Numerically Solving the Poisson Equation

Erlend Lima, Frederik J. Mellbye, Aram Salihi, and Halvard Sutterud
University of Oslo, Oslo, Norway
Source code available at: `https://github.com/Caronthir/FYS3150/tree/master/Project1`

(Dated: September 13, 2017)

Three numerical methods of solving the one-dimensional Poisson equation with Dirichlet boundary conditions were investigated. The discretized equation was converted to a linear algebra problem, solved with the common LU-decomposition and the less common Thomas algorithm. The ensuing numerical errors and calculation speeds are discussed. The general and a specialized version of the Thomas algorithm was shown to be much faster and less memory consuming than using a LU decomposition. Moreover, the Thomas algorithm produced solutions with smaller errors than the LU method.

CONTENTS

I. INTRODUCTION

The Poisson equation is amongst the most important and widely used differential equations used in science. Many important differential equations can be re-written to the form

$$\nabla^2 u = f$$

by variable substitutions. It is therefore crucial to develop algorithms that can easily solve these types of equations, preferably with small errors.

By discretizing the Poisson equation it can be written as a tridiagonal matrix equation, and solving the equation turns into a simpler linear algebra problem. The matrix equation can easily be solved using the famous LU-decomposition, but because of the tridiagonal nature of the matrix, a more efficient (but conceptually equal) approach to solve the problem is possible using the Thomas algorithm.

First the general version of this algorithm is implemented, after which a specially tailored version is developed for the particular tridiagonal matrix considered in this paper. As the algorithms become more specialized, more terms and factors can be precomputed, which saves memory space, and more importantly reduces computation speeds and the size of the numerical errors.

II. THEORY

A. Discretizing the Poisson Equation

The one-dimensional Poisson equation with Dirchelet boundary conditions is

$$\frac{d^2u}{dx^2} = -f$$

The second derivative of $u(x)$ can be approximated by Taylor expanding $u(x)$ and solving for $\frac{d^2u}{dx^2}$:

$$\frac{d^2u}{dx^2} \approx \frac{u(x+h) + u(x-h) - 2u(x)}{h^2} + O(h^2)$$

See A for the derivation. The equation is discretized with grid points $x_1, x_2, \ldots, x_n$. Imposing Dirichlet boundary conditions forces $x_1 = x_n = 0$, using the handy notation $u(x_i + h) = u_{i+1}$ and inserting the approximated second derivative into the Poisson equation yields

$$\frac{u_{i+1} + u_{i-1} - 2u_i}{h^2} = -f_i$$

where $h = \frac{1}{n+1}$ is the step size, i.e. the distance between grid points, and $f_i = f(x_i)$. This discretized version of the one dimensional Poisson equation is conveniently a linear system of equations, and can therefore be written as the matrix equation

$$A\mathbf{u} = h^2\mathbf{f}$$

where

$$A = \begin{bmatrix} 2 & -1 & 0 & \dots & \dots & 0 \\ -1 & 2 & -1 & 0 & & \vdots \\ 0 & -1 & 2 & -1 & 0 & \\ \vdots & 0 & -1 & 2 & -1 & 0 \\ & & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & \dots & 0 & -1 & 2 \end{bmatrix}$$

is a $n \times n$ tridiagonal matrix, the only non-zero elements are on, directly above or directly below the diagonal. Row reducing the matrix can therefore be done in an efficient way using the Thomas algorithm, as we shall see in the next paragraph.

B. Solving a General Tridiagonal Matrix Problem

A general tridiagonal matrix equation is on the form

$$\begin{bmatrix} b_1 & c_1 & 0 & \dots & \dots & 0 \\ a_2 & b_2 & c_2 & 0 & & \vdots \\ 0 & a_3 & b_3 & c_3 & 0 & \\ \vdots & 0 & a_4 & b_4 & c_4 & 0 \\ & & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & \dots & 0 & a_n & b_n \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ \\ u_n \end{bmatrix} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \\ \\ v_n \end{bmatrix}$$

where $v_i = h^2 f_i$ in the case examined in this project. To get the matrix in row reduced echelon form and solve the problem, the a_i's are first eliminated by Gaussian elimination. First, $\frac{a_2}{b_1}$ times the first row is subtracted from the second. This gives

