Homework #4

Sam Fleischer

December 2, 2016

Part I

White 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.

Note: If you cannot get a V-cycle code working, write a simple code such as a 2-grid code. You can also experiment in one dimension (do not use GSRB in 1D). You may turn in one of these simplified codes for reduced credit. You should state what your code does, and use your code for the assignment.

1. Use your V-cycle code to solve
$$\nabla^2 u = -\exp\left[-(x-0.25)^2-(y-0.6)^2\right]$$
on the unit square $(0,1)\times(0,1)$ with homogeneous Dirichlet boundary conditions for different grid spaces. 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.

Rather than working with grids of size 2^n+1 for integers n and include 0s on the boundary, I chose to work with grids of size 2^n-1 for integers n. My code (written in Python) only takes the interior points into account. Here is my restriction operator:

```
1  def restriction(u,power):
2      # u:     the fine mesh to restrict
3      # power: the power of 2 of the fine mesh
4
5      # get the size of the coarse grid
6      coarse_grid = 2**(power-1)-1
7      # initialize the coarse grid
8      v = np.zeros((coarse_grid,coarse_grid))
9      # loop through the coarse grid
10     for i in xrange(0,coarse_grid):
11         for j in xrange(0,coarse_grid):
12             # gather data from the first row in the fine grid
13             row_1 = u.item(((2*i),(2*j)))
14             row_1 += 2*u.item(((2*i),(2*j)+1))
15             row_1 += u.item(((2*i),(2*j)+2))
16             # gather data from the second row in the fine grid
17             row_2 = 2*u.item(((2*i)+1,(2*j)))
18             row_2 += 4*u.item(((2*i)+1,(2*j)+1))
19             row_2 += 2*u.item(((2*i)+1,(2*j)+2))
20             # gather data from the third row in the fine grid
21             row_3 = u.item(((2*i)+2,(2*j)))
22             row_3 += 2*u.item(((2*i)+2,(2*j)+1))
23             row_3 += u.item(((2*i)+2,(2*j)+2))
24             # combine data for coarse grid
25             v[i][j] = (1/16)*(row_1 + row_2 + row_3)
26     # return the coarse grid
27     return v
```

and here is my interpolation operator:

```
1  def interpolation(v,power):
2      # v:     the coarse mesh to interpolate
3      # power: the power of 2 of the fine mesh
4
```

```
    # get the size of the coarse and fine grids
    coarse_grid = 2**(power-1)-1
    fine_grid = 2**power-1
    # initialize the fine grid
    U = np.zeros((fine_grid,fine_grid))
    # loop through the coarse grid
    for i in xrange(coarse_grid):
        for j in xrange(coarse_grid):
            # add coarse grid data to the first row of the fine grid
            U[(2*i)][(2*j)] += (1/4)*v[i][j]
            U[(2*i)][(2*j)+1] += (1/2)*v[i][j]
            U[(2*i)][(2*j)+2] += (1/4)*v[i][j]
            # add coarse grid data to the second row of the fine grid
            U[(2*i)+1][(2*j)] += (1/2)*v[i][j]
            U[(2*i)+1][(2*j)+1] += v[i][j]
            U[(2*i)+1][(2*j)+2] += (1/2)*v[i][j]
            # add coarse grid data to the third row of the fine grid
            U[(2*i)+2][(2*j)] += (1/4)*v[i][j]
            U[(2*i)+2][(2*j)+1] += (1/2)*v[i][j]
            U[(2*i)+2][(2*j)+2] += (1/4)*v[i][j]
    #return the fine grid
    return U
```

