MATH 561: Numerical Analysis I

Instructor: Prof. Wolfgang Bangerth
bangerth@colostate.edu

Answers for homework assignment 3

Problem 1 (Convergence of Richardson iteration). This method's operator has the form $T = I - BA = I - \omega A$ because $B = \omega I$. The eigenvectors of T are – as is easy to verify, the same as those of A, and the eigenvalues of T are $\lambda_i(T) = 1 - \omega\lambda_i(A)$. To prove that the method converges, it is convenient to verify that the l_2 norm of T is less than one because we can then express everything in terms of the eigenvalues of T. Recall that the l_2 norm of a square, invertible matrix equals its largest eigenvalue by magnitude. Now, since A is positive definite, all of its eigenvalues are positive. As a consequence, with $\omega > 0$, the eigenvalues of T all satisfy $\lambda_i(T) = 1 - \omega\lambda_i(A) < 1$, but without a condition on ω, they can be arbitrarily negative. To make sure that they are less than one *by magnitude*, we need to require that $\omega < 2/\lambda_{\max}(A)$ since then $-1 < \lambda_i(T) = 1 - \omega\lambda_i(A) < 1$, and consequently $\|T\|_{l_2} < 1$. This condition ensures convergence of the Richardson method.

In practice, one often finds that as problems grow, the largest eigenvalues of matrices also grow. This means that we will need to choose ω smaller and smaller as problems grow larger and larger, and in applications this typically leads to slower and slower convergence of the method. (Recall that if we set $\omega = 0$, then the iteration stays where it is and makes no progress at all.) Richardson's method is therefore more of theoretical interest and not typically used in practice.

Problem 2 (Condition numbers). The condition number is defined as $\kappa(A) = \|A\|\,\|A^{-1}\|$, in the matrix norm you choose to use. The matrix in question here has an inverse that is easy to compute, and then you only need to remember that the l_1 norm is the maximum column sum, and the l_∞ norm is the maximum row sum.

With this, it is easy to verify that $\kappa_1(A) = 4.004 \cdot 10^6$ and $\kappa_\infty(A) = 4.004 \cdot 10^6$.

The l_2 norm condition number requires only slightly more work. The l_2 norm of a matrix is equal to the largest eigenvalue by magnitude (or, more generally, the square root of the largest eigenvalue of A^TA). For the given matrix, the eigenvalues are approximately equal to 2 and $5 \cdot 10^{-7}$. Thus, $\|A\|_{l_2} = 2$. Because the matrix is invertible, the eigenvalues of A^{-1} are $\lambda_i(A^{-1}) = 1/\lambda_i(A)$, so we do not even have to compute A^{-1} if we already know the eigenvalues of A – the l_2 norm of A^{-1} is simply $\|A^{-1}\|_{l_2} = 1/\lambda_{\min}(A) = (5 \cdot 10^{-7})^{-1} = 2 \cdot 10^6$. Consequently, $\kappa_{l_2}(A) = 2 \cdot 2 \cdot 10^6 = 4 \cdot 10^6$.

Problem 3 (Error propagation). The solutions of the two problems are $x = A^{-1}b = (-1000, 1000)^T$ and $\tilde{x} = A^{-1}\tilde{b} = (-2000, 2000)^T$. That is, of course, a rather drastic difference for such a small change in the right hand side! In fact, while $\|b - \tilde{b}\|_{l_1} = 0.001$, we have that $\|x - \tilde{x}\|_{l_1} = 2000$, i.e., even though the difference in the right hand sides is only 10^{-3}, the difference in the solutions is around 10^6 times larger!

However, it is quite within the bounds that we derived in class that state that

$$\frac{\|x - \tilde{x}\|_{l_1}}{\|x\|_{l_1}} \le \kappa_{l_1}(A) \frac{\|b - \tilde{b}\|_{l_1}}{\|b\|_{l_1}}.$$

Problem 4 (Matrix norms). The first part is pretty simple: The induced norm of a matrix is defined by

$$\|A\| = \max_{v \in \mathbb{R}^n, v \neq 0} \frac{\|Ax\|}{\|x\|}$$

and if we apply this to the identity matrix $A = I$, we get

$$\|I\| = \max_{v\in\mathbb{R}^n, v\neq 0} \frac{\|Ix\|}{\|x\|} = \max_{v\in\mathbb{R}^n, v\neq 0} \frac{\|x\|}{\|x\|} = \max_{v\in\mathbb{R}^n, v\neq 0} 1 = 1.$$

It is simple to construct counter examples to this statement for non-induced norms. Indeed, let $\|A\|$ be *any* matrix norm (induced or not) of a matrix A with $\|I\| = 1$, then it is easy to verify that $\|A\|' = 2\|A\|$ is *also* a norm because it satisfies the three norm axioms. However, we will then have $\|I\|' = 2$. It is easy to convince yourself that the matrix norm $\|A\|'$ is not induced by a vector norm that is simply twice another vector norm.

