```
import numpy as np
from scipy.sparse import *
from scipy import *
import numpy.linalg as la
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

import SimpleEquationDiscretizer as sed
import EquationDiscretizer1D as sed1D
import SolverMethods as sm
import FunctionExamples as fe
import TimeEquationDiscretizer as ted

tol = 0.00001

def is_pos_def(x):
    return np.all(np.linalg.eigvals(x) > 0)

flops = 0

# Class encapsulating the Multigrid method for a given 2D Poisson equation instance
class MultiGrid2D:

        def __init__(self, maxN, borderFunction, valueFunction, niu1 = 4, niu2 = 4, omega =
 1.95):
                # BorderFunction and valueFunction pass the function values in the Poisson
equation
                self.borderFunction = borderFunction
                self.valueFunction = valueFunction
                self.omega = omega

                # Niu1 and niu2 are the number of relaxation steeps per going-down and goin
g-up parts of the v-cycle
                self.niu1 = niu1
                self.niu2 = niu2

                # N is the discretization grid size
                self.N = maxN

                # Create and store grid for different levels of discretization
                self.discrLevel = []
                self.flops = 0
                i = 0
                while(maxN > 2):
                        assert(maxN % 2 == 0)
                        self.discrLevel.append(sed.SimpleEquationDiscretizer(maxN, self.bor
derFunction, self.valueFunction))
                        i += 1
                        maxN /= 2

        # Helper function for grid level N
        def getCoordinates(self, row, N):
                return int(row / (N + 1)), row % (N + 1)

        # Helper function for grid level N
        def getRow(self, i, j, N):
                return(i * (N + 1) + j)

        # Restrict function from grid level fineN to grid level coarseN using trivial injec
tion
        def restrict(self, r, fineN, coarseN):
                restr = []

                for(i, elem) in enumerate(r):
                        (x, y) = self.getCoordinates(i, fineN)

                        if(x % 2 == 0 and y % 2 == 0):
                                restr.append(elem)
```

```
                return restr


        # Interpolate function from grid level coarseN to grid level fineN with full weight
 stencil
        def interpolate(self, r, fineN, coarseN):
                interp = []
                flops = 0
                for i in range((fineN + 1) * (fineN + 1)):
                        (x, y) = self.getCoordinates(i, fineN)

                        if(x % 2 == 0 and y % 2 == 0):
                                index = self.getRow(x / 2, y / 2, coarseN)
                                value = r[index]
                                flops += 1

                        elif(x % 2 == 1 and y % 2 == 0):
                                index1 = self.getRow((x - 1) / 2, y / 2, coarseN)
                                index2 = self.getRow((x + 1) / 2, y / 2, coarseN)
                                value = (r[index1] + r[index2]) / 2.0
                                flops += 2

                        elif(x % 2 == 0 and y % 2 == 1):
                                index1 = self.getRow(x / 2, (y - 1) / 2, coarseN)
                                index2 = self.getRow(x / 2, (y + 1) / 2, coarseN)
                                value = (r[index1] + r[index2]) / 2.0
                                flops += 2

                        else:
                                index1 = self.getRow((x - 1) / 2, (y - 1) / 2, coarseN)
                                index2 = self.getRow((x + 1) / 2, (y - 1) / 2, coarseN)
                                index3 = self.getRow((x - 1) / 2, (y + 1) / 2, coarseN)
                                index4 = self.getRow((x + 1) / 2, (y + 1) / 2, coarseN)
                                value = (r[index1] + r[index2] + r[index3] + r[index4]) / 4
.0
                                flops += 4

                        if(x == 0 or y == 0 or x == fineN or y == fineN):
                                value = 0

                        interp.append(value)

                self.flops += flops
                return interp

        # Restrict function from grid level fineN to grid level coarseN using the transpose
 action of interpolate
        def restrictTransposeAction(self, r, fineN, coarseN):
                restr = []
                flops = 0