Here is my Gauss-Seidel Red-Black code:

```
def get_checkerboard_of_size(s):
    # s: size of grid

    # make a row of 1 0 1 0 ..
    red = np.r_[int(s/2)*[1,0] + [1]]
    # make a row of 0 1 0 1 ..
    black = np.r_[int(s/2)*[0,1] + [0]]
    # put the rows in one list
    t = int(s/2)*(red, black)
    t += red,
    # stack the rows in a matrix
    checkerboard = np.row_stack(t)
    # return the checkerboard
    return checkerboard

def GS_iteration(inds, u, rhs, N, h):
    # inds: which indices to iteration
    # u:    the grid to smooth
    # rhs:  which right hand side function to use
    # N:    the size of the grid
    # h:    the grid spacing

    # copy the grid to save data from un-iterated indices
    # v = deepcopy(u)
    # loop over relevant indices
    for (i,j) in inds:
        # get the data from the grid point above (0 if top row)
        first  = 0 if i == 0 else u[i-1][j]
        # get the data from the grid point to the left (0 if first column)
        second = 0 if j == 0 else u[i][j-1]
```

```
        # get the data from the grid point below (0 if bottom row)
        third  = 0 if i == N else u[i+1][j]
        # get the data from the grid point to the right (0 if last column)
        fourth = 0 if j == N else u[i][j+1]
        # replace data point
        u[i][j] = (1/4)*(first + second + third + fourth - (h**2)*rhs[i][j])
    # return replaced grid
    return u

def GSRB(u,rhs,N,h):
    # u:   the grid to smooth
    # rhs: the right hand side function to use
    # N:   the size of the grid
    # h:   the grid spacing

    # Get a checkerboard of 1s and 0s
    checkerboard = get_checkerboard_of_size(u.shape[0])
    # Gather the indices of the "red" and "black" points
    red_indices = np.where(checkerboard == 1)
    black_indices = np.where(checkerboard == 0)
    red_indices = zip(red_indices[0],red_indices[1])
    black_indices = zip(black_indices[0],black_indices[1])
    # Iterate over the red points
    v = GS_iteration(red_indices,u,rhs,N-3,h)
    # Iterate over the black points
    z = GS_iteration(black_indices,v,rhs,N-3,h)
    #return the smoothed grid
    return z
```

Here is a simple function to get a mesh of size $2^n - 1$:

```
def get_mesh(power):
    # power: the power of two to use

    # get the grid size (including boundaries)
    N = 2**power+1
    # get equally spaced grid between 0 and 1 (inclusive) and step size
    x,h = np.linspace(0,1,N,retstep=True)
    # omit 0 and 1
    x = x[1:len(x)-1:]
    # make the mesh
    X,Y = np.meshgrid(x,x)
    # return the mesh, number of grid points, and gridsize
    return X,Y,N,h
```

Here is a function to generate a discrete Laplacian in 1D:

```
def Lap_1D(N):
    # N: size of the 1D discrete Laplacian

    # define the Laplacian as the tridiagonal matrix with 1s
    # on the super- and sub-diagonals and -2 on the diagonal
    off_diag = 1*np.ones(N)
    diag = (-2)*np.ones(N)
    A = np.vstack((off_diag,diag,off_diag))
    Laplacian_1D = scipy.sparse.dia_matrix((A,[-1,0,1]),shape=(N,N))
```

```
    # return the Laplacian
    return Laplacian_1D
```

And here is a function to cleverly generate the discrete Laplacian in 2D using Kronecker products:

```
def get_laplacian(N):
    # N: N^2 x N^2 is the size of the 2D Laplacian

    # get the spacing
    h = 1/(N+1)
    # Get the 1D Laplacian
    Laplacian_1D = Lap_1D(N)
    # Get an NxN identity matrix
    I = scipy.sparse.identity(N)
    # Get kronecker products
    kron_prod_1 = scipy.sparse.kron(Laplacian_1D,I)
    kron_prod_2 = scipy.sparse.kron(I,Laplacian_1D)
    # Cleverly define the 2D Laplacian as the sum of Kronecker products
    Laplacian_2D = (1/(h**2))*(kron_prod_1 + kron_prod_2)
    # return the 2D Laplacian
    return Laplacian_2D
```

Here is a function to compute the residual:

```
def compute_residual(u,f,Ls):
    # u: approximate solution
    # f: right-hand-side of Au = f

    # Get the Laplacian of the appropriate size
    A = Ls[str(int(log(u.shape[0]+1,2)))]
    # Flatten u
    uflat = u.flatten()
    # Compute Au and reshape it back into the appropriate shape
    Au = A.dot(uflat).reshape(u.shape)
    # define the residual as f - Au
    R = f - Au
    # return the residual
    return R
```

Here is a simple function to directly solve the system with Dirichlet boundary conditions when the grid size is 1:

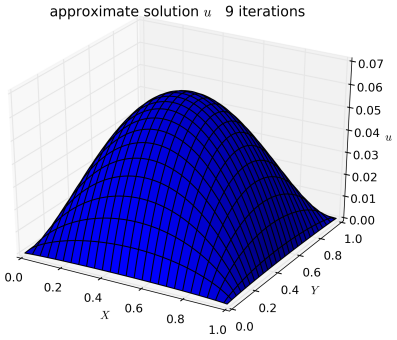
```
def trivial_direct_solve(h,r):
    # h: grid spacing
    # r: residual input (1x1 matrix)

    # get the only item in the matrix
    R = r.item((0,0))
    # solve the equation
    sol = (-(h**2)/4)*R
    # return the solution, reshaped into a 1x1 matrix
    return np.asarray(sol).reshape((1,1))
```

Finally, here is the V-cycle code:

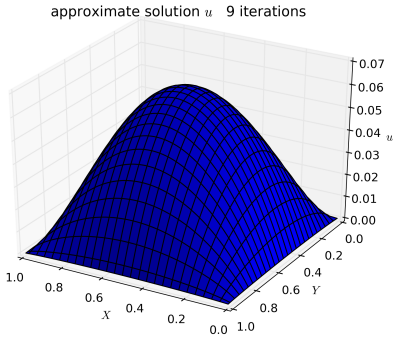
```
def V_cycle(power,u,f,nu,X,Y,N,h,Ls):
    # power: the power of 2
    # u:     the grid to iterate
    # f:     the righthand side function to use
```

```
    # nu:    a 2-tuple of pre- and post-smooth numbers
    # X,Y:   the mesh
    # N:     number of points in the grid (including 0 and 1)
    # h:     grid spacing

    # copy the input to a new grid
    v = deepcopy(u)

    # presmooth nu_1 times
    for i in xrange(nu[0]):
        # smooth the grid
        v = GSRB(v,f,N,h)

    # Compute the residual
    r = compute_residual(v,f,Ls)

    # restrict the residual
    r = restriction(r,power)

    # if this is the smallest possible grid, direct solve
    if power == 2:
        # get the error analytically
        e = trivial_direct_solve(2*h,r)
    # otherwise, call the V_cycle code recursively
    else:
        # create a coarse mesh
        X_c,Y_c,N_c,h_c = get_mesh(power-1)
        # initialize a guess for the error
        e_guess = np.zeros((2**(power-1) - 1, 2**(power-1) - 1))
        # run the V_cycle code for the residual equation
        e = V_cycle(power-1, e_guess, r, nu, X_c, Y_c, N_c, h_c, Ls)

    # interpolate the error
    e = interpolation(e,power)

    # correct the solution
    v = v + e

    # postsmooth nu_2 times
    for i in xrange(nu[1]):
        # smooth the grid
        v = GSRB(v,f,N,h)

    # return the smoothed and iterated grid
    return v
```

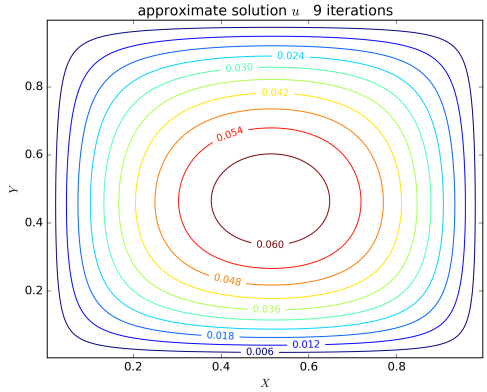
Here are some results of the V-cycle code:



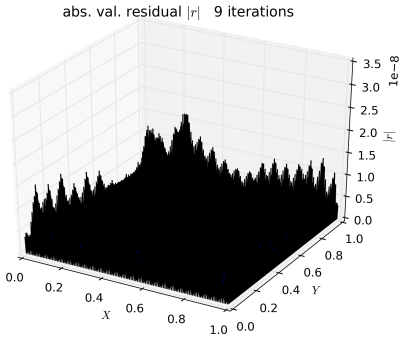
solution, azimuth= −60



solution, azimuth= 120



solution contour

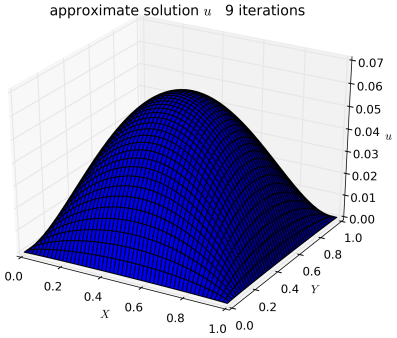


absolute value of the residual

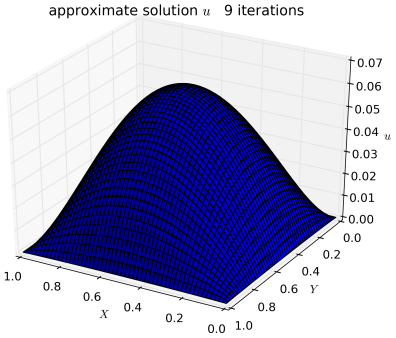
The above simulation was run with relative tolerance 10^{-7}, $h = 2^{-8}$, $\nu_1 = \nu_2 = 1$, and took 9 iterations to converge. Top row: approximate solution u from two different views (note the X and Y axes). Bottom Left: approximate solution contour plot. Bottom right: The residual of the approximate solution (note the residual is on the order of 10^{-8}).

k	$\|r_k\|_\infty$	$\frac{\|r_k\|_\infty}{\|r_{k-1}\|_\infty}$
1	0.368102	-
2	0.0487561	0.132452981403
3	0.00636013	0.130447720916
4	0.0008433	0.132591695754
5	0.00011053	0.131068270613
6	1.43467e-05	0.129798992138
7	1.84936e-06	0.128905142346
8	2.37102e-07	0.128207510041
9	3.02627e-08	0.127635734106

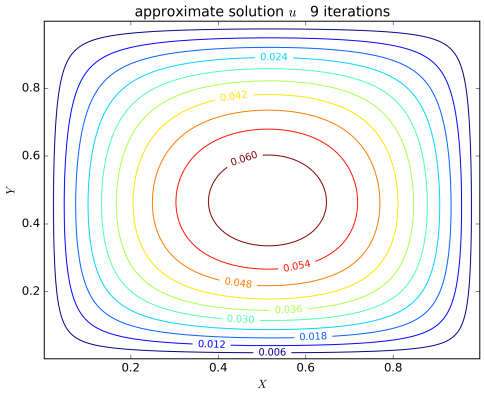
The above table shows the relative residuals of the above simulation. We see ≈ 87% reduction in residual each iteration. In the next simulation, we will change ν_1 and get better convergence.



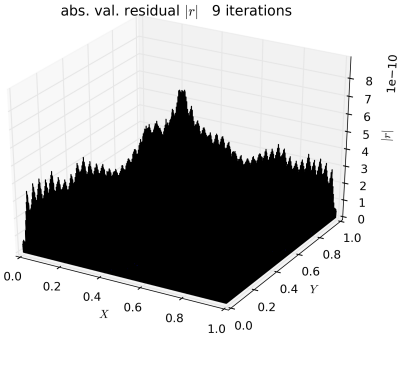
solution, azimuth= −60



solution, azimuth= 120



solution contour



absolute value of the residual

This simulation was run with relative tolerance 10^{-9}, $h = 2^{-9}$, $\nu_1 = 2$, $\nu_2 = 1$, and took 9 iterations to converge. Note the residual is on the order of 10^{-10}.

k	$\|r_k\|_\infty$	$\frac{\|r_k\|_\infty}{\|r_{k-1}\|_\infty}$
1	0.242426	-
2	0.0222294	0.0916957149997
3	0.00198422	0.0892610393309
4	0.000174193	0.0877893609761
5	1.52764e-05	0.0876981172233
6	1.34612e-06	0.0881171180771
7	1.1793e-07	0.0876074906573
8	1.03002e-08	0.087341465529
9	8.88707e-10	0.0862809077436

The above table shows the relative residuals of the above simulation. We see ≈ 91% reduction in residual each iteration.

The following table shows the work needed to solve this system for $h = 2^{-6}$ and tol $= 10^{-6}$.

Method	Iterations	Work per Iteration	Total Work
SOR	144	1 work unit	144 work units
MG $(\nu_1, \nu_2) = (1,1)$	8	$\frac{4}{3}(2+4)$ work units	64 work units
MG $(\nu_1, \nu_2) = (2,1)$	7	$\frac{4}{3}(3+4)$ work units	65 work units
MG $(\nu_1, \nu_2) = (2,2)$	6	$\frac{4}{3}(4+4)$ work units	64 work units

and the following table shows the work needed to solve this system for $h = 2^{-8}$ and tol $= 10^{-7}$.

