = (1/4)*(u[i-1][j]+u[i][j-1]+u[i+1][j] + u[i][j+1] - (h**2)*f[i][j])

    return u
```

```
#Compute Residual
```

```
def apply_Laplacian(u, h, A):
    #remove Dirichlet BC's from u to work with Au
    #and flatten the matrix into a vector for Au multiplication
    u_inner = u[1:-1, 1:-1].flatten(order='C')
    # A = get_Laplacian(h)

    #apply Laplacian to get matrix-vector product
    product = A.dot(u_inner)

    #shape matrix-vector product back into matrix with Dirichlet BC padding
    n = int(1/h)-1
    product_matrix = np.reshape(product, (n, n))
    padded_prod_matrix = np.pad(product_matrix, ((1,1),(1,1)), mode='constant')

    return padded_prod_matrix

def get_Laplacian(h):
    N = int(1/h)-1

    # Define diagonals
    off_diag = 1*np.ones(N)
    diag = (-2)*np.ones(N)

    # Generate the diagonal and off-diagonal matrices
    A = np.vstack((off_diag,diag,off_diag))
    lap1D = scipy.sparse.dia_matrix((A,[-1,0,1]),shape=(N,N))
    speye = scipy.sparse.identity(N)
    L = (1/(h**2))*(scipy.sparse.kron(lap1D,speye) + scipy.sparse.kron(speye,lap1D))
    return L

def compute_residual(u, f, h, L):
    #apply Laplacian to u
    Au = apply_Laplacian(u,h, L)

    #compute residual
    residual = f - Au

    return residual
```

#Full-Weighting Restriction

```
def full_weighting_restriction(u, h):
    u_f = u + 0
    h2 = 2*h
    n2 = int(1/h2)-1
    u_c = np.zeros((n2+2, n2+2), dtype=float)

    #loop over coarse mesh, holding boundary entries zero as per Dirichlet BC
    for i in range(1,n2+1):
        for j in range(1,n2+1):
            u_c[i][j] = 1/16*(4*u_f[2*i][2*j] + 2*u_f[2*i-1][2*j] + 2*u_f[2*i][2*j-1] + 2*u_f[2*i][2*j+1])
    return u_c
```

#Direct Solve

```
def trivial_direct_solve(residual, h):
    #for when h=1/2 and direct solve is completely trivial
    n = int(1/h)-1
    approx_u = np.zeros((n+2, n+2), dtype=float)
    approx_u[1][1] = -h**2 * residual[1][1] / 4
    return approx_u
```

#Bilinear Interpolation

```
def bilinear_interpolation(u_c, h):
    n = int(1/h)-1
    n2 = int(1/(2*h))-1
    u_f = np.zeros((n2+2, n2+2), dtype=float)

    #loop over coarse mesh, holding boundary entries zero as per Dirichlet BC
    for i in range(1,n+1):
        for j in range(1,n+1):
            #the 4 contribution from the stencil
            u_f[2*i][2*j] = u_c[i][j]
            #the 2 contributions from the stencil
            u_f[2*i+1][2*j] += u_c[i][j]/2
            u_f[2*i][2*j+1] += u_c[i][j]/2
            u_f[2*i-1][2*j] += u_c[i][j]/2
            u_f[2*i][2*j-1] += u_c[i][j]/2
            #the 1 contributions from the stencil
            u_f[2*i+1][2*j+1] += u_c[i][j]/4
            u_f[2*i+1][2*j-1] += u_c[i][j]/4
            u_f[2*i-1][2*j+1] += u_c[i][j]/4
            u_f[2*i-1][2*j-1] += u_c[i][j]/4

    return u_f
```

```
#Problem 1 Main

#Use Multigrid V-cycle with
#full-weighting, bilinear interpol, GS-RB smoothing
#to find soluton to Laplacian  $u = -e^{-(x-.25)^2 - (y-.6)^2}$ 
#with homogeneous Dirichlet BC's
#for different grid spacings

from __future__ import division

import numpy as np
from math import exp, sin, pi

from tqdm import tqdm
import tabulate
import copy
import argparse
from time import clock
from multigrid_V_cycle import V_cycle
from compute_residual import compute_residual, get_Laplacian, apply_Laplacian_nomatrix
from GS_RB_smoother import GS_RB_smoother

def RHS_function_sampled(h):
    n = int(1/h)-1
    #set x,y grid point vectors (n x 1)
    x = [i*h for i in range(n+2)]
    y = [j*h for j in range(n+2)]

    f = np.zeros((n+2,n+2),dtype=float)
    for i in range(1,n+1):
        for j in range(1,n+1):
            f[i][j] = -exp(-(x[i]-0.25)**2 - (y[j]-0.6)**2)
    return f

def main():
    #create grid spacings data
    grid_spacings = [2**(-5), 2**(-6), 2**(-7), 2**(-8), 2**(-9)]