                for i in range((coarseN + 1) * (coarseN + 1)):
                        (x, y) = self.getCoordinates(i, coarseN)
                        (x, y) = (2 * x, 2 * y)
                        newEntry = r[self.getRow(x, y, fineN)]

                        divideFactor = 1.0

                        for (dX, dY) in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
                                newX = x + dX
                                newY = y + dY
                                if(0 <= newX and newX <= fineN and 0 <= newY and newY <= fi
neN):
                                        index = self.getRow(newX, newY, fineN)
                                        newEntry += 0.5 * r[index]
                                        divideFactor += 0.5
                                        flops += 2
```

```
                        for (dX, dY) in [(1, 1), (-1, 1), (-1, -1), (1, -1)]:
                                newX = x + dX
                                newY = y + dY
                                if(0 <= newX and newX <= fineN and 0 <= newY and newY <= fi
neN):
                                        index = self.getRow(newX, newY, fineN)
                                        newEntry += 0.25 * r[index]
                                        divideFactor += 0.25
                                        flops += 2

                        newEntry = 1.0 * newEntry / divideFactor
                        if(divideFactor < 4.0):
                                if(not(x == 0 or y == 0 or x == fineN or y == fineN)):
                                        print("Error1")

                        if(x == 0 or y == 0 or x == fineN or y == fineN):
                                newEntry = 0.0

                        restr.append(newEntry)

                self.flops += flops
                return restr

        # Vcycle iterates once a V-relaxation scheme of multigrid starting at level L for a
 grid size N
        def vcycle(self, N, L, f, initSol = [], omega = 1.95):
                level = L
                discr = self.discrLevel[level]
                fSize = len(f)

                if(level == len(self.discrLevel) - 1):
                        v = la.solve(discr.M.todense(), f)
                        return v

                else:
                        solver1 = sm.SolverMethods(
                                iterationConstant = self.niu1,
                                eqDiscretizer = discr,
                                b = f,
                                initSol = initSol,
                                )

                        # Omega is the smoother parameter
                        omega = self.omega
                        # omega = 2.0/(1.0 + math.sin(math.pi * 1.0 / N))
                        # print(self.niu1, self.niu2, omega)
                        v, _, _, _, flops= solver1.SSORIterate(omega)
                        self.flops += flops

                        coarseN = N  / 2

                        Mv = discr.M.dot(v)
                        self.flops += (2 * discr.M.getnnz() - len(v))

                        residual = np.subtract(f, Mv)
                        self.flops += len(f)

                        coarseResidual = self.restrictTransposeAction(residual, N, coarseN)

                        coarseV = self.vcycle(coarseN, level + 1, coarseResidual)

                        fineV = self.interpolate(coarseV, N, coarseN)
                        w = np.add(v, fineV)
                        self.flops += len(v)

                        solver2 = sm.SolverMethods(
                                iterationConstant = self.niu2,
```

```
                                eqDiscretizer = discr,
                                b = f,
                                initSol = w,
                                )

                        v2, _, _, _, flops= solver2.SSORIterate(omega)
                        self.flops += flops
                        return v2

        # IterateVCycles iterates the function vcycle() for t times to obtain a better appr
oximation
        def iterateVCycles(self, t):
                initSol = []
                N = self.N

                for i in range((N + 1) * (N + 1)):
                        (x, y) = self.getCoordinates(i, N)
                        if(x == 0 or y == 0 or x == N or y == N):
                                initSol.append(self.borderFunction(1.0 * x / N, 1.0 * y / N
))
                        else:
