**Problem Set #4**

Numerical analysis

201621505 채진기

**1-(a)** 코딩한 결과와 소스 코드를 첨부하였습니다.

h를 1, 1/2, 1/3, ... 1/8까지 바꾸어 프로그램을 실행하면서, 어떤 양수 t 값까지

근삿값이 크게 튀지 않고 계산되는지 확인하였습니다. 계산 값들을 정리한 엑셀

파일을 첨부하였고 핵심이 되는 값들은 다음과 같습니다. (값이 튀는 지점)

|  |  |  |
| --- | --- | --- |
| **h** | **t** | **approximation of y** |
| 1 | 5 | -1.33E+21 |
| 1/2 | 6 | -3.1632077212E+23 |
| 1/3 | 10.333 | -8.45E+41 |
| 1/4 | 9.25 | -1.24E+43 |
| 1/5 | 11.4 | -1.64E+55 |
| 1/6 | 13 | -5.45E+12 |
| 1/7 | 15.42857 | -5.68E+101 |
| 1/8 | 17.125 | -5.73E+23 |

**1-(b)**

**1-(c)**

**2~3.**코딩한 결과와 소스 코드를 첨부하였습니다.

다음 자료는 세가지 방법의 근삿값을 비교한 표입니다.

Iteration 횟수를 보면 큰 차이가 있음을 볼 수 있습니다.

Jacobi method 보다는 Gauss-Seidel method 가,

Gauss-Seidel method 보다는 SOR (w = 0.9) method 가 더 efficient 한 근사를

제공하는 것을 볼 수 있습니다.

**4.** 코딩한 결과와 소스 코드를 첨부하였습니다.

다음과 같이 간단한 일반화를 할 수 있습니다.

Let Sol = [0, 0, 0] means that [x, y, z], B = [0.7, 0.5, 1.2]

if i == 0:  
 res = B[i] + 0.1 \* (Sol[1])\*\*(2) - 0.05 \* (Sol[2])\*\*(2)  
 return res  
elif i == 1:  
 res = B[i] -0.3 \* (Sol[0])\*\*(2) + 0.1 \* (Sol[0] \* Sol[2])  
 return res  
else:  
 res = B[i] -0.4 \* (Sol[1])\*\*(2) -0.1 \* (Sol[0] \* Sol[1])  
 return res

일반화된 식을 함수로 정의하고 반복 연산해주면 다음 계산 값을 얻습니다.

[x, y, z] = [0.6597726581087282, 0.4415162558961959, 1.0928953229366771]

Stopping criteria로는 2번 문제와 같은 기준을 두었습니다.

res1 = max([abs(a-b) for a, b in zip(prev, Sol)])  
res2 = max(abs(a) for a in Sol)  
cut = res1 / res2  
if cut < 10 \*\*(-6):  
 break

전 항과의 차이 중 절댓값이 가장 큰 값을 현재 항의 절댓값이 가장 큰 값으로

나눈 것이 10의 -6승 보다 작을 때 멈추게 하였습니다.

**5.**