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

    #create iteration count holder and runtimes holder
    itcounts = []
    times = []
```

```
#set tolerance for stopping criterion
tol = 10**(-10)

#loop over different grid spacings
for h in tqdm(grid_spacings):
    #time multigrid code for grid spacing h
    toc = clock()

    #set up initial solution guess
    n = int(1/h - 1)
    u = np.zeros((n+2, n+2), dtype=float)

    #get RHS
    f = RHS_function_sampled(h)

    #use multigrid algorithm
    itcount = 0
    while True:
        itcount += 1

        #use a V-cycle iteration
        u=V_cycle(u, f, h, 2,2, Laplacians)

        #compute residual of solution
        res = compute_residual(u, f, h, Laplacians[int(-2-np.log2(h))])

        #check convergence using norm of residual relative to norm of RHS function
        if np.amax(np.abs(res)) < tol*np.amax(np.abs(f)):
            break

    #stop timer, collect time and iteration count into data table
    tic = clock()
    itcounts.append(itcount)
    times.append(tic-toc)

#create table of output data for test case and for problem solution
table = [[grid_spacings[i], itcounts[i], times[i]] for i in range(np.size(grid_spacings))]
print tabulate.tabulate(table, headers = ["grid spacing h", "iteration count", "run time (

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


┌ **Problem 2.** Numerically estimate the average convergence factor,

$$E_k = \left(\frac{\|e^{(k)}\|_\infty}{\|e^{(0)}\|_\infty} \right)^{1/k},$$

for different numbers of presmoothing steps, ν_1 , and postsmoothing steps, ν_2 , for $\nu = \nu_1 + \nu_2 \leq 4$. Be sure to use a small value of k because convergence may be reached very quickly. What test problem did you use? Do your results depend on the grid spacing? Report the results in a table, and discuss which choices of ν_1 and ν_2 give the most efficient solver.

└

I created a problem using the “known solution” $u(x, y) = \sin(\pi x) \sin(\pi y)$ sampled on a mesh on the unit square $(0, 1) \times (0, 1)$ with homogeneous Dirichlet boundary conditions, and created a “right-hand side” function by applying the Discrete Laplacian to this solution vector. I then used my multigrid program from problem 1 to solve this algebraic problem. I performed an analysis of all the different pairings (ν_1, ν_2) for grid spacings $h = 2^{-5}, 2^{-6}, 2^{-7}, 2^{-8}$ with stopping criterion relative iterate differences with tolerance 10^{-10} , and these all achieved relatively similar results. The results do not depend significantly on the grid spacing, as the data will testify. I report the average convergence factors for 1-5 iterations (e.g., E_3 is the average convergence factor among 3 iterations, while E_5 is the average convergence factor among 5 iterations), and 5 was chosen as the largest k to consider since my lowest reported iteration count for a multigrid solve was 8. I found that despite the larger average convergence factor for (1,1) pre/post-smooth steps MG, the less work it entails results in faster performance on finer grids, so the optimal MG uses (1,1) smoothing. The higher the smooth count, the lower the average convergence factor, but the extra work involved doesn’t end up paying off. There also weren’t major differences between similar smoothing patterns, e.g., (1,2) and (2,1) performed about the same, with one winning over the other seemingly randomly, and (2,2), (1,3), and (3,1) performing similarly as well. The following table is for $h = 2^{-5}$, tolerance 10^{-10} .

