Jacobi Iterative Solution of Poisson's Equation in 1D

John Burkardt
Department of Scientific Computing
Florida State University
.....
http://people.sc.fsu.edu/∼jburkardt/presentations/jacobi_poisson_1d.pdf

November 8, 2011

Abstract

This document investigates the use of a Jacobi iterative solver to compute approximate solutions to a discretization of Poisson's equation in 1D. The document is intended as a record and guide for a particular investigation into this problem. Therefore, we specify a particular set of data that represents an instance of the Poisson equation; we discuss the form of a discretization of the equation which results in a linear system; we consider a specific implementation of the Jacobi iterative method that was used to solve the linear system; we then consider the convergence behavior of the iterative method as the size of the grid increases and look for an alternative solution procedure that will give us the answer more efficiently. The expectation is that the multigrid method will enable us to solve the 1D problem more quickly, and to proceed to the 2D problems that are of greater interest.

1 The Poisson Equation in 1D

We consider a 1D domain, in particular, a closed interval $[a, b]$, over which some forcing function $f(x) \in C[a, b]$ has been specified. Now consider the following differential equation, which is the 1D form of *Poisson's equation*:

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

We say that the function $u \in C^2[a, b]$ is a solution if it satisfies Poisson's equation for every value x in (a, b). (The behavior of $u(x)$ at the endpoints a and b will be regarded momentarily.)

If $u(x)$ is such a solution, then so is any function of the form $u(x) + c + d * x$ where c and d are arbitrary constants. But mathematically, it is desirable that a problem have a unique solution, so generally, Poisson's equation is posed with additional constraints. The most common constraint is to impose Dirichlet boundary conditions at the left and right endpoints, so that the solution $u(x)$ is also required to satisfy:

$$u(a) = u_a$$
$$u(b) = u_b$$

for some given values u_a and u_b. It is also possible to replace one of the Dirichlet conditions by a Neumann condition involving the value of the first derivative, such as, perhaps

$$\frac{du}{dx}(a) = p_a$$

although, again for uniqueness reasons, it is generally not possible to replace both Dirichlet boundary conditions by Neumann conditions.

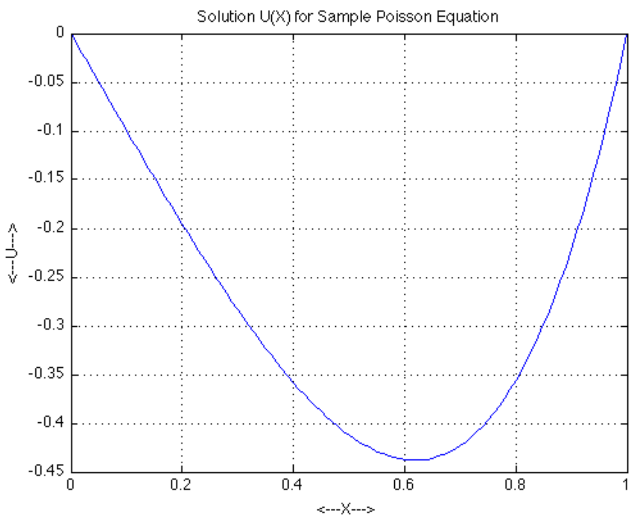

Figure 1: The Exact Solution to the Sample Poisson Equation.

In the interest of brevity, from this point in the discussion, the term "Poisson equation" should be understood to refer exclusively to the Poisson equation over a 1D domain with a pair of Dirichlet boundary conditions.

2 Data for the Poisson Equation in 1D

We now wish to consider a specific example of the Poisson equation, in which we specify the remaining data. As our discussion continues into the questions of discretization and solution methods, this is the problem we will refer to when we wish to have a specific example.

The problem data for the example includes:

- **the interval**: $[a, b] = [0, 1]$;
- **boundary conditions**: $u(0) = u(1) = 0$;
- **forcing term**: $f(x) = -x * (x + 3) * e^x$;
- **exact solution**: $u(x) = x * (x - 1) * e^x$;

3 Discretizing the Poisson Geometry

A discretization of the Poisson equation begins by choosing a discretization of the geometry. In the 1D case, we select a grid of N points or *nodes* in the interval $[A, B]$. While it can be advantageous to vary the spacing of these points, we will choose them uniformly. While it is common to omit the endpoints, since the Dirichlet boundary conditions supply the solution value there, we will nonetheless always include the endpoints, and when we set up the linear system, we will include equations associated with the endpoints, as though the solution was unknown there as well.

