Approximating Poisson's Equation

Yuuma Odaka, Tom Lambert

February 2018

Contents

1 Introduction

Back in Series 3, we solved lower-upper decompositions of tridiagonal matrices. This time, we want to apply similar algorithms to higher-dimensional problems, more specifically Poisson's differential equation. The equation is:

$$\triangle\varphi = f$$

where $\triangle$ is the Laplace-operator, the sum of the second partial derivatives, f is a given function and φ is unknown. Poisson's equation is not specific to a certain number of dimensions, but in this case we are using two dimensions, so the Laplace Operator is:

$$\triangle u = \frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2}$$

We can solve the equation on an arbitrarily large grid of points in $\mathbb{R}^2$ with the matrix equation $A\hat{u} = b$, where

$$A = \begin{pmatrix} T & -I & 0 & \cdots & 0 \\ -I & T & -I & \cdots & 0 \\ 0 & -I & T & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & T \end{pmatrix} T = \begin{pmatrix} 4 & -1 & 0 & \cdots & 0 \\ -1 & 4 & -1 & \cdots & 0 \\ 0 & -1 & 4 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 4 \end{pmatrix} b = \begin{pmatrix} f_{11}h^2 \\ f_{21}h^2 \\ \vdots \\ f_{(n-1)1}h^2 \\ f_{12}h^2 \\ \vdots \\ f_{(n-1)(n-1)}h^2 \end{pmatrix}$$

with h $= \frac{1}{n}, n \in \mathbb{N}, n > 2$. The vector u is sought.

2 Implementation

We tried the algorithm for several values

2.1 start5.py

All of the experiments can be run by choosing parameters in start5.py and then running it. "b" will calculate the exact values of u and use the SOR method. "c" will show a chart with the maximum error for different n and ε values.
Each plotting algorithm accepts a dictionary with additional configuration values as a second parameter.

2.2 poisson.py

2.2.1 Functions without class

- **rhs**
  Calculates the function's value for a given point in $\mathbb{R}^2$.
- **lgs**
  Generates the linear equation components matrix A and vector b and returns them.
- **exactu**
  Calculates the exact value of the function u at a given point in $\mathbb{R}^2$.

2.2.2 Functions in Iterative Class

- **diskreteLsgSOR**
  Iteratively solves a given linear equation system using the successive-over-relaxation method and returns the results.
- **get_error**
  Compares the result generated by the iterative algorithm with the exact value and returns the difference.

2.3 main5.py

- **plot_b**
  Plots the exact- and the SOR solution as a 3D graph and shows it to the user.
- **plot_c**
  Plots the the maximum error for different n and ε values.

3 Experiments

We applied the algorithm for several integers between 3 and 15. It took over a minute to compute for n = 15, so it is generally infeasible for particulartly large values with ordinary computers. We also varied epsilon, the minimum value before which the iteration could stop.