(ν_1, ν_2)	E_1	E_2	E_3	E_4	iterations	run times
(0, 1)	0.305393	0.301592	0.297944	0.291537	20	0.519646
(1, 0)	0.293605	0.288987	0.291686	0.291592	24	0.68032
(1, 1)	0.11991	0.116894	0.0975584	0.156273	12	0.404654
(0, 2)	0.179352	0.176355	0.1692	0.118281	14	0.515413
(2, 0)	0.193867	0.186207	0.180232	0.149008	16	0.589619
(1, 2)	0.0805982	0.0760598	0.0647618	0.166059	10	0.390981
(2, 1)	0.0809173	0.0763635	0.064411	0.166031	10	0.405133
(3, 0)	0.140947	0.131789	0.117586	0.162013	13	0.514157
(0, 3)	0.121375	0.117422	0.0979251	0.156212	12	0.471079
(2, 2)	0.0606403	0.0543182	0.0831176	0.167628	9	0.41648
(1, 3)	0.0606286	0.0543059	0.0831232	0.167629	9	0.407873
(3, 1)	0.0607858	0.0544584	0.0830569	0.167623	9	0.443
(4, 0)	0.109314	0.100546	0.0728343	0.166327	12	0.590817
(0, 4)	0.0914259	0.0863728	0.0404893	0.164779	10	0.490951

For $h = 2^{-6}$, tolerance 10^{-10} ,

(ν_1, ν_2)	E_1	E_2	E_3	E_4	iterations	run times
(0, 1)	0.305759	0.303893	0.302047	0.29956	20	1.60016
(1, 0)	0.294261	0.295252	0.298011	0.300046	26	2.12789
(1, 1)	0.12071	0.119945	0.115927	0.0577091	12	1.3413
(0, 2)	0.180233	0.179461	0.177762	0.170408	14	1.42908
(2, 0)	0.194353	0.191018	0.188791	0.182702	17	1.83374
(1, 2)	0.0815659	0.0804702	0.0700874	0.111815	10	1.3723
(2, 1)	0.0818544	0.0807546	0.070458	0.11171	10	1.37951
(3, 0)	0.141378	0.137511	0.133371	0.111955	14	1.91811
(0, 3)	0.122455	0.121482	0.117529	0.0682134	12	1.613
(2, 2)	0.0616633	0.0601744	0.0327965	0.11682	9	1.46634
(1, 3)	0.0616527	0.0601636	0.0327591	0.116821	9	1.42421
(3, 1)	0.061792	0.0603017	0.0332196	0.116802	9	1.41787
(4, 0)	0.109726	0.106782	0.100253	0.113894	13	2.00789
(0, 4)	0.0926053	0.0913771	0.0837286	0.106415	10	1.51836

For $h = 2^{-7}$, tolerance 10^{-10} :

(ν_1, ν_2)	E_1	E_2	E_3	E_4	E_5	iterations	run times
(0, 1)	0.305883	0.304538	0.303238	0.301932	0.300611	21	6.12058
(1, 0)	0.294453	0.297938	0.300834	0.302942	0.310499	27	7.75794
(1, 1)	0.120953	0.1209	0.12086	0.120824	0.120791	12	4.72569
(0, 2)	0.180494	0.180366	0.180299	0.180247	0.180203	14	5.4967
(2, 0)	0.194508	0.193079	0.192567	0.193106	0.192909	18	7.12078
(1, 2)	0.0818537	0.0818409	0.0818362	0.0818337	0.0818319	10	5.45242
(2, 1)	0.0821343	0.0821198	0.0821137	0.0821098	0.0821067	10	5.38711
(3, 0)	0.141522	0.139438	0.138227	0.137223	0.135833	15	8.59654
(0, 3)	0.122769	0.122683	0.122654	0.122639	0.12263	12	5.94317
(2, 2)	0.061966	0.0619567	0.0619537	0.0619522	0.0619514	9	5.2689
(1, 3)	0.0619555	0.0619462	0.0619431	0.0619415	0.0619406	9	5.2587
(3, 1)	0.0620902	0.0620807	0.0620776	0.0620761	0.0620753	9	5.29586
(4, 0)	0.109868	0.108229	0.107211	0.105716	0.104187	13	7.61545
(0, 4)	0.0929452	0.092863	0.0928356	0.0928219	0.0928137	10	5.8819