                                initSol.append(0.0)

                vErrors = []
                discr = self.discrLevel[0]
                f = np.copy(discr.valueVector2D)
                normF = la.norm(f)

                currSol = np.copy(initSol)

                for i in range(t):
                        if(i % 10 == 0):
                                print(i)

                        residual = np.subtract(f, discr.M.dot(currSol))
                        absErr = 1.0 * la.norm(residual) / (normF)
                        vErrors.append(math.log(absErr))

                        resSol = self.vcycle(N, 0, residual, np.zeros_like(currSol))
                        prevSol = np.copy(currSol)
                        currSol = np.add(currSol, resSol)

                        if(absErr < tol):
                                break

                return currSol, vErrors, self.flops


# Class encapsulating the Multigrid method for a given 1d Poisson equation instance
class MultiGrid:

        def __init__(self, maxN, borderFunction, valueFunction, niu1 = 4, niu2 = 4):
                self.borderFunction = borderFunction
                self.valueFunction = valueFunction
                self.niu1 = niu1
                self.niu2 = niu2
                self.discrLevel = []
                self.N = maxN

                self.flops = 0

                i = 0
                while(maxN > 2):
                        assert(maxN % 2 == 0)
                        self.discrLevel.append(sed1D.EquationDiscretizer1D(maxN, self.borde
rFunction, self.valueFunction))
                        i += 1
                        maxN /= 2
```

```
        def getCoordinates(self, row, N):
                return int(row / (N + 1)), row % (N + 1)

        def getRow(self, i, j, N):
                return(i * (N + 1) + j)



        def restrict1D(self, r, fineN, coarseN):
                restr = []
                for(i, elem) in enumerate(r):
                        if(i%2 == 0):
                                restr.append(elem)
                return restr

        def restrict(self, r, fineN, coarseN):
                restr = []

                for(i, elem) in enumerate(r):
                        (x, y) = self.getCoordinates(i, fineN)

                        if(x % 2 == 0 and y % 2 == 0):
                                restr.append(elem)

                return restr


        def interpolate1D(self, r, fineN, coarseN):
                interp = []
                for i in range(fineN + 1):
                        if(i%2 == 0):
                                interp.append(r[i/2])
                        else:
                                interp.append((r[(i-1)/2]+r[(i+1)/2])/2.0)
                return interp

        def interpolate(self, r, fineN, coarseN):
                interp = []
                flops = 0
                for i in range((fineN + 1) * (fineN + 1)):
                        (x, y) = self.getCoordinates(i, fineN)

                        if(x % 2 == 0 and y % 2 == 0):
                                index = self.getRow(x / 2, y / 2, coarseN)
                                value = r[index]
                                flops += 1

                        elif(x % 2 == 1 and y % 2 == 0):
                                index1 = self.getRow((x - 1) / 2, y / 2, coarseN)
                                index2 = self.getRow((x + 1) / 2, y / 2, coarseN)
                                value = (r[index1] + r[index2]) / 2.0
                                flops += 2

                        elif(x % 2 == 0 and y % 2 == 1):
                                index1 = self.getRow(x / 2, (y - 1) / 2, coarseN)
                                index2 = self.getRow(x / 2, (y + 1) / 2, coarseN)
                                value = (r[index1] + r[index2]) / 2.0
                                flops += 2

                        else:
                                index1 = self.getRow((x - 1) / 2, (y - 1) / 2, coarseN)
                                index2 = self.getRow((x + 1) / 2, (y - 1) / 2, coarseN)
                                index3 = self.getRow((x - 1) / 2, (y + 1) / 2, coarseN)
                                index4 = self.getRow((x + 1) / 2, (y + 1) / 2, coarseN)
                                value = (r[index1] + r[index2] + r[index3] + r[index4]) / 4
.0
```

```
                                flops += 4

                        if(x == 0 or y == 0 or x == fineN or y == fineN):