$$\begin{bmatrix} b_1 & c_1 & 0 & \dots & \dots & 0 \\ 0 & b'_2 & c_2 & 0 & & \vdots \\ 0 & a_3 & b_3 & c_3 & 0 & \\ \vdots & 0 & a_4 & b_4 & c_4 & 0 \\ & & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & \dots & 0 & a_n & b_n \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_n \end{bmatrix} = \begin{bmatrix} v_1 \\ v'_2 \\ v_3 \\ \vdots \\ v_n \end{bmatrix}$$

where $b'_2 = b_2 - \frac{a_2}{b_1} c_1$ and $v'_2 = v_2 - \frac{a_2}{b_1} v_1$. For the next row, exactly the same operation is used, but notice how the previous elements b'_2 and v'_2 now have been updated in the last iteration. Generalizing this argument for each row, a general expression for the diagonal elements is

$$b'_i = b_i - \frac{a_i}{b'_{i-1}} c_{i-1}$$

where $b'_1 = b_1$ and $i = 2, 3, ..., n$. Similarly, elements in the right-hand side vector $\mathbf{v}'$ are modified to

$$v'_i = v_i - \frac{a_i}{b'_{i-1}} v'_{i-1}$$

where $v'_1 = v_1$ and $i = 2, \dots, n$. After completing the forward substitution, the matrix equation reads

$$\begin{bmatrix} b'_1 & c_1 & 0 & \dots & \dots & 0 \\ 0 & b'_2 & c_2 & 0 & & \vdots \\ 0 & 0 & b'_3 & c_3 & 0 & \\ \vdots & 0 & 0 & b'_4 & c_4 & 0 \\ & & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & \dots & 0 & 0 & b'_n \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_n \end{bmatrix} = \begin{bmatrix} v'_1 \\ v'_2 \\ v'_3 \\ \vdots \\ v'_n \end{bmatrix}$$

By inspection, the solution for the final point u_n is explicitly given by the bottom row:

$$b'_n u_n = v'_n \rightarrow u_n = \frac{v'_n}{b'_n}$$

The rest of the solution $u(x_i)$ is then computed by backward substitution, given u_i one can compute u_{i-1}. For instance, row $n-1$ reads

$$b'_{n-1} u_{n-1} + c_{n-1} u_n = v'_{n-1}$$
$$u_{n-1} = \frac{v'_{n-1} - c_{n-1} u_n}{b'_{n-1}}$$

In general then, the back substitution is done by

$$u_i = \frac{v'_i - c_i u_{i+1}}{b'_i}$$

where $i = n-1, n-2, \dots, 1$. Some details were taken from [1].

C. Optimizing the Thomas Algorithm for a Specific Case

In the case presented in this project, $a_i = c_i = -1$ and $b_i = 2$. The fact that there are just two different values in the matrix greatly simplifies the calculations, since all the calculations involving only a_i, b_i and c_i can be done beforehand. Also, it is possible to derive an analytic expression for b'_i. This is given by

$$b'_i = \frac{i+1}{i}$$

and is found and proved by induction in section B.

D. Solving the General Tridiagonal Matrix Problem by LU-decomposition

Another way to solve matrix equations is LU-decomposing A, and then solving two matrix equations. This is a frequently used method for solving more general sets of linear equations. The LU-decomposition is a form of Gaussian elimination, where A is factored into matrices L and U. L is lower triangular with 1 in each element on the diagonal, and U is upper triangular. The matrix equation is

$$A\mathbf{u} = \mathbf{v}$$
$$LU\mathbf{u} = \mathbf{v}$$

Introducing $\mathbf{y} = U\mathbf{u}$, the solution $\mathbf{u}$ can now be computed in two steps. First, $\mathbf{y}$ is found by solving

$$L\mathbf{y} = \mathbf{v}$$

which is then inserted in

$$U\mathbf{u} = \mathbf{y}$$

and solved for $\mathbf{u}$.

E. Counting the Number of FLOPS for Each Method

A simple way of determining the computational speed of an algorith is simply counting the number of floating point operations. The fewer calculations the computer has to make, the faster an algorithm will run. In this section approximate numbers for the amount of floating point operations (flops) are determined.