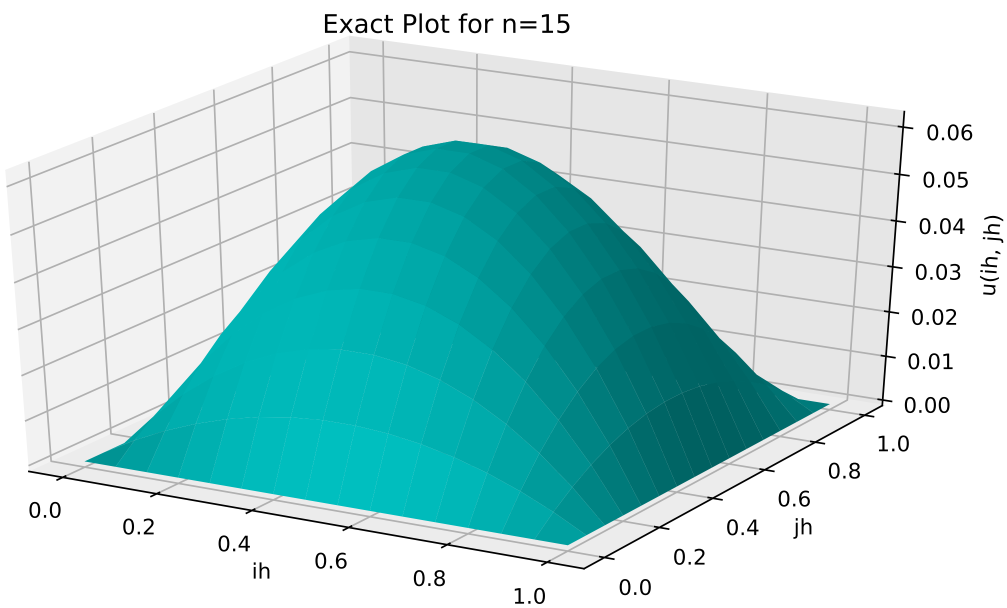
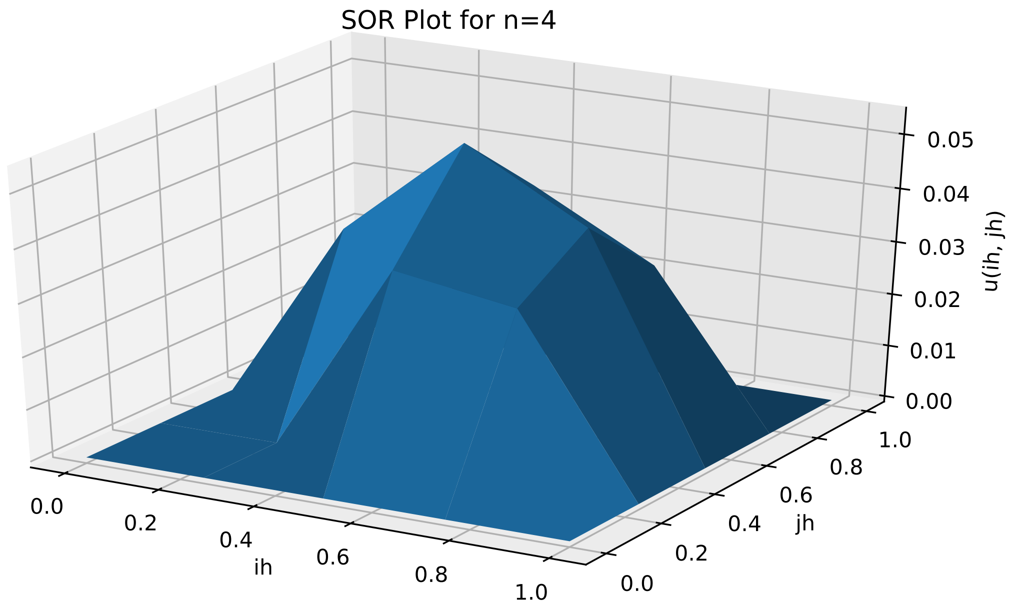
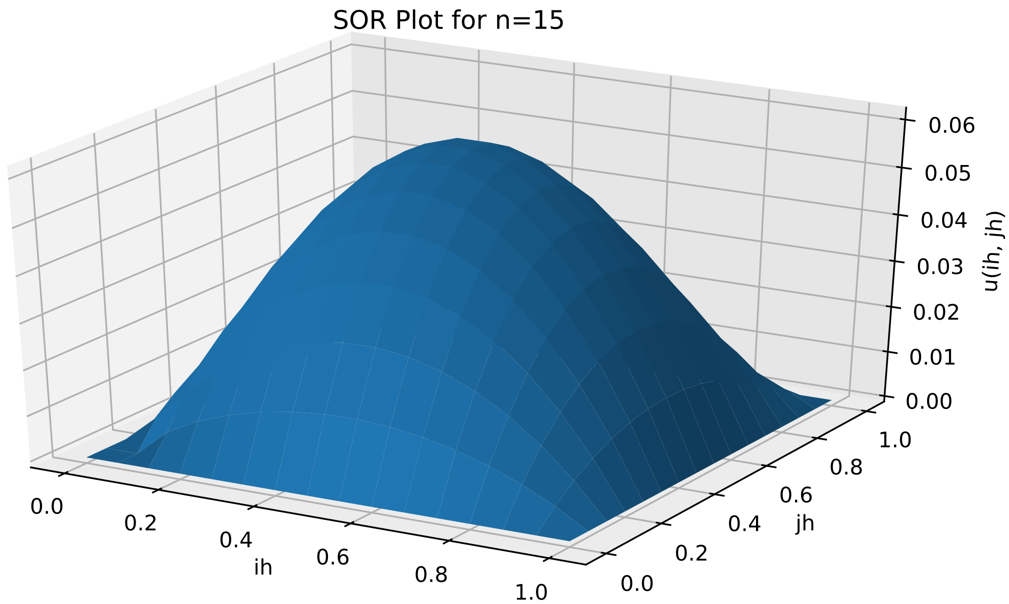