Method	Iterations	Work per Iteration	Total Work
SOR	658	1 work unit	658 work units
MG $(\nu_1, \nu_2) = (1,1)$	9	$\frac{4}{3}(2+4)$ work units	72 work units
MG $(\nu_1, \nu_2) = (2,1)$	8	$\frac{4}{3}(3+4)$ work units	75 work units
MG $(\nu_1, \nu_2) = (2,2)$	7	$\frac{4}{3}(4+4)$ work units	75 work units

The work per iteration for the multigrid code is $\frac{4}{3}(\nu_1 + \nu_2 + 4)$ where 4 is for the computing the residual, restriction, interpolation, and correction. Clearly the Multigrid code is desirable.

Part II

Choose **one** of the following problems.

2. Numerically estimate the average convergence factor,

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

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.

3. The multigrid V-cycle iteration is of the form

$$u^{k+1} = (I - BA)u^k + Bf,$$

where $M = I - BA$ is the multigrid iteration matrix. To compute the kth column of the multigrid iteration matrix, apply a single V-cycle to a problem with zero right-hand-side, f, and as an initial guess, u^0, that has a 1 in the kth entry and zeros everywhere else. For a small problem, e.g. $h = 2^{-5}$ or $h = 2^{-6}$, form the multigrid matrix, compute the eigenvalues, and plot them in the complex plane. Compute the spectral radius and 2-norm of the multigrid iteration matrix for different numbers of presmoothing steps ν_1, and postsmoothing steps, ν_2. Comment on your results.

For the following tables, I am defining $E_k := \left(\frac{\left\| e^{(k)} \right\|_\infty}{\left\| e^{(0)} \right\|_\infty} \right)^{1/k}$. To test this convergence rate, I initialized a random vector u, then passed it through the Laplacian ∇^2 to generate f. Then I initialized a guess of $u_0 \equiv 0$ and ran the V-cycle code for a number of combinations of ν_1 and ν_2. I solved the system numerically for $h = 2^{-i}$ for $i = 6, 7, 8, 9$ and tol $= 10^{-10}$. The following four tables show the results for the four starting grid spacings. The first two columns specify which values of ν used. The third-seventh columns show the E_1-E_5. The eighth column shows the num-

ber of iterations until convergence, and the ninth column shows the total time to completion. Red entries are the smallest values in their respective columns, for a given $\nu = \nu_1 + \nu_2$.

We see the best overall solver is $\nu_1 = \nu_2 = 1$. Even though solvers with more smoothing steps have faster convergence with respect to a single iteration, each iteration takes longer, resulting in a slowdown. In other words, the decrease in iteration count does not make up for the increase in smoothing time. We also see that fast initial residual decrease does not necessarily mean it will converge first. These results are independent of grid spacing.

ν_1	ν_2	E_1	E_2	E_3	E_4	E_5	iterations	time
0	1	0.648938	0.43606	0.411233	0.399007	0.390484	19	0.724328
1	0	0.487977	0.358102	0.347434	0.342882	0.340151	21	0.804955
0	2	0.357835	0.242341	0.225531	0.218747	0.2141	12	0.767544
1	1	0.116416	0.10496	0.107306	0.109805	0.112209	9	0.617404
2	0	0.204627	0.180517	0.183596	0.184137	0.183818	13	0.879385
0	3	0.278089	0.174162	0.156014	0.147177	0.141667	10	0.828848
1	2	0.0640793	0.0687802	0.0735935	0.0760558	0.077577	8	0.664636
2	1	0.0676902	0.0720939	0.076164	0.0780719	0.079116	8	0.658428
3	0	0.141664	0.126482	0.126586	0.125945	0.125153	11	0.93447
0	4	0.232654	0.137391	0.119456	0.110986	0.106034	9	1.05532
1	3	0.0465466	0.053058	0.0563639	0.0581954	0.0592385	7	0.818631
2	2	0.0457535	0.0536427	0.0570361	0.0587056	0.0596329	7	0.73804
3	1	0.0459446	0.0541799	0.0574189	0.0589796	0.0598539	7	0.73296
4	0	0.108241	0.0995232	0.099261	0.0981586	0.0970414	10	1.03422