                                value = 0

                        interp.append(value)

                self.flops += flops
                return interp

        def restrictTransposeAction(self, r, fineN, coarseN):
                restr = []
                flops = 0

                for i in range((coarseN + 1) * (coarseN + 1)):
                        (x, y) = self.getCoordinates(i, coarseN)
                        (x, y) = (2 * x, 2 * y)
                        newEntry = r[self.getRow(x, y, fineN)]

                        divideFactor = 1.0

                        for (dX, dY) in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
                                newX = x + dX
                                newY = y + dY
                                if(0 <= newX and newX <= fineN and 0 <= newY and newY <= fi
neN):
                                        index = self.getRow(newX, newY, fineN)
                                        newEntry += 0.5 * r[index]
                                        divideFactor += 0.5
                                        flops += 2

                        for (dX, dY) in [(1, 1), (-1, 1), (-1, -1), (1, -1)]:
                                newX = x + dX
                                newY = y + dY
                                if(0 <= newX and newX <= fineN and 0 <= newY and newY <= fi
neN):
                                        index = self.getRow(newX, newY, fineN)
                                        newEntry += 0.25 * r[index]
                                        divideFactor += 0.25
                                        flops += 2

                        newEntry = 1.0 * newEntry / divideFactor
                        if(divideFactor < 4.0):
                                if(not(x == 0 or y == 0 or x == fineN or y == fineN)):
                                        print("Error1")

                        if(x == 0 or y == 0 or x == fineN or y == fineN):
                                newEntry = 0.0

                        restr.append(newEntry)

                self.flops += flops
                return restr


        def vcycle(self, N, level, f, initSol = [], omega = 1.95):
                discr = self.discrLevel[level]
                fSize = len(f)

                if(level == len(self.discrLevel) - 1):
                        v = la.solve(discr.M.todense(), f)
                        return v

                else:
                        solver1 = sm.SolverMethods(
                                iterationConstant = self.niu1,
                                eqDiscretizer = discr,
                                b = f,
```

```
                        initSol = initSol,
                        )

                # omega = 2.0/(1.0 + math.sin(math.pi/N))
                # omega = 1.95
                omega = 1.0
                v, _, _, flops = solver1.SSORIterate(omega)
                self.flops += flops
                if(not(v[0] == 0)):
                        print("ERROR")
                if(not(v[N] == 0)):
                        print("ERROR")
                assert(N % 2 == 0)
                coarseN = N  / 2

                Mv = discr.M.dot(v)
                flops += (2 * discr.M.getnnz() - len(v))

                residual = np.subtract(f, Mv)
                flops += len(f)

                coarseResidual = self.restrict1D(residual, N, coarseN)

                if(not(coarseResidual[0] == 0)):
                        print("ERROR")
                if(not(coarseResidual[coarseN] == 0)):
                        print("ERROR")

                coarseV = self.vcycle(coarseN, level + 1, coarseResidual)

                if(not(coarseV[0] == 0)):
                        print("ERROR")
                if(not(coarseV[coarseN] == 0)):
                        print("ERROR")

                fineV = self.interpolate1D(coarseV, N, coarseN)
                w = np.add(v, fineV)
                flops += len(v)

                solver2 = sm.SolverMethods(
                        iterationConstant = self.niu2,
                        eqDiscretizer = discr,
                        b = f,
                        initSol = w,
                        )

                v2, _, _, _, flops = solver2.SSORIterate(omega)
                self.flops += flops
                return v2

        def iterateVCycles(self, t):
                initSol = []
                N = self.N

                for i in range((N + 1)):
                        x = i