It is necessary to keep the multigrid method in mind while designing the grid. In fact, we will want to be able to make a nested sequence of grids, with the k-th grid having n_k nodes, so that

$$n_k = 2^k + 1, \text{ for } k = 0, ...$$

Since we assume the nodes are equally spaced between A and B, inclusively, and since we know the number of nodes in each grid, we can display a formula for the location of the j-th node in the k-th grid:

$$x_j = \frac{((n_k - j) * A + (j-1) * B)}{n_k - 1}, \text{ for } j = 1, ..., n_k;$$

In the k-th grid, there will be n_k subintervals $[x_j, x_{j+1}]$. We assume the nodes are evenly spaced. The uniform length of the intervals is known as the *mesh spacing* or *mesh size*. Using the given rule for the number of nodes in each grid, the k-th grid will have a mesh spacing

$$h_k = \frac{b-a}{2^k}, \text{ for } k = 0, ...$$

The discrete analog to the solution function $u(x)$ is a solution vector u_j, with a value given at each node x_j of the mesh. A pair of discrete nodes and values associated in this way is sometimes called a *mesh function*. We can plot this solution vector, and as an aid to visualization, connect successive points (x_j, u_j) to suggest a continuous curve, but our solution procedure will really only be producing finitely many points.

4 Discretizing the Poisson Operator

In order to complete our discretization of the Poisson equation, we must replace the negative second derivative or "Poisson operator" by some function that can be applied to a mesh function.

For convenience, we will designate our mesh function by (x_j, u_j). Let us suppose that the values u_j are actually derived from a function $u(x)$ defined over the entire interval, so that $u_j = u(x_j)$. Moreover, let us assume that $u(x) \in C^2[a, b]$. If the nodes are equally spaced by h, then the difference quotient $\frac{u_{j+1}-u_j}{h}$ is an estimate for the derivative $u'(x)$ at the midpoint $\frac{x_{j+1}+x_j}{2}$ to the right of x_j, while the difference quotient $\frac{u_j-u_{j-1}}{h}$ is an estimate for the derivative $u'(x)$ at the midpoint $\frac{x_j+x_{j-1}}{2}$ to the left of x_j.

We can regard the second derivative as the derivative of the first derivative, and so it makes sense to approximate the second derivative as a difference of the approximations to the first derivative. Indeed, if we compute the difference of the two first differences, to the left and right of x_j, it turns out we get a good approximation to the second derivative at x_j:

$$\begin{aligned} -\frac{d^2u}{dx^2}(x_j) \approx & -\frac{\frac{u_{j+1}-u_j}{h} - \frac{u_j-u_{j-1}}{h}}{h} \\ = & \frac{-u_{j-1} + 2u_j - u_{j+1}}{h^2} \end{aligned}$$

This difference quotient is our discrete approximation to the Poisson operator.

5 The Discretized Poisson System

We now write down a system of equations associated with our nodes x_j and involving the values of u_j, a vector of data that we regard as a pointwise approximation to the solution of the original Poisson equation. Our first and last equations are simply the boundary conditions. The remaining equations express the discretized version of the Poisson equation at each of the nodes. The typical equation at node x_j will be

$$\frac{-u_{j-1} + 2u_j - u_{j+1}}{h^2} = f(x_j)$$

If we multiply all but the first and last equation by h^2, we have:

$$\begin{aligned}
u_1 =& u_a \\
-u_1 + 2u_2 - u_3 =& h^2 f(x_2) \\
-u_2 + 2u_3 - u_4 =& h^2 f(x_3) \\
& ... \\
-u_{n-2} + 2u_{n-1} - u_n =& h^2 f(x_{n-1}) \\
u_n =& u_b
\end{aligned}$$

and it is easy to see that this constitutes an n by n linear system for the unknowns u_j.