A slightly more constructive example would be the Frobenius norm $\|A\|_F = \sqrt{\sum_i \sum_j |a_{ij}|}$. It yields $\|I\|_F = \sqrt{n}$ for $n \times n$ matrices.

Problem 5 (Jacobi iteration). The Gauss-Seidel iteration of the previous homework converged pretty quickly, so it is easy to run it for, say, 500 iterations and just write out the solution vector. With this, we can then run the following slight variation of the Jacobi program from last week's homework:

```
#include <iostream>
#include <cmath>
#include <stdlib.h>

const unsigned int N = 100;

double x[N];
double residual[N];

double b[N];

const double exact_solution[N] =
{
      0.0440078737,  0.0884558262,  0.132535005,  0.175452634,  0.216443544,
      0.254781354,   0.289789124,   0.320849314,  0.347412865,  0.369007265,
      0.385243468,   0.39582154,    0.400534955,  0.399273452,  0.392024416,
      0.378872752,   0.359999251,   0.335677471,  0.306269187,  0.272218462,
      0.234044451,   0.192333032,   0.147727406,  0.100917809,  0.0526304912,
      0.00361614596, -0.0453620379, -0.0935405097, -0.140167488, -0.184514895,
      -0.225889915,  -0.263645981,  -0.297193036, -0.326006889, -0.349637532,
      -0.367716279,  -0.379961624,  -0.386183713, -0.386287372, -0.380273637,
      -0.368239766,  -0.350377727,  -0.326971195, -0.298391096, -0.265089776,
      -0.227593882,  -0.186496074,  -0.142445691, -0.0961385186, -0.0483058328,
      0.000297127082, 0.0489030582, 0.0967446876, 0.143066865,  0.187138465,
      0.228263914,    0.265794149,  0.299136854,  0.327765796,  0.351229116,
      0.369156457,    0.381264797,  0.387362913,  0.387354391,  0.381239145,
      0.369113418,    0.351168261,  0.327686515,  0.299038355,  0.265675446,
      0.228123821,    0.186975581,  0.142879561,  0.0965310903, 0.0486610319,
      2.42514473e-05, -0.0486122865, -0.096481615, -0.142828861, -0.18692315,
      -0.228069133,  -0.265617956,  -0.298977486, -0.32762166,  -0.35109877,
      -0.369038598,  -0.381158247,  -0.387266606, -0.387267363, -0.381160527,
      -0.369042423,  -0.351104179,  -0.327628706, -0.29898624,  -0.265628505,
      -0.228081584,  -0.186937626,  -0.142845507, -0.0965005981, -0.0486337963
};
```

```
// compute the l2-norm difference between the current iterate 'x'
// and the exact solution of which we have the values above
double evaluate_error ()
{
  double sum_of_squares = 0;

  for (unsigned int i=0; i<N; ++i)
    sum_of_squares += std::pow((x[i]-exact_solution[i]), 2.0);

  return std::sqrt (sum_of_squares);
}




int main ()
{
  // initialize x with random values between
  // 0 and 1. set b to the values given in
  // the problem description
  for (unsigned int i=0; i<N; ++i)
    {
      x[i] = 1.*rand() / RAND_MAX;
      b[i] = std::sin(2.*3.14159265358*i/50)/100;
    }

  // do the iterations
  for (unsigned int it = 0; it<=2000; ++it)
    {
      // compute the residual -(Ax-b) as
      // described above
      residual[0] = - (2.01*x[0] - x[1] - b[0]);
      for (unsigned int i=1; i<N-1; ++i)
        residual[i] = - (2.01*x[i] - x[i-1] - x[i+1] - b[i]);
      residual[N-1] = - (2.01*x[N-1] - x[N-2] - b[N-1]);

      // then update x
      for (unsigned int i=0; i<N; ++i)
        x[i] = x[i] + residual[i]/2.01;

      // at the end of each iteration output the error
      std::cout << it << ' ' << evaluate_error() << std::endl;
    }
}
```

The results are shown in the figure for Problem 8 below.

