CSE3365 - Project 3
Analysis of two-point boundary value problems

Eric Smith
Bobby B. Lyle School of Engineering,
Computer Science

May 2017

I. Project Description

This article attempts to demonstrate solving a two-point boundary value problem with several different methods. We will solve the differential equation:

$$Ay'' + By' + Cy = r(x), \text{ where } x \in [a, b] \quad (1)$$

with the two-point boundary conditions:

$$y(a) = \alpha, \; y(b) = \beta \quad (2)$$

To do this, we set up the linear algebraic system:

$$A_h w_h \;=\; r_h \quad (3)$$

using a finite difference scheme. We then solve the resulting system of equations using *Thomas's Algorithm* and the iterative methods of *Jacobi*, *Guass-Seidel*, and *SOR*.

II. Concepts and Theory

To test our solution, we will solve the actual differential equation:

$$y'' + 2y' + y = -4e^x \quad (4)$$

with the boundary conditions:

$$y(a) = -1, \; y(b) = \frac{4}{e - e^2} \quad (5)$$

and test them against our real analytical solution:

$$y(x) = (2e)xe^{-x} - e^x \quad (6)$$

To solve the linear system, we will have to discretize the derivative terms, y' and y'', using finite difference. This method allows us to replace derivatives in a differential equation with approximations. We will solve the resulting algebraic equations to get an approximate solution.

To discretize the first order derivative in our equation, y', we will use central difference formula:

$$y' = \frac{y_{i+1} - y_{i-1}}{2h} \quad (7)$$

Then, to discretize the second order derivative in our equation, y'', we will combine the forward and central difference formulas to get:

$$y'' = \frac{y_{i+1} - 2y_i + y_{i-1}}{h^2} \quad (8)$$

The error of both equations can be described by the asymptotic upper bound *O(h^2)*.

III. Numerical Implementation

Using Equation 1 in the context of our algebraic system, $A_h w_h$ = r_h, we get the discretized form:

$$\begin{aligned} &A(\frac{w_{i+1} - 2w_i + w_{i-1}}{h^2}) + \\ &B(\frac{w_{i+1} - w_{i-1}}{2h}) + Cy = r(x) \end{aligned} \quad (9)$$

Here, A_h is the matrix we will fill to solve the system of equations, w_h is the vector of unknowns, and r_h is the right hand side of our differential equation.

Using Equation 9, we will use the following equations to fill A:

$$w_{i+1} = \frac{A}{h^2} + \frac{B}{2h} \qquad w_{i+1} = \frac{A}{h^2} - \frac{B}{2h}$$
$$w_i = \frac{-2A}{h^2} + C \quad (10)$$

With our tridiagonal matrix in place, we can use Thomas' Algorithm to solve the system. A special case of *LU Factorization*, the algorithm runs in *O(n)* time complexity. In our implementation, we use a series of separate vectors, rather than a matrix, to reduce the memory usage.

Three iterative methods, Jacobi, Gauss-Seidel, and SOR were also used. Here, we used the following discretized equations:

$$Jacobi: \quad w_i^{(k+1)} = \frac{r_i - a_{i-1}w_{i-1}^{(k)} - a_{i+1}w_{i+1}^{(k)}}{a_i} \quad (11)$$

$$G\text{-}S: \; w_i^{(k+1)} = \frac{r_i - a_{i-1}w_{i-1}^{(k+1)} - a_{i+1}w_{i+1}^{(k)}}{a_i} \quad (12)$$

$$SOR: \quad w_i^{(k+1)} = \frac{a_i w_i^{(k)} + w^*(r_i - a_{i-1} w_{i-1}^{(k+1)} - a_i w_i^{(k)} - a_{i+1} w_{i+1}^{(k)})}{a_i} \tag{13}$$

IV. Numerical Results and Discussion

h	$\|\|y_h - w_h\|\|_\infty$
0.5	0.088485
0.25	0.023489
0.125	0.005924
0.0625	0.001485
$\frac{\|\|y_{h-1} - w_{h-1}\|\|_\infty}{\|\|y_h - w_h\|\|_\infty}$	$log(\frac{\|\|y_{h-1} - w_{h-1}\|\|_\infty}{\|\|y_h - w_h\|\|_\infty})$
0.000000	0.000000
3.766950	1.913397
3.964634	1.987187
3.987718	1.995563

Table 1: *Thomas' Algorithm*

Table 1 shows the results of running Thomas' Algorithm with four difference step sizes. As you can see in the second column, error was reduced with the increase in the number of steps. However, even two step sizes was accurate as you can see in Figure 1.