For $h = 2^{-8}$, tolerance 10^{-10} ,

(ν_1, ν_2)	E_1	E_2	E_3	E_4	iterations	run times
(0, 1)	0.305871	0.304606	0.303314	0.301955	21	28.4614
(1, 0)	0.294466	0.29897	0.302007	0.304028	29	38.5927
(1, 1)	0.120959	0.120882	0.120614	0.119052	12	21.1899
(0, 2)	0.180508	0.180424	0.180285	0.179836	14	25.349
(2, 0)	0.194505	0.193764	0.193112	0.193811	19	34.2363
(1, 2)	0.0818681	0.0817999	0.081246	0.0754371	10	21.9906
(2, 1)	0.0821466	0.0820787	0.081529	0.0757927	10	22.4021
(3, 0)	0.141514	0.139864	0.139102	0.138534	16	35.8415
(0, 3)	0.122792	0.12273	0.122495	0.121037	12	26.4578
(2, 2)	0.0619828	0.0618914	0.0608837	0.038593	9	24.3085
(1, 3)	0.0619724	0.0618809	0.0608727	0.0385485	9	24.4019
(3, 1)	0.0621058	0.0620156	0.0610131	0.0391059	9	23.787
(4, 0)	0.109856	0.108447	0.107895	0.105943	14	37.0952
(0, 4)	0.0929733	0.0928971	0.0924709	0.088774	10	26.4858

#MAT228A HW 4

#Number 2

#Numerically estimate error convergence of

#multigrid solver

#with homogeneous Dirichlet BC's

#using known problem $u_{xx} + u_{yy} = -2 \pi^2 \sin \pi x \sin \pi y$

#which has solution $u(x,y) = \sin \pi x \sin \pi y$

```
from __future__ import division
```

```
import numpy as np
```

```
from math import exp, sin, pi
```

```
from tqdm import tqdm
```

```
import matplotlib.pyplot as plt
```

```
import scipy.sparse as sparse
```

```
import scipy.sparse.linalg
```

```
import tabulate
```

```
import argparse
```

```
from timeit import default_timer as timer
```

```
from time import clock
```

```
from scipy.stats.mstats import gmean
```

```
from multigrid_V_cycle import V_cycle
```

```
from compute_residual import compute_residual, get_Laplacian, apply_Laplacian
```

```

def known_solution(h):
    n = int(1/h)-1
    #set x,y grid point vectors (n x 1)
    x = [i*h for i in range(n+2)]
    y = [j*h for j in range(n+2)]

    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*x[i])*sin(pi*y[j])
    return u

def main():
    #test problem grid spacing size
    h= 2**(-7)
    n = int(1/h - 1)

    #initialize 0th iterate u
    u = np.zeros((n+2,n+2))

    #compute a known solution
    SOL = known_solution(h)

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

    #generate a RHS that SOL is the solution to the algebraic eqn Lu = f
    f = apply_Laplacian(SOL,h, Laplacians[int(-2-np.log2(h))])

    #initialize the error vector between our known solution SOL and 0th iterate u
    init_error = np.amax(np.abs(SOL-u))

    #tolerance for stopping criterion
    tol = 10**(-10)

    #smoothing step counts to check over
    smooth_steps = [(0,1), (1, 0), (1, 1), (0,2), (2,0), (1,2), (2, 1), (3,0), (0,3), (2, 2),
    step_number=0
    conv_Factor = np.zeros((15,8))
    itcounts = np.zeros(15)
    times = np.zeros(15)

    for step in tqdm(smooth_steps):

```

```

errors = [init_error]
u = np.zeros((n+2,n+2))
#do multigrid iteration
itcount=0
tic = clock()
while True:
    u_old = u+0
    itcount += 1
    #use a V-cycle iteration with smoothing steps as given
    u = V_cycle(u_old, f, h, step[0], step[1], Laplacians)
    res = compute_residual(u, f, h, Laplacians[int(-2-np.log2(h))])

    #calculate convergence factor
    error_k = np.amax(np.abs(SOL-u))
    errors.append(error_k)

    #stop when iterate differences within relative tol
    if np.amax(np.abs(res)) < tol*np.amax(np.abs(f)):
        break

toc = clock()
print toc-tic

#compute convergence factors
conv_factors = [(errors[k]/errors[0])** (1/k) for k in range(1,6)]
#save data for table
for i in range(5):
    conv_Factor[step_number][i] = conv_factors[i]
    itcounts[step_number] = itcount
    times[step_number]=toc-tic

step_number+=1

# print errors[itcount]

#create table of output data
print "h = ", h
print "tol =", tol
smoothing_table = [[smoothing_steps[i], conv_Factor[i][0], conv_Factor[i][1], conv_Factor[i][2]]
print tabulate.tabulate(smoothing_table, headers = ["v1, v2", "ave of 1", "average of 2",

if __name__ == "__main__":
    main()

```