Figure 1: Caption

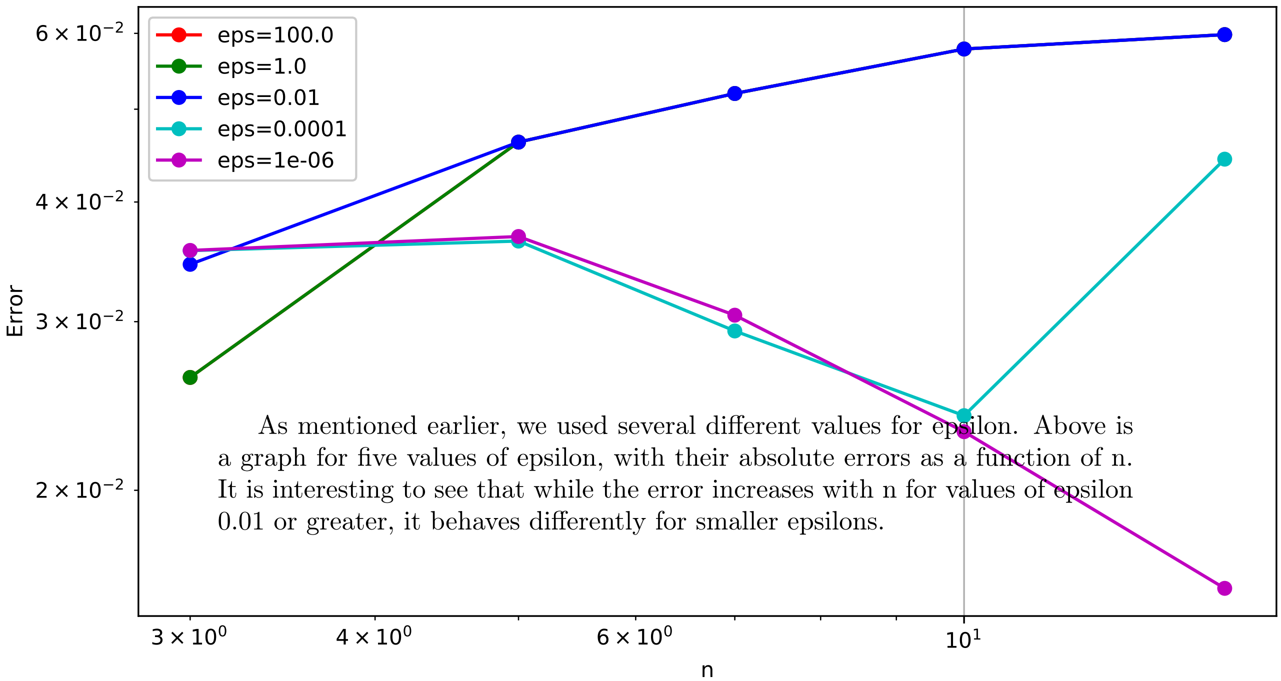
This is the exact solution for the differential equation for n $=15$.



We can see that for n = 4, the our results already have a similarity to the exact values despite the low precision.



For n = 15, the results are almost identical to the exact values. This implies that the SOR method is reasonably accurate for for our given parameters.



As mentioned earlier, we used several different values for epsilon. Above is a graph for five values of epsilon, with their absolute errors as a function of n. It is interesting to see that while the error increases with n for values of epsilon 0.01 or greater, it behaves differently for smaller epsilons.