1. The General Thomas Algorithm

The general algorithm with arbitrary tridiagonal elements consists of two steps, the forward substitution (two equations) and the backward substitution (one equation). In the forward substitution, some of the operations can be precomputed. Since a_i, c_i is known for all i considered, the product $a_i c_{i-1}$ can be precomputed and does not provide any flops. Therefore, computing a b'_i requires only two operations, and these are done $(n-1)$ times. Computing a v'_i requires three flops, one product, one division and an addition. These operations are also repeated $(n-1)$ times, so the total amount of flops in the forward substitution is:

$$2(n-1) + 3(n-1) = 5(n-1)$$

In the backward substitution, no quantities can be precomputed as they are produced by the algorithm itself. Counting the number of operatons yields $3(n-1)$ flops. In total, then, the number of flops required in the general Thomas algorithm is

$$\text{flops}_{\text{Thomas}} = 8(n-1)$$

2. The Special Case Thomas Algorithm

For the specialized algorithm, even more quantities can be precomuted and the amount of flops can therefore be reduced further. First of all, an analytic expression for b'_i, the diagonal elements, does not contribute with any flops as they can now be fully precomputed. The computation of v'_i now only requires two operations, since $a_i = -1$ simply changes the sign. The backward substitution now consists of one subtraction and one division operation, so the total amount of flops for the tailored algorithm is

$$4(n-1) \tag{1}$$

This should therefore cut the running time of the algorithm approximately in half.

3. The LU Decomposition Method

Compared to the Thomas algorithm, the LU decomposition requires a much larger amount of flops. Where the Thomas neatly avoids dealing with all the $\sim (n^2 - 3n)$ zeros, the LU algorithm consideres every element in every row of the matrix. It is possible to show that the number of flops for this method is in the order $\frac{2}{3}N^3$. This should become apparent when the computation times are analyzed, but the time difference of the LU method compared to the Thomas methods might be smaller because the armadillo functions are optimized to work faster.

F. Memory Usage

Since all of the discretized values need to be stored in memory, it is convenient to consider the memory required to run the programs. For the Thomas methods, arrays with lengths of order n are used to store the arrays. The LU decomposition stores values in a $n \times n$ matrix, so the number of entries that need to be stored is of order n^2. Therefore, it will be possible to solve much larger systems of equations with the Thomas algorithm (of course this requires the matrix to be tridiagonal) than with the LU decomposition.

G. Analytic Solution for a Specific Problem

In this paper the following special case of the Poisson equation is examined:

$$-\frac{d^2u}{dx^2} = 100e^{-10x} \tag{2}$$

where $0 < x < 1$ and $u(0) = u(1) = 0$, i.e. Dirchelet boundaries. An analytic solution is obtained by integrating with respect to x twice

$$u(x) = -e^{-10x} + A + Bx$$

Imposing the boundary condition $x(0) = 0$ gives $u(0) = 0 = -1 + A$, so $A = 1$. Applying $x(1) = 0$ gives $0 = -e^{-10} + 1 + B$, so $B = e^{-10} - 1$. Inserting the integration constants returns the analytical solution

$$u(x) = 1 - (1 - e^{-10})x - e^{-10x} \tag{3}$$

where $0 < x < 1$ and $u(0) = u(1) = 0$.

III. METHOD

The algorithms were implemented in C++ and tested using Google Test. The source code is available in the attached GIT-repository. The calculations were performed several times, and for each method the algorithm times were averaged to produce an average time. The implementations are explained and listed in the following paragraphs with pseudocode. The analysis of the solutions and errors was done in Python.

A. The General Thomas Algorithm

The general Thomas algorithm consists of the forward and backward substitutions examined in the theory section. In pseudocode the loop was implemented as

```
b_prime[0] = b[0];
v_prime[0] = v[0];

// Forward substitution
for(i = 1; i <= n-1; i++)
    b_prime[i] = b[i] - (a[i]/b_prime[i
        -1])*c[i-1];
    v_prime[i] = v[i] - (a[i]/b_prime[i
        -1])*v_prime[i-1];
}
// Backward substitution
for(i = n-2; i >= 1; i--){
    u[i] = (v_prime[i] - c[i]*u[i+1])/
        b_prime[i];
}
```