This scaling reveals a simple structure to the matrix for any grid index k. However, we will find that we should not use this scaling when solving a sequence of problems for different grid indices. Were we to use the scaling, then as the grid index k rises, we are scaling the residuals for later grids by a factor h_k^2 that is going to zero. This would have the unfortunate effect that a fixed convergence tolerance **tol** could not be used for the sequence of linear systems. Therefore, if convenient, we will temporarily suppress the h_k^2 divisor when discussing the linear system matrix, but in those cases, we will identify the matrix as $h^2 * A$.

6 Properties of the Linear System

The linear system defined above can be represented as $A * u = f$ where u is the n-vector of solution values at the nodes, f contains the boundary conditions in its first and last entries, and otherwise the values of the forcing function at the nodes. The interesting object is the scaled matrix $h^2 * A$, which has the form

$$\begin{vmatrix}
1 & 0 & 0 & 0 & ... & 0 & 0 & 0 \\
-1 & 2 & -1 & 0 & ... & 0 & 0 & 0 \\
0 & -1 & 2 & -1 & ... & 0 & 0 & 0 \\
... & ... & ... & ... & ... & ... & ... & ... \\
0 & 0 & 0 & 0 & ... & -1 & 2 & -1 \\
0 & 0 & 0 & 0 & ... & 0 & 0 & 1
\end{vmatrix}$$

By inspection, it is clear that $h^2 * A$ is *banded* and *symmetric.* The matrix has another important property that is not immediately obvious: it is *positive definite.* Note that the matrix retains these properties whether or not we scale it by h^2. The fact that our system matrix is symmetric and positive definite has important implications when we look at our options for solving the linear system.

7 Direct or Iterative Solution

Once we have chosen a grid size n, we can define the linear system $A * u = f$, and we have only to solve this linear system in order to obtain the vector u. The natural approach to the linear system would be a direct linear solver, based on Gauss elimination. However, direct linear solvers have some disadvantages in terms of data storage and the cost of computing the solution. Moreover, these disadvantages can become extreme as the problem size n grows. For this reason, it is worth considering alternatives to a direct solver; linear systems arising from the discretized Poisson problem, in particular, are well known to be amenable to other solution techniques.

An alternative approach is to try an iterative solver. An iterative solver produces a sequence of approximations to the solution of a linear system. Each iterate is very cheap to compute, and it is often the case that approximations improve quickly enough that the total iteration cost incurred while getting an acceptable approximate solution is still significantly less than that for a direct solver.

One of the simplest iterative solvers for linear systems is known as the *Jacobi iteration.* The satisfactory use of an iterative solver generally requires that the linear system satisfy some conditions; in the case of the Jacobi iteration, it is enough that the matrix A is positive definite and symmetric. The Jacobi iteration is an easy iteration to implement and study; we will be able to solve small problems with it, but when we begin to explore larger linear systems, we will see that we will need a more powerful iterative solver.

8 The Jacobi Iteration

Let us assume that we start with a linear system $A * x = f$ and an initial approximate solution x^0. We can also let x^0 be the zero vector if no better first approximation comes to mind. The Jacobi iteration can be thought of as a procedure which makes a simple correction to each entry of an approximate solution to produce a somewhat better approximate solution.

To compute the i-th element of the new approximate solution x^1, we "solve" the i-th equation for the corrected value of the i-th variable:

$$x_i^1 = \frac{f_i - \sum_{j=1;j\neq i}^{n} A_{i,j} * x_j^0}{A_{i,i}}$$

If x^0 was an exact solution, this process would not change any entries.

The update of each element of the new approximate solution is independent of the others. If we decompose the matrix as $A = L+D+U$, with L, D and U representing the lower triangle, diagonal and upper triangular submatrices, we can also write

$$x^1 = D^{-1} * (f - (L+U) * x^0)$$

a concise vectorized format that is very suitable for rapid calculation in a programming language such as MATLAB.

The Jacobi iteration, obviously, consists of starting with an initial approximation x^0, and repeatedly applying the Jacobi update, creating a sequence $x^0, x^1, x^2, ...$ which converges to the exact solution.

9 Convergence Test for the Jacobi Iteration

Although we can expect our sequence of Jacobi iterates to converge to the exact solution, we don't know what that solution is, so we cannot measure the solution error directly. There are several things we can measure, however, in order to control the iteration.