                        if(x == 0 or x == N):
                                initSol.append(self.borderFunction(1.0 * x / N))
                        else:
                                initSol.append(0.0)

                vErrors = []
                discr = self.discrLevel[0]
                f = np.copy(discr.valueVector2D)
                normF = la.norm(f)

                currSol = np.copy(initSol)
```

```
                for i in range(t):
                        residual = np.subtract(f, discr.M.dot(currSol))
                        absErr = 1.0 * la.norm(residual) / (normF)
                        vErrors.append(math.log(absErr))

                        resSol = self.vcycle(N, 0, residual, np.zeros_like(currSol))
                        prevSol = np.copy(currSol)
                        currSol = np.add(currSol, resSol)

                        if(absErr < tol):
                                break
                print(vErrors)
                return currSol, vErrors


class MultiGridAsPreconditioner:

        def __init__(self, borderFunction, valueFunction, maxN, bVector = [], niu1 = 2 , ni
u2 = 2, omega = 1.93):
                self.borderFunction = borderFunction
                self.valueFunction = valueFunction
                self.niu1 = niu1
                self.niu2 = niu2
                self.maxN = maxN
                self.bVector = bVector
                self.discrLevel = []
                self.omega = omega

                self.flops = 0

                i = 0
                while(maxN >= 2):
                        assert(maxN % 2 == 0)
                        self.discrLevel.append(sed.SimpleEquationDiscretizer(maxN, self.bor
derFunction, self.valueFunction))
                        i += 1
                        maxN /= 2


        def getCoordinates(self, row, N):
                return int(row / (N + 1)), row % (N + 1)

        def getRow(self, i, j, N):
                return(i * (N + 1) + j)


        def restrict(self, r, fineN, coarseN):
                restr = []

                for(i, elem) in enumerate(r):
                        (x, y) = self.getCoordinates(i, fineN)

                        if(x % 2 == 0 and y % 2 == 0):
                                restr.append(elem)

                return restr

        def interpolate(self, r, fineN, coarseN):
                interp = []
                flops = 0
                for i in range((fineN + 1) * (fineN + 1)):
                        (x, y) = self.getCoordinates(i, fineN)

                        if(x % 2 == 0 and y % 2 == 0):
                                index = self.getRow(x / 2, y / 2, coarseN)
                                value = r[index]
                                flops += 1
```

```
                        elif(x % 2 == 1 and y % 2 == 0):
                                index1 = self.getRow((x - 1) / 2, y / 2, coarseN)
                                index2 = self.getRow((x + 1) / 2, y / 2, coarseN)
                                value = (r[index1] + r[index2]) / 2.0
                                flops += 2

                        elif(x % 2 == 0 and y % 2 == 1):
                                index1 = self.getRow(x / 2, (y - 1) / 2, coarseN)
                                index2 = self.getRow(x / 2, (y + 1) / 2, coarseN)
                                value = (r[index1] + r[index2]) / 2.0
                                flops += 2

                        else:
                                index1 = self.getRow((x - 1) / 2, (y - 1) / 2, coarseN)
                                index2 = self.getRow((x + 1) / 2, (y - 1) / 2, coarseN)
                                index3 = self.getRow((x - 1) / 2, (y + 1) / 2, coarseN)
                                index4 = self.getRow((x + 1) / 2, (y + 1) / 2, coarseN)
                                value = (r[index1] + r[index2] + r[index3] + r[index4]) / 4
.0

                                flops += 4

                        if(x == 0 or y == 0 or x == fineN or y == fineN):
                                value = 0

                        interp.append(value)

                self.flops += flops

                return interp

        def restrictTransposeAction(self, r, fineN, coarseN):
                restr = []
                flops = 0

                for i in range((coarseN + 1) * (coarseN + 1)):
                        (x, y) = self.getCoordinates(i, coarseN)
                        (x, y) = (2 * x, 2 * y)