Problem 6 (Gauss-Seidel and Symmetric Gauss-Seidel iterations). This problem works the same way as the previous one. The following variation of the code from the last homework achieves the objective:

```
#include <iostream>
```

```
#include <cmath>
#include <stdlib.h>

const unsigned int N = 100;

double x[N];
double residual[N];

double b[N];

const double exact_solution[N] =
{
      0.0440078737,  0.0884558262,   0.132535005,   0.175452634,   0.216443544,
      0.254781354,   0.289789124,    0.320849314,   0.347412865,   0.369007265,
      0.385243468,   0.39582154,     0.400534955,   0.399273452,   0.392024416,
      0.378872752,   0.359999251,    0.335677471,   0.306269187,   0.272218462,
      0.234044451,   0.192333032,    0.147727406,   0.100917809,   0.0526304912,
      0.00361614596, -0.0453620379, -0.0935405097, -0.140167488,  -0.184514895,
      -0.225889915,  -0.263645981,  -0.297193036,  -0.326006889,  -0.349637532,
      -0.367716279,  -0.379961624,  -0.386183713,  -0.386287372,  -0.380273637,
      -0.368239766,  -0.350377727,  -0.326971195,  -0.298391096,  -0.265089776,
      -0.227593882,  -0.186496074,  -0.142445691,  -0.0961385186, -0.0483058328,
      0.000297127082, 0.0489030582,  0.0967446876,  0.143066865,   0.187138465,
      0.228263914,    0.265794149,   0.299136854,   0.327765796,   0.351229116,
      0.369156457,    0.381264797,   0.387362913,   0.387354391,   0.381239145,
      0.369113418,    0.351168261,   0.327686515,   0.299038355,   0.265675446,
      0.228123821,    0.186975581,   0.142879561,   0.0965310903,  0.0486610319,
      2.42514473e-05, -0.0486122865, -0.096481615, -0.142828861,  -0.18692315,
      -0.228069133,  -0.265617956,  -0.298977486,  -0.32762166,   -0.35109877,
      -0.369038598,  -0.381158247,  -0.387266606,  -0.387267363,  -0.381160527,
      -0.369042423,  -0.351104179,  -0.327628706,  -0.29898624,   -0.265628505,
      -0.228081584,  -0.186937626,  -0.142845507,  -0.0965005981, -0.0486337963
};


// compute the l2-norm difference between the current iterate 'x'
// and the exact solution of which we have the values above
double evaluate_error ()
{
  double sum_of_squares = 0;

  for (unsigned int i=0; i<N; ++i)
    sum_of_squares += std::pow((x[i]-exact_solution[i]), 2.0);

  return std::sqrt (sum_of_squares);
}



int main ()
{
```

```
  // initialize x with random values between
  // 0 and 1. set b to the values given in
  // the problem description
  for (unsigned int i=0; i<N; ++i)
    {
      x[i] = 1.*rand() / RAND_MAX;
      b[i] = std::sin(2.*3.14159265358*i/50)/100;
    }

  // do the iterations
  for (unsigned int it = 0; it <=2500; ++it)
    {
      // update the solution vector as discussed in class, by just
      // overwriting the previous value
      //
      // in the following, we need to recall that whenever we use the
      // element a_ii, we know that it is 2.01; similarly, the
      // *immediate* off-diagonal entries are -1, and all other matrix
      // entries are zero. this makes multiplying with the matrix
      // relatively straightforward, except that we have to pay
      // attention to the first and last row
      x[0]   = 1./2.01 * (b[0] - (-1 * x[1]));
      for (unsigned int i=1; i<N-1; ++i)
        x[i] = 1./2.01 * (b[i] - (-1 * x[i-1]) - (-1 * x[i+1]));
      x[N-1] = 1./2.01 * (b[N-1] - (-1 * x[N-2]));

      // at the end of each iteration output the error
      std::cout << it << ' ' << evaluate_error() << std::endl;
    }
}
```

Again, the results are shown in the figure for Problem 8 below.