Listing 1: Thomas algorithm implementation for a general tridiagonal matrix

B. The Optimized Algorithm

Because of the fact that the matrix examined in this paper is both tridiagonal and symmetric, further optimizations to the algorithm can be made. Since $a_i = c_i = -1$ and $b'_i = (i+1)/i$ can be precomputed, the algorithm simplifies to

```
// b_prime is now precomputed
for(i = 1; i <= n-1; i++)
    b_prime[i] = (i+1)/i;

v_prime[0] = v[0];

// Forward substitution
for(i = 1; i <= n-1; i++)
    v_prime[i] = v[i] + (v_prime[i-1]/
        b_prime[i-1]);
}
// Backward substitution
for(i = n-2; i >= 1; i--){
    u[i] = (v_prime[i] + u[i+1])/b_prime
        [i];
}
```

Listing 2: Optimized version of Thomas algorithm for a symmetric tridiagonal matrix

C. The LU-Decomposition

Solving the matrix equation using the LU-decomposition method was done with functions from the Armadillo C++ Linear Algebra Library. First, the tridiagonal matrix A is implemented, and then the Armadillo functions `lu()` and `solve()` are used to solve the two factored matrix equations. In pseudocode, the implementation is

```
#include <armadillo>

for (row = 0; row < size-1; row++){
    A(row, row) = 2;
    if (row > 0)
        A(row, row-1) = -1;
    if (row < size-1)
        A(row, row+1) = -1;

lu(L,U,A);
y = solve(L, btilde);
u = solve(U, y);
```

Listing 3: Pseudocode of LU-decomposition method implementation

D. Error Analysis

The relative error of the numerical solution at x_i is given by

$$\epsilon_i = \left| \frac{v_i - u_i}{u_i} \right|$$

where v_i is the numerically calculated value and u_i is the analytic value at x_i. The error ϵ_i is calculated for multiple values of n, from which the maximum error $\epsilon_{\mathrm{n,max}} = \max \epsilon_i$ is saved and plotted against the stepsize $h = \frac{1}{n+1}$. The general method was used for calculating the numerical solution.

E. Timing of the Algorithms

In order to get an accurate timing of the algoritms, each computation was repeated 10^6 times where this was practical. For the general and special algorithms, this limit was at matrices of size 10^4. For matrices of size 10^{5-7}, as well as for all matrices using LU-decomposition, the computations were repeated 10^3 times. The final estimate of their time usage was taken as their averages.

IV. RESULTS

A. Solution Using the Algorithms

All of the three methods converged to the solution, albeit somewhat differently. The general method and the LU-decomposition converged upwards while the special method converged downwards. Figure 1 shows a plot where the general method is plotted using a differing number of points n, while an equivalent plot using the specialized method is shown in figure 2.

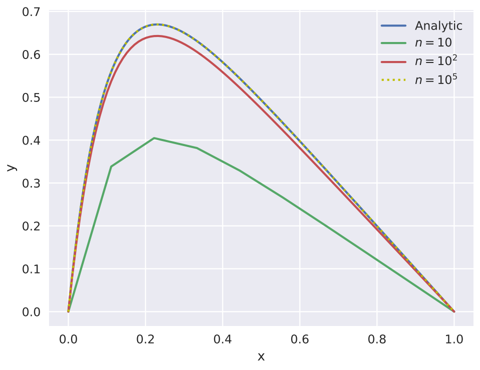

Figure 1: The convergence of the numerical method is apparent as its graph approaches the analytical as the number of points n grows. When $n = 10^5$, shown as a dashed yellow line, the numerical solution becomes indistinguishable from the analytical solution.

B. Error Analysis

The relative error produced by the general method is shown in figure 3, where it is plotted against stepsize on a log-log scale.

C. Timing the Algorithms

The time used by each algorithm is summarized in table I.

Table I: Time usage of each algorithm. These are averages of the time used by each algorithm. Data is lacking for LU decomposition for matrices larger than 10^3 as the memory consumption became too great.