                        newEntry = r[self.getRow(x, y, fineN)]

                        divideFactor = 1.0

                        for (dX, dY) in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
                                newX = x + dX
                                newY = y + dY
                                if(0 <= newX and newX <= fineN and 0 <= newY and newY <= fi
neN):

                                        index = self.getRow(newX, newY, fineN)
                                        newEntry += 0.5 * r[index]
                                        divideFactor += 0.5
                                        flops += 2

                        for (dX, dY) in [(1, 1), (-1, 1), (-1, -1), (1, -1)]:
                                newX = x + dX
                                newY = y + dY
                                if(0 <= newX and newX <= fineN and 0 <= newY and newY <= fi
neN):

                                        index = self.getRow(newX, newY, fineN)
                                        newEntry += 0.25 * r[index]
                                        divideFactor += 0.25
                                        flops += 2

                        newEntry = 1.0 * newEntry / divideFactor
                        if(divideFactor < 4.0):
                                if(not(x == 0 or y == 0 or x == fineN or y == fineN)):
                                        print("Error1")

                        if(x == 0 or y == 0 or x == fineN or y == fineN):
                                newEntry = 0.0
```

```
                restr.append(newEntry)

            self.flops += flops
            return restr


    def vcycle(self, N, level, f, initSol = []):

            discr = self.discrLevel[level]
            fSize = len(f)
            if(fSize < 20):
                v = la.solve(discr.M.todense(), f)
                return v

            omega = self.omega
            # omega = 2.0/(1.0 + math.sin(math.pi/N))

            solver1 = sm.SolverMethods(self.niu1, discr, f, initSol)
            v, _, _, _, flops = solver1.SSORIterate(omega)
            self.flops += flops

            assert(N % 2 == 0)
            coarseN = N  / 2

            Mv = discr.M.dot(v)
            self.flops += (2*discr.M.getnnz() - len(v))

            residual = np.subtract(np.array(f), Mv)
            self.flops += len(f)

            coarseResidual = self.restrictTransposeAction(residual, N, coarseN)

            coarseV = self.vcycle(coarseN, level + 1, coarseResidual)
            fineV = self.interpolate(coarseV, N, coarseN)
            v = np.add(v, fineV)
            self.flops += len(v)

            solver2 = sm.SolverMethods(self.niu2, discr, f, v)
            v2, _, _, _, flops = solver2.SSORIterate(omega)
            self.flops += flops
            return v2

    def iterateVCycles(self, N, t):
            self.flops = 0
            initSol = []
            vErrors = []
            discr = sed.SimpleEquationDiscretizer(N, self.borderFunction, self.valueFun
ction)

            if(self.bVector == []):
                f = discr.valueVector2D
            else:
                f = self.bVector

            for i in range(t):
                currSol = self.vcycle(N, 0, f, initSol)

                err = np.subtract(discr.M.dot(currSol), f)
                absErr = np.linalg.norm(err) / np.linalg.norm(f)
                vErrors.append(math.log(absErr))

                if(absErr < tol):
                    break

                initSol = currSol

            return currSol, vErrors, self.flops
```

```
def MultiGridPrecondCG(borderFunction, valueFunction, N, niu1 =1, niu2 = 1, omega = 1.95):
        avoidDivByZeroError = 0.000000000001
        errorDataMGCG = []

        totalFlops = 0
        matrixDots = 0
        vectorAddSub = 0
        vectorDotVector = 0

        mg = MultiGridAsPreconditioner(borderFunction, valueFunction, N, niu1 =niu1, niu2 =
 niu2, omega = omega)
        f = mg.discrLevel[0].valueVector2D
        M = mg.discrLevel[0].M

        x = np.zeros_like(f, dtype = np.float)
        r = np.subtract(f, M.dot(x))
        matrixDots += 1
        vectorAddSub += 1

        mg.bVector = np.copy(r)
        rTilda, _, flops = mg.iterateVCycles(N, 1)
        totalFlops += flops

        rTilda = np.array(rTilda)

        p = np.copy(rTilda)

        convergence = False

        while(not convergence):
                solutionError = np.subtract(M.dot(x), f)
