```
import numpy as np

def BVPsolve(xi, xf, yi, yf, h, params, F, solveMethod='Gauss'):
    tVals = np.arange(xi, xf + h, h)
    yVals = np.ones_like(tVals)
    num_interior_nodes = len(tVals) - 2
    bVector = np.zeros(len(tVals) - 2)

    for i in range(num_interior_nodes):
        bVector[i] = F(tVals[i + 1])

    # Define finite difference coefficients for the given equation
    alpha, beta, gamma = params

    # Apply boundary conditions to the first and last elements of bVector
    bVector[0] -= yi * alpha
    bVector[-1] -= yf * gamma
    A = np.zeros((num_interior_nodes, num_interior_nodes))
    A = A + np.diag(beta * np.ones(num_interior_nodes))
    A = A + np.diag(gamma * np.ones(num_interior_nodes - 1), 1)
    A = A + np.diag(alpha * np.ones(num_interior_nodes - 1), -1)

    #Solve matrix
    if solveMethod == 'Seidel':
        x = np.random.random(len(bVector)) * 10*( yi + yf)/2
        interior_nodes = seidel_solve(A, bVector, x)
    else:
        interior_nodes = naive_gauss_elimination(A, bVector)

    #output results including boundary conditions
    output =[yi]
    output.extend(interior_nodes)
    output.append(yf)
    return output, tVals

#from another HW
def seidel_solve(A, b, x, tol=1e-6, max_iter=1000):
    n = len(b)
    iter = 0
    while iter < max_iter:
        previous_x = x.copy()  # Make a full copy of the current solution
        for i in range(n):
            sigma = 0
            for j in range(n):
                if i != j:
                    sigma += A[i][j] * x[j]
            x[i] = (b[i] - sigma) / A[i][i]  # Gauss-Seidel update
        if max(np.abs(np.subtract(previous_x, x))) < tol:
            return x
        iter += 1

    print("Warning: Max iterations exceeded without convergence")
    return x


#from another HW
def naive_gauss_elimination(a,b):
    #size checking:
    m,n = np.shape(a)
    if m != n:
        raise TypeError('A is not a square matrix')
    if len(a) != len(b):
        raise TypeError('A is not the same size as B')

    #Forward elimination
    for k in range(0, n-1):
        for i in range(k+1, n):
            s = a[i][k]/a[k][k]
            for j in range(k, n):
                a[i][j] = a[i][j] - s*a[k][j]
            b[i] = b[i] - s*b[k]

    #Backwards solve
    x = np.zeros(n)
    x[-1] = b[-1]/a[-1][-1]
    for i in range(n-2, -1, -1):
        s = 0
        for j in range( i +1, n):
```

```
        s = s + a[i][j]*x[j]
    x[i] = (b[i] - s)/a[i][i]

return x
```