N	Time usage $[s]$		
	General	Special	LU
10^1	1.592×10^{-7}	1.298×10^{-7}	1.378×10^{-5}
10^2	1.683×10^{-6}	1.555×10^{-6}	9.927×10^{-3}
10^3	1.687×10^{-5}	1.549×10^{-5}	6.214×10^{-1}
10^4	1.682×10^{-4}	1.555×10^{-4}	-
10^5	1.734×10^{-3}	1.599×10^{-3}	-
10^6	1.749×10^{-2}	1.611×10^{-2}	-
10^7	2.413×10^{-1}	2.340×10^{-1}	-

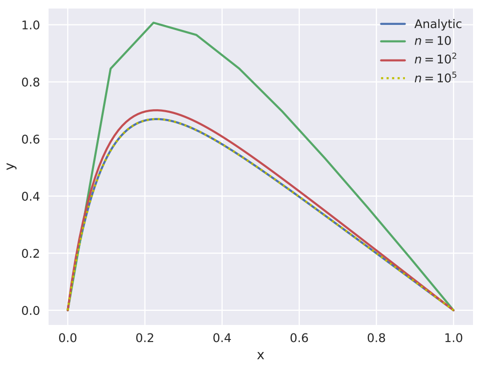

Figure 2: The convergence of the special method. Note how the graphs approach the solution from above.

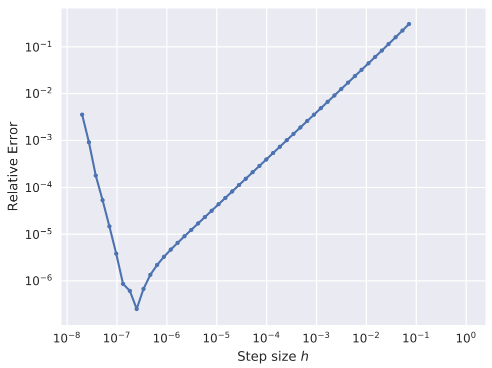

Figure 3: The relative error plotted against the step size, produced with the general method.

V. DISCUSSION

All three methods for computing the solution to the Poisson equation converged to the known analytical solution as expected. It was noted that the general method and LU-decomposition behaved differently than the specialized method. A likely reason for this is that some of the algorithms might have small implementation errors. This error, most likely in the special algorithm, was not discovered, even after extensive attempts to locate the bug and re-implementing the method. Because the method still rapidly converges to the analytic solution, albeit in a different way, the method is still valid as it is implemented. However, because of the different convergence patterns, slight optimizations are most likely available in one of the implementations of the methods, since they should produce the same results, even for larger step sizes.

Also, the LU decomposition was not possible for matrices where $n \geq 10^4$, because the method requires more memory than what was available. On the other hand, the Thomas algorithm variations ran for $n = 10^8$ before memory issues appeared.

A. Error Analysis

The convergence towards the analytical solution is also evident from figure 3, as n increases. As n is further increased, a minimum error is reached before further lowering of the step size introduces rapidly increasing trunctation/round-off errors.

B. Algorithm Execution Speeds

Solely in terms of the timings (see table I), the Thomas methods are far superior to the LU decomposition, by around the same order as predicted by counting the number of FLOPS. Among the Thomas algorithm variations, the special algorithm is slightly faster, but not by a factor 2 as predicted in the theory section. There are several possible causes for this. For instance, compiler optimizations might speed up the general method more than the special method, and the way the arrays are passed between functions might be slower than a more optimal implementation.

VI. CONCLUSION

As a concluding remark, it is evident that carefully examining the problem at hand, and making several simplifications created an algorithm that solved the problem faster, for much larger matrices and with remarkably smaller errors. The Thomas algorithm is essentially the same as the LU-decomposition method, but with all the redundant operations (for instance operations including all the zeros) removed.

Furthermore the methods developed in this project are now readily available to be used to solve any equation that can be transformed into the one-dimensional Poisson equation.

It is also evident that an important factor in this and similar problems is carefully choosing an ideal step size, to balance out numerical errors and the trunctation error. If one is to ignore these effects, the solutions might include errors of several orders, there is no way to tell as there is no analytic solution to compare to.

[1] M. Hjort-Jensen, "Computaional physics lectures: Linear algebra methods," `https://compphysics.github.io/ComputationalPhysics/doc/pub/linalg/html/linalg.html` (2017).