The weakest control is simply to impose a maximum iteration limit. While it may be a good idea to forbid the iteration to exceed some huge number of iterations, it is important that such a control only be used to catch an error condition. It is possible, as we will see, for a Jacobi iteration to be carried out a million times without achieving convergence. Therefore, an iteration that is terminated by the iteration limit should be regarded as having failed.

A control that pays more attention to the behavior of the approximations to the solution checks the change between successive iterates. Thus, as soon as x^1 is computed, one computes $||x^1 - x^0||$. In order to make this measure comparable across different mesh counts, it is preferable to use the RMS version of this measure, that is $\frac{||x^1-x^0||}{\sqrt{n}}$. However, as we will see, when a Jacobi iteration is converging very slowly, it is easily possible for the RMS norm of the iterate difference to go to zero long before the iterates themselves are good approximations to the true solution.

The control that makes sense to apply to the iteration checks the *residual*, that is, having computed the j-th iterate x^j, we define the residual r^j by: defined by

$$r^j = A * x^j - f$$

and control error using the RMS norm $\frac{||r^j||}{\sqrt{n}}$.

The advantage of looking at the residual error is that you guarantee that if your residual norm is small, then your approximate solution satisfies the equations, on average, to the given tolerance. Notice that this does not say that our approximate solution is actually that close to the true solution, or even that it is close at all to the true solution. However, as long as we don't actually know the true solution, monitoring the residual is the proper way to control and terminate an iteration.

10 A Sample Calculation

We can examine the results of our discretization and interative approximations for the sample problem with a grid index $k = 5$, resulting in n_k=33 points. Our Jacobi iteration will use an RMS residual tolerance of 0.000001. We compare the exact solution to the continuous problem against the solution to the discretized problem computed directly and with the Jacobi iteration.

The closeness of the results suggest that the spatial discretization is fine enough that we are getting good approximations to the exact solution of the continuous problem. Moreover, the approximations produced by the Jacobi iteration are very close to those computed directly. On the other hand, this iteration required 3,088 steps, which might seem a surprisingly high cost. We will consider this question in more detail shortly.

```
  I         X        U_Exact    U_Direct    U_Jacobi

  1      0.0000          -0  -1.402e-15           0
  2      0.0312    -0.03123    -0.03121    -0.03121
  3      0.0625    -0.06237    -0.06233    -0.06233
  4      0.0938    -0.09331    -0.09325    -0.09325
  5      0.1250     -0.1239     -0.1239     -0.1239
  6      0.1562     -0.1541      -0.154      -0.154
  7      0.1875     -0.1838     -0.1837     -0.1837
  8      0.2188     -0.2127     -0.2126     -0.2126
  9      0.2500     -0.2408     -0.2406     -0.2406
 10      0.2812     -0.2678     -0.2677     -0.2677
 11      0.3125     -0.2937     -0.2935     -0.2935
 12      0.3438     -0.3181      -0.318      -0.318
 13      0.3750      -0.341     -0.3408     -0.3408
 14      0.4062     -0.3621     -0.3619     -0.3619
 15      0.4375     -0.3812      -0.381      -0.381
 16      0.4688     -0.3979     -0.3977     -0.3977
 17      0.5000     -0.4122      -0.412      -0.412
 18      0.5312     -0.4236     -0.4234     -0.4234
 19      0.5625     -0.4319     -0.4317     -0.4317
 20      0.5938     -0.4368     -0.4366     -0.4366
 21      0.6250     -0.4379     -0.4377     -0.4377
 22      0.6562     -0.4348     -0.4346     -0.4346
 23      0.6875     -0.4273     -0.4271     -0.4271
 24      0.7188     -0.4148     -0.4146     -0.4146
 25      0.7500     -0.3969     -0.3968     -0.3968
```

```
   26      0.7812     -0.3733     -0.3731     -0.3731
   27      0.8125     -0.3433     -0.3432     -0.3432
   28      0.8438     -0.3065     -0.3064     -0.3064
   29      0.8750     -0.2624     -0.2623     -0.2623
   30      0.9062     -0.2103     -0.2102     -0.2102
   31      0.9375     -0.1496     -0.1496     -0.1496
   32      0.9688    -0.07976    -0.07973    -0.07973
   33      1.0000           0           0           0
```