The symmetric variant is just a simple variation on the source code that either does a forward or backward sweep over all variables:

```
#include <iostream>
#include <cmath>
#include <stdlib.h>

const unsigned int N = 100;

double x[N];
double residual[N];

double b[N];

const double exact_solution[N] =
{
        0.0440078737, 0.0884558262, 0.132535005, 0.175452634, 0.216443544,
        0.254781354, 0.289789124, 0.320849314, 0.347412865, 0.369007265,
        0.385243468, 0.39582154, 0.400534955, 0.399273452, 0.392024416,
        0.378872752, 0.359999251, 0.335677471, 0.306269187, 0.272218462,
```

```
    0.234044451,   0.192333032,   0.147727406,   0.100917809,   0.0526304912,
    0.00361614596, -0.0453620379, -0.0935405097, -0.140167488,  -0.184514895,
    -0.225889915,  -0.263645981,  -0.297193036,  -0.326006889,  -0.349637532,
    -0.367716279,  -0.379961624,  -0.386183713,  -0.386287372,  -0.380273637,
    -0.368239766,  -0.350377727,  -0.326971195,  -0.298391096,  -0.265089776,
    -0.227593882,  -0.186496074,  -0.142445691,  -0.0961385186, -0.0483058328,
    0.000297127082, 0.0489030582,  0.0967446876,  0.143066865,   0.187138465,
    0.228263914,    0.265794149,   0.299136854,   0.327765796,   0.351229116,
    0.369156457,    0.381264797,   0.387362913,   0.387354391,   0.381239145,
    0.369113418,    0.351168261,   0.327686515,   0.299038355,   0.265675446,
    0.228123821,    0.186975581,   0.142879561,   0.0965310903,  0.0486610319,
    2.42514473e-05, -0.0486122865, -0.096481615, -0.142828861,  -0.18692315,
    -0.228069133,  -0.265617956,  -0.298977486,  -0.32762166,   -0.35109877,
    -0.369038598,  -0.381158247,  -0.387266606,  -0.387267363,  -0.381160527,
    -0.369042423,  -0.351104179,  -0.327628706,  -0.29898624,   -0.265628505,
    -0.228081584,  -0.186937626,  -0.142845507,  -0.0965005981, -0.0486337963
};


// compute the l2-norm difference between the current iterate 'x'
// and the exact solution of which we have the values above
double evaluate_error ()
{
  double sum_of_squares = 0;

  for (unsigned int i=0; i<N; ++i)
    sum_of_squares += std::pow((x[i]-exact_solution[i]), 2.0);

  return std::sqrt (sum_of_squares);
}



int main ()
{
  // initialize x with random values between
  // 0 and 1. set b to the values given in
  // the problem description
  for (unsigned int i=0; i<N; ++i)
    {
      x[i] = 1.*rand() / RAND_MAX;
      b[i] = std::sin(2.*3.14159265358*i/50)/100;
    }

  // do the iterations
  for (unsigned int it = 0; it<=2500; ++it)
    {
      // if this is an even iteration, then do a forward sweep:
      if (it % 2 == 0)
        {
          x[0]   = 1./2.01 * (b[0] - (-1 * x[1]));
```

```
          for (unsigned int i=1; i<N-1; ++i)
            x[i] = 1./2.01 * (b[i] - (-1 * x[i-1]) - (-1 * x[i+1]));
          x[N-1] = 1./2.01 * (b[N-1] - (-1 * x[N-2]));
        }
      else
        // but if it is an odd iteration number, do a reverse,
        // backward sweep
        {
          x[N-1] = 1./2.01 * (b[N-1] - (-1 * x[N-2]));
          for (unsigned int i=N-2; i>0; --i)
            x[i] = 1./2.01 * (b[i] - (-1 * x[i-1]) - (-1 * x[i+1]));
          x[0]   = 1./2.01 * (b[0] - (-1 * x[1]));
        }

      // at the end of each iteration output the error
      std::cout << it << ' ' << evaluate_error() << std::endl;
    }
}
```

In this particular case, the results of the symmetric Gauss-Seidel iteration happen to have the same error as the original Gauss-Seidel algorithm; this is not always the case, but happens to be so here.

Problem 7 (Steepest Descent iteration). Computing steepest descent iterations is slightly different than for the previous two algorithms, but still not overly difficult. There are three parts that we need to do: (i) compute the residual $r^{