                absErr = 1.0 * np.linalg.norm(solutionError) / np.linalg.norm(f)
                errorDataMGCG.append(math.log(absErr))

                if(absErr < tol):
                        convergence = True
                        break

                Mp = M.dot(p)
                alpha_numerator = rTilda.dot(r)
                alpha_denominator = p.dot(Mp)

                vectorDotVector += 1
                matrixDots += 1

                if(alpha_denominator < avoidDivByZeroError):
                        convergence = True
                        break

                alpha = 1.0 * alpha_numerator / alpha_denominator
                totalFlops += 1

                x = np.add(x, np.multiply(p, alpha))
                vectorAddSub += 1
                totalFlops += len(p)

                newR = np.subtract(r, np.multiply(Mp, alpha))
                totalFlops += len(Mp)

                vectorAddSub += 1

                mg.bVector = np.copy(newR)
                newR_tilda, _, flops = mg.iterateVCycles(N, 1)
                totalFlops += flops

                newR_tilda = np.array(newR_tilda)
```

```
                beta_numerator = newR_tilda.dot(newR)
                beta_denominator = rTilda.dot(r)
                vectorDotVector += 1

                if(beta_denominator < avoidDivByZeroError):
                        convergence = True
                        break

                beta = 1.0 * beta_numerator / beta_denominator
                p = newR_tilda + np.multiply(p, beta)
                totalFlops += 1
                totalFlops += len(p)

                r = newR
                rTilda = newR_tilda

        NNZ = M.getnnz()
        totalFlops += vectorAddSub * len(x) + vectorDotVector * (2*len(x) -1) + matrixDots
* (2* NNZ - len(x))
        return x, absErr, errorDataMGCG, totalFlops



def JacobiPrecondCG(borderFunction, valueFunction, N):
        avoidDivByZeroError = 0.000000000000000000001
        errorDataMGCG = []

        mg = sed.SimpleEquationDiscretizer(N, borderFunction, valueFunction)
        solver = sm.SolverMethods(5, mg)
        f = mg.valueVector2D
        M = mg.M

        x = np.zeros_like(f, dtype = np.float)
        r = np.subtract(f, M.dot(x))

        solver.b = r
        rTilda, _ , _, _= solver.JacobiIterate(0.2)
        rTilda = np.array(rTilda)

        p = np.copy(rTilda)

        convergence = False

        while(not convergence):
                solutionError = np.subtract(M.dot(x), f)
                absErr = 1.0 * np.linalg.norm(solutionError) / np.linalg.norm(f)
                errorDataMGCG.append(math.log(absErr))
                print(absErr)

                if(absErr < tol):
                        convergence = True
                        break

                alpha_numerator = rTilda.dot(r)
                alpha_denominator = p.dot(M.dot(p))

                if(alpha_denominator < avoidDivByZeroError):
                        convergence = True
                        break

                alpha = 1.0 * alpha_numerator / alpha_denominator

                x = np.add(x, np.multiply(p, alpha))

                newR = np.subtract(r, np.multiply(M.dot(p), alpha))

                solver.b = newR
```

```
                newR_tilda, _, _, _ = solver.JacobiIterate(0.2)
                newR_tilda = np.array(newR_tilda)

                beta_numerator = newR_tilda.dot(newR)
                beta_denominator = rTilda.dot(r)

                if(beta_denominator < avoidDivByZeroError):
                        convergence = True
                        break

                beta = 1.0 * beta_numerator / beta_denominator
                p = newR_tilda + np.multiply(p, beta)

                r = newR
                rTilda = newR_tilda

        return x, absErr, errorDataMGCG


#Idea : compute finer solutions as we advance in the timestep
# i.e. 100 iterations for t=0, 120 for t = 1, 140 for t = 2, etc.
def solveHeatEquationForAllTimeSteps(discr):
        solHeat = discr.initialHeatTimeSolution()
        valueVector = discr.computeVectorAtTimestep(1, solHeat)
        solver = sm.SolverMethods(1000, discr, valueVector)
        sol =[solHeat]

        for k in range(1, discr.T + 1):
                t = k * discr.dT
                valueVector = discr.computeVectorAtTimestep(k, solHeat)
                solver.b = valueVector
                (solHeat, err, _) = solver.ConjugateGradientsHS()
                sol.append(solHeat)
                print(err)

        return sol
```