11 Jacobi Convergence With Increasing N

A fundamental technique of computational science is to look at the discretization process as involving a discretization scale h, which might represent the maximum size of an interval. In that case, we can imagine that we have the option of choosing a sequence of scaled sizes h_k, such that $\lim_{k\to\infty} h_k = 0$, or, equivalently, an increasing sequence of mesh counts n_k such that $\lim_{k\to\infty} n_k = \infty$, which we expect will produce a sequence of increasingly better approximations to the continuous solution $u(x)$.

While we will not address that fundamental convergence issue yet, we do note that it is a natural procedure to seek to improve an approximation at mesh count n_k by comparing with the solution at the next larger mesh count n_{k+1}, and to note various statistics and properties of the corresponding solutions.

Because we are producing our solution of the linear system using the Jacobi iteration with a fixed tolerance, one natural statistic to consider is the number m_k of such iterations required to meet the specified tolerance at mesh count n_k. Roughly speaking, the cost of computing the approximate solution u_k will be of the order of $c_k = o(m_k * n_k)$, that is, the number of iterations times the cost, during one iteration, of updating each of n_k solution entries. It is clear that the computational cost must grow at least linearly. To make a stronger statement, we must understand what to expect for the behavior of m_k with increasing n_k.

We can easily create a table for our example problem, using a tolerance of 1.0E-10 that is applied to the RMS residual norm:

k	n_k	m_k
0	2	1
1	3	1
2	5	43
3	9	188
4	17	768
5	33	3,088
6	65	12,373
7	129	49,520
8	257	198,122
9	513	792,553
10	1025	3,170,329

It is easy to see from this table that as n_k doubles, m_k tends to increase by about a factor of 4, which in turn, means that we can expect the computational cost c_k to increase by a factor of 8. This suggests that the computational cost depends on the grid size by a relationship of the form

$$c_k \sim o(n_k^3)$$

For such a cost growth rate, the algorithm will quickly become infeasible with increasing n. What is worse, we are so far only considering a 1D problem, which we expect might have more tractable cost growth rates than the higher dimensional cases we are also interested in. We clearly need to reconsider the use of an iterative solution of the linear system by Jacobi's method, if we hope to solve problems on a fine grid, or extend this approach to higher-dimensional geometries, or cases in which a nonlinearity means that we need to solve many linear systems efficiently.

12 Sample Program

We include here a sample MATLAB program which was used to carry out the convergence study for the example problem. The program takes as input the value **k**, which is the grid index, defines a problem of the corresponding size, and runs the Jacobi iteration until the convergence tolerance is achieved.