$h = 2^{-6}$, tol $= 10^{-10}$

ν_1	ν_2	E_1	E_2	E_3	E_4	E_5	iterations	time
0	1	0.70366	0.481388	0.429346	0.417088	0.408186	20	3.32238
1	0	0.485865	0.362892	0.348612	0.345926	0.346277	21	3.33875
0	2	0.434735	0.276942	0.252261	0.237177	0.226658	12	3.77767
1	1	0.122205	0.106187	0.108739	0.110965	0.11267	9	2.35127
2	0	0.235093	0.17992	0.18189	0.185694	0.187109	13	3.3626
0	3	0.335241	0.198951	0.177639	0.164716	0.156347	10	3.50262
1	2	0.0745486	0.0726438	0.07437	0.0761915	0.0774869	8	2.77373
2	1	0.0734148	0.0731706	0.0759087	0.077723	0.0788225	8	2.68093
3	0	0.151107	0.124623	0.126975	0.1282	0.12819	11	3.7284
0	4	0.275804	0.156577	0.137083	0.125968	0.119102	8	3.52157
1	3	0.0565209	0.0550546	0.0566352	0.058069	0.0590243	7	2.97657
2	2	0.05011	0.0543289	0.0568788	0.0583994	0.0593422	7	3.1198
3	1	0.0513003	0.0548133	0.0570978	0.058626	0.0595583	7	3.09173
4	0	0.104464	0.0981119	0.100041	0.100405	0.099938	10	4.3511

$h = 2^{-7}$, tol $= 10^{-10}$

ν_1	ν_2	E_1	E_2	E_3	E_4	E_5	iterations	time
0	1	0.706771	0.489336	0.445883	0.429412	0.420973	20	12.6659
1	0	0.534644	0.390108	0.360545	0.35716	0.357252	21	13.5175
0	2	0.465575	0.293528	0.267626	0.253044	0.242728	12	12.1829
1	1	0.136665	0.113503	0.11083	0.112133	0.113674	9	9.0864
2	0	0.226812	0.187928	0.187011	0.191024	0.192129	13	14.2484
0	3	0.364358	0.212581	0.187315	0.174013	0.164484	10	14.1303
1	2	0.0754427	0.0724907	0.0746506	0.0766038	0.0779877	8	11.2709
2	1	0.0723146	0.0735791	0.076192	0.0781318	0.0792078	8	10.9372
3	0	0.150527	0.127825	0.129261	0.130377	0.131182	11	14.9868
0	4	0.298608	0.172087	0.141158	0.129261	0.122096	8	13.7246
1	3	0.0540812	0.0543202	0.0567906	0.0583588	0.0594019	7	12.2086
2	2	0.0538676	0.0546608	0.0572044	0.0587821	0.0597278	7	12.1011
3	1	0.0519433	0.0548087	0.0573617	0.0590001	0.0599155	7	12.0719
4	0	0.108124	0.0997209	0.101444	0.102353	0.102668	10	17.1891

$h = 2^{-8}$, tol $= 10^{-10}$

ν_1	ν_2	E_1	E_2	E_3	E_4	E_5	iterations	time
0	1	0.814503	0.506634	0.47523	0.460882	0.448367	20	53.3505
1	0	0.552184	0.387626	0.371623	0.368553	0.367149	22	57.5223
0	2	0.474977	0.311891	0.274758	0.263955	0.255797	12	48.9866
1	1	0.138544	0.11245	0.1112	0.112476	0.114036	9	36.7199
2	0	0.224892	0.189581	0.1904	0.190866	0.191885	13	52.9136
0	3	0.374745	0.24126	0.198747	0.1787	0.169587	10	55.0487
1	2	0.0735864	0.0725248	0.0753096	0.0771247	0.0783378	8	44.2941
2	1	0.0745542	0.0756904	0.0774981	0.0789767	0.079857	8	44.0936
3	0	0.148192	0.12764	0.129203	0.130529	0.131821	11	60.5826
0	4	0.325174	0.199243	0.16092	0.144477	0.135421	8	56.1599
1	3	0.0534238	0.0550205	0.057335	0.0587958	0.0597648	7	49.0491
2	2	0.0525316	0.0556535	0.0579426	0.0594209	0.060221	7	49.1172
3	1	0.0536964	0.0562632	0.0582736	0.0596727	0.0604218	7	48.8892
4	0	0.109677	0.0996846	0.101494	0.103119	0.103739	10	70.1302

$h = 2^{-9}$, tol $= 10^{-10}$