For the iterative methods, the graphs and errors are almost the same. As notable in Table 2, Gauss-Seidel was able to produce the same results as Jacobi in about half the number of iterations. This is because the improved Equation 12 which uses ${w_{i-1}}^{(k+1)}$ at each iteration. Furthermore, SOR maintained the same accuracy, but used much less iterations that Gauss-Seidel, because of its optimal parameter, w^*.

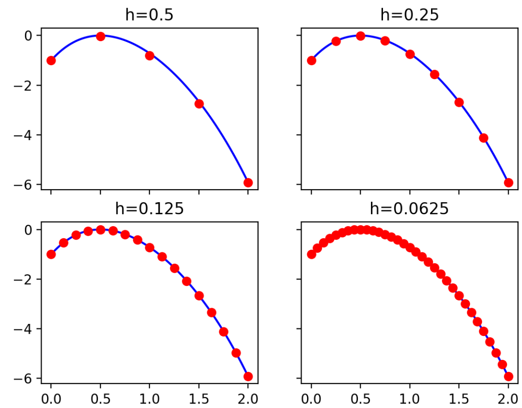

Figure 1: *Thomas' Algorithm*

Case	*Jacobi*	*Gauss-Seidel*	*SOR*
0	55	29	29
1	234	116	82
2	886	430	171
3	3279	1563	335

Table 2: *Iterations of each Iterative Method*

V. Conclusion

In this article, we looked at four different methods for solving a two-point boundary problem. In *Concepts and Theory*, we discussed in detail the problem we were going to solve with the differential equation, boundaries, and parameters. We also discretized the derivatives of our original equation. Then, in *Numerical Implementation*, we laid out the different equations we would use for our iterative methods: Jacobi, Gauss-Seidel, and SOR.

Finally, in *Numerical Results and Discussion*, we saw the accuracy of our solution versus the analytical solution. We also discussed the advantages and disadvantages of our solutions. Our conclusion is as follows: all methods are about equally accurate with Thomas' Algorithm being the most efficient.

Attached you will find the Python code used to calculate the data sets in this article.

```python
"""
Edited by Eric Smith
May 10th, 2017

This function solves a two-point boundary value problem
Ay''+By'+Cy=r, y(a)=alpha, y(b)=beta,
using a finite-difference scheme.
The discreitzed system is solved by direct tridiagonal LU method
and iterative methods including Jacobi, Gauss-Seidel, and SOR
Note:
1. Replace single line *** by single line code and
relace double line *** by multiple lines of code.
2. Python array index start with 0.
"""

import numpy
from numpy import exp, zeros, arange, absolute, max
from math import exp, log, sqrt, cos, pi, sin, pi, e
import matplotlib.pyplot as plt


# solution function
def func(x):
    y = (2 * exp(1)) * x * (exp(-x)) - exp(x)
    return y


# righthand side function
def rfunc(x):
    y = -(4 * exp(x))
    return y


# coefficients of the differential equations
A = 1;
B = 2;
C = 1

# the domain [xa,xb]
xa = 0;
xb = 2

# boudary conditions
alpha = func(xa);
beta = func(xb)

# index for methods: 0:Thomas, 1:Jacobi, 2:Gauss-Sidel, 3: SOR
imethod = 1

# number of cases, each case has different # of unknowns
ncase = 4

# table for results
tbErr = zeros((ncase, 4), float)

# the counter for iterative methods
icount = zeros(ncase, int)


# LU factorization based on Thomas's algorithm
# input: a, b, c, r - matrix elements and right hand side vector
```

```python
# output: w - solution of linear system
def LU3315(a, b, c, r):
    n = len(r)
    w = zeros(n, float)
    l = zeros(n, float)
    u = zeros(n, float)
    z = zeros(n, float)

    # Determine L,U factors
    u[0] = b[0]
    for k in range(1, n):
        l[k] = a[k] / u[k - 1]
        u[k] = b[k] - l[k] * c[k - 1]

    # Solve Lz = r
    z[0] = r[0]
    for k in range(1, n):
        z[k] = r[k] - l[k] * z[k - 1]

    # Solve Uw = z.
    w[n - 1] = z[n - 1] / u[n - 1]
    for k in range(n - 2, -1, -1):
        w[k] = (z[k] - (c[k] * w[k + 1])) / u[k]

    return w


# the main code starts here
fig, ax = plt.subplots(2, 2, sharex=True, sharey=True)

for icase in range(ncase):
    n = 2 ** (icase + 2) - 1  # number of unknowns
    h = (xb - xa) / (n + 1);  # mesh size

    wopt = 2 / (1 + sqrt(1 - cos(pi * h) ** 2))  # optimized omega

    # exact value at a fine mesh
    d = 0.0025
    xe = arange(xa, xb + d, d)
    ye = xe.copy()
    for i in range(len(xe)):
        ye[i] = func(xe[i])

    # matrix entry on tri-dialgonals
    coA = (A / (h ** 2)) - (B / (2 * h))
    coB = ((-2 * A) / (h ** 2)) + C
    coC = A / (h ** 2) + B / (2 * h)

    # claim the vectors needed
    xh = zeros(n, float)  # x-values
    yh = zeros(n, float)  # true y-values at grids
    wh = zeros(n, float)  # computed y-values at grids
    r = zeros(n, float)  # right handside of the equations

    # begin
    # assisgn values for xh,yh,r

    for i in range(0, n):
        xh[i] = xa + ((i + 1) * h)
        yh[i] = func(xh[i])