A copy of the program is available at
http://people.sc.fsu.edu/∼jburkardt/m_src/jacobi_poisson_1d/jacobi_poisson_1d.m

```
function jacobi_poisson_1d ( k )

%*****************************************************************************80
%
%% JACOBI_POISSON_1D uses Jacobi iteration for the 1D Poisson equation.
%
%  Parameters:
%
%    Commandline input, integer K, the grid index.
%    K specifies the number of nodes, by the formula NK = 2^K + 1.
%
}}
  fprintf ( 1, '\n' );
  fprintf ( 1, 'JACOBI_POISSON_1D:\n' );
  fprintf ( 1, '  Use Jacobi iteration for the 1D Poisson equation.\n' );
%
%  Set boundaries.
%
  a = 0.0;
  b = 1.0;
%
%  Set boundary conditions.
%
  ua = 0.0;
  ub = 0.0;
%
%  Get NK.
%
  nk = 2^k + 1;
%
%  Set XK.
%
  xk = ( linspace ( a, b, nk ) )';
%
%  Get HK.
%
  hk = ( b - a ) / ( nk - 1 );
```

```
%
%  Set FK.
%
  fk = force ( xk );
  fk(1) = ua;
  fk(nk) = ub;
%
%  Set the -1/2/-1 entries of A.
%
%  In order that the operator A approximation the Poisson operator,
%  and in order that we can compare linear systems for successive grids,
%  we should NOT multiply through by hk^2.
%
%  Though it is tempting to try to "normalize" the matrix A, the
%  unintended result is to scale our right hand side a multiplicative
%  factor of hk^2, which means that we make it easier and easier to
%  satisfy the RMS residual tolerance, as NK increases, with solution
%  vectors that are actually worse and worse.
%
  sup  = sparse ( 2:nk-1, 3:nk,   -1.0, nk, nk );
  diag = sparse ( 2:nk-1, 2:nk-1,  2.0, nk, nk );
  sub  = sparse ( 2:nk-1, 1:nk-2, -1.0, nk, nk );
  A = ( sup + diag + sub ) / hk^2;
  A(1,1) = 1.0;
  A(nk,nk) = 1.0;
%
%  Just because we can, ask MATLAB to get the exact solution of the linear system
%  directly.
%
  udk = A \ fk;
%
%  Sample the solution to the continuous problem.
%
  uek = exact ( xk );
%
%  Use Jacobi iteration to solve the linear system to the given tolerance.
%
  ujk = zeros ( nk, 1 );
  tol = 0.000001;

  [ ujk, it_num ] = jacobi ( nk, A, fk, ujk, tol );
%
%  Examine errors:
%
  fprintf ( 1, '\n' );
  fprintf ( 1, '  Using grid index K = %d\n', k );
  fprintf ( 1, '  System size NK was %d\n', nk );
  fprintf ( 1, '  Tolerance for linear residual %g\n', tol );
  fprintf ( 1, '  Number of Jacobi iterations required was %d\n', it_num );
  fprintf ( 1, '  RMS Jacobi error in solution of linear system = %g\n', ...
    norm ( udk - ujk ) / sqrt ( nk ) );
  fprintf ( 1, '  RMS discretization error in Poisson solution = %g\n', ...
    norm ( uek - ujk ) / sqrt ( nk ) );

  fprintf ( 1, '\n' );
  fprintf ( 1, '     I        X        U_Exact    U_Direct    U_Jacobi\n' );
  fprintf ( 1, '\n' );
  for i = 1 : nk
    fprintf ( 1, '  %4d  %10.4f  %10.4g  %10.4g  %10.4g\n', ...
      i, xk(i), uek(i), udk(i), ujk(i) );
  end
%
%  Terminate.
%
  fprintf ( 1, '\n' );
  fprintf ( 1, 'JACOBI_POISSON_1D:\n' );
  fprintf ( 1, '  Normal end of execution.\n' );

  return
end
function uex = exact ( x )

%*****************************************************************************80
%
%% UEX evaluates the solution of the continuous problem.
%
  uex = x .* ( x - 1 ) .* exp ( x );

  return
end
function f = force ( x )

%*****************************************************************************80
%
%% FORCE evaluates the forcing term.
%
  f = - x .* ( x + 3 ) .* exp ( x );

  return
end
function [ u, it ] = jacobi ( n, A, f, u, tol )

%*****************************************************************************80
%
%% JACOBI carries out the Jacobi iteration.
%
  fprintf ( 1, '\n' );
  fprintf ( 1, '    Step    Residual      Change\n' );
  fprintf ( 1, '\n' );

  it = 0;
  while ( 1 )

    u_old = u;
    u = ( f - A * u_old + ( diag ( A ) .* u_old ) ) ./ diag ( A );
    r = A * u - f;
    it = it + 1;

    fprintf ( 1, '  %6d  %10.4g  %10.4g\n', ...
      it, norm ( r ) / sqrt ( n ), norm ( u - u_old ) / sqrt ( n ) );

    if ( norm ( r ) / sqrt ( n ) <= tol )
      break;
    end

  end

  return
end
```

References

[1] Richard Barrett, Michael Berry, Tony Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roidan Pozo, Charles Romine, Henk van der Vorst, **Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods**, SIAM, 1994.

[2] William Briggs, Van Emden Henson, Steve McCormick, **A Multigrid Tutorial**, SIAM, 2000.

[3] Howard Elman, Alison Ramage, David Silvester, **Finite Elements and Fast Iterative Solvers with Applications in Incompressible Fluid Dynamics**, Oxford, 2005.