Appendix A: Second Derivative

Section 2 introduced an approximated expression for the second derivative. To derive this expression, consider a general Taylor expansion of a real valued function $u(x)$:

$$u(x) = \sum_{n=0}^{\infty} \frac{u^n(a)}{n!}(x-a)^n$$

The Taylor expansion of $u(x+h)$ and $u(x-h)$ around $a = x$ of third order is therefore

$$u(x+h) \approx u(x) + h\frac{\mathrm{d}u}{\mathrm{d}x} + \frac{h^2}{2!}\frac{\mathrm{d}^2u}{\mathrm{d}x^2} + \frac{h^3}{3!}\frac{\mathrm{d}^3u}{\mathrm{d}x^3} - \frac{h^4}{4!}\frac{\mathrm{d}^4u}{\mathrm{d}x^4}|_{\xi+}$$
$$u(x-h) \approx u(x) - h\frac{\mathrm{d}u}{\mathrm{d}x} + \frac{h^2}{2!}\frac{\mathrm{d}^2u}{\mathrm{d}x^2} - \frac{h^3}{3!}\frac{\mathrm{d}^3u}{\mathrm{d}x^3} + \frac{h^4}{4!}\frac{\mathrm{d}^4u}{\mathrm{d}x^4}|_{\xi-}$$

where $\xi_\pm \in [x-h, x+h]$.

Adding these two expression cancels out the first and third derivatives, leaving

$$u(x+h) + u(x-h) \approx 2u(x) + \frac{\mathrm{d}^2u}{\mathrm{d}x^2}h^2 + \frac{h^4}{12}\frac{\mathrm{d}^4u}{\mathrm{d}x^4}|_\xi$$

Solving for $\frac{\mathrm{d}^2u}{\mathrm{d}x^2}$ then gives

$$\frac{\mathrm{d}^2u}{\mathrm{d}x^2} \approx \frac{u(x+h) + u(x-h) - 2u(x)}{h^2} - \frac{h^2}{12}\frac{\mathrm{d}^4u}{\mathrm{d}x^4}|_\xi$$

By letting $h \to 0$ this expression approaches the an exact expression for the second derivative.

$$\frac{\mathrm{d}^2u}{\mathrm{d}x^2} = \lim_{h\to 0}\frac{u(x+h) + u(x-h) - 2u(x))}{h^2} - \frac{h^2}{12}\frac{\mathrm{d}^4u}{\mathrm{d}x^4}|_\xi$$

The error term is of order $\mathcal{O}(h^2)$.

Appendix B: Deriving a General Expression for the Diagonal Elements after Forward Substituting

The general expression for the diagonal elements in A after forward substituting was

$$b'_i = b_i - \frac{a_i}{b'_{i-1}c_i}$$

for $i = 2, \ldots, n$. Since $a_i = c_i$ for all i in the special case examined in this paper, the expression for b'_i simplifies to

$$b'_i = b_i - \frac{1}{b'_{i-1}}$$

In an attempt at further simplify the expression, a few iterations are calculated, and a pattern seems to emerge:

$$\begin{aligned} b'_1 &= b_1 = 2 \\ b'_2 &= b_2 - \frac{1}{b'_1} = 2 - \frac{1}{2} = \frac{3}{2} \\ b'_3 &= b_3 - \frac{1}{b'_2} = 2 - \frac{2}{3} = \frac{4}{3} \\ b'_4 &= b_4 - \frac{1}{b'_3} = 2 - \frac{3}{4} = \frac{5}{4} \end{aligned}$$

By inspection, b'_i seems to follow the pattern

$$b'_i = \frac{i+1}{i}$$

which is assumed valid for $i = 1, 2, \ldots, n$. A proof by induction is used to show that the formula is indeed correct for all $i \geq 1$.

For any integer $i \geq 1$, let $S(i)$ denote the statement

$$S(i) : b'_i = 2 - \frac{1}{b'_{i-1}} = \frac{i+1}{i}$$

where we have defined $b'_1 = 2$.

In the case $i = 1$, $S(1)$ is obviously true because b'_1 is defined as $b'_1 = b_i = 2$. Now, for some fixed $j \geq 1$, assume that $S(j)$ holds. Then, b_{j+1} is given by

$$b'_{j+1} = 2 - \frac{1}{b'_j}$$

But since $S(j)$ holds, $b'_j = \frac{j+1}{j}$, and

$$b'_{j+1} = 2 - \frac{j}{j+1} = \frac{j+2}{j+1} = \frac{(j+1)+1}{(j+1)}$$

which proves $S(i)$ is true for all $i \geq 1$.