```

```
    r[0] = rfunc(xh[0]) - (coA * alpha)
    r[n - 1] = rfunc(xh[n - 1]) - (coC * beta)
    for i in range(1, n - 1):
        r[i] = rfunc(xh[i])  # assign values in the middle
    # end

    # vectors needed for direct methods
    if (imethod == 0):
        # Thomas's algorithm
        a = zeros(n, float);
        b = zeros(n, float);
        c = zeros(n, float)
        # assign values for a,b,c
        for i in range(n):
            b[i] = coB
            if (i > 0):
                a[i] = coA
            if (i < n):
                c[i] = coC

        wh = LU3315(a, b, c, r)

    else:
        # Iterative Methods
        tol = 10 ** (-8)
        err = 1  # initial error and error tolerance
        wh1 = zeros(n, float)  # vectors for computed values

        while (err > tol):
            icount[icase] = icount[icase] + 1
            if (imethod == 1):
                # Jacobi
                wh[0] = (r[0] - (coC * wh1[1])) / coB
                for i in range(1, n - 1):
                    wh[i] = (r[i] - coA * wh1[i - 1] -
                             coC * wh1[i + 1]) / coB
                wh[n - 1] = (r[n - 1] - (coA * wh1[n - 2])) / coB

            elif (imethod == 2):
                # Gauss-Seidel
                wh[0] = (r[0] - coC * wh[1]) / coB
                for i in range(1, n - 1):
                    wh[i] = (r[i] -
                             (coA * wh[i - 1]) -
                             (coC * wh[i + 1])) / coB
                wh[n - 1] = (r[n - 1] - coA * wh[n - 2]) / coB
            else:
                # SOR
                wh[0] = (coB *
                         wh[0] + (wopt * (r[0] -
                                          coC * wh[1] -
                                          coB * wh[0]))) / coB
                for i in range(1, n - 1):
                    wh[i] = (coB * wh[i] +
                             (wopt * (r[i] -
                                      coA * wh[i - 1] -
                                      coB * wh[i] -
                                      coC * wh[i + 1]))) / coB
                wh[n - 1] = (coB * wh[n - 1] +
                             wopt * (r[n - 1] -
                                     coA * wh[n - 2] -
```

```
                                         coB * wh[n - 1])) / coB

                err = max(absolute(wh1 - wh))
                wh1 = wh.copy()

        # output
        tbErr[icase, 0] = h
        tbErr[icase, 1] = max(absolute(yh - wh))

        if (icase > 0):
            tbErr[icase, 2] = (tbErr[icase - 1, 1] / tbErr[icase, 1])
            tbErr[icase, 3] = log(tbErr[icase, 2], 2)

        if (imethod != 0):
            print('case ', icase, 'iteration number= ', icount[icase])

        if (icase == ncase - 1):
            print(tbErr)

        # plot: you don't need to change anything here
        xplot = zeros(n + 2, float);
        wplot = zeros(n + 2, float)
        xplot[0] = xa;
        xplot[n + 1] = xb
        wplot[0] = alpha;
        wplot[n + 1] = beta
        for i in range(1, n + 1):
            xplot[i] = xh[i - 1]
            wplot[i] = wh[i - 1]
        kx = int(icase / 2);
        ky = icase % 2

        ax[kx, ky].plot(xe, ye, '-b', xplot, wplot, 'ro')
        ax[kx, ky].set_title('h=' + str(h))

plt.savefig('result.pdf', format='pdf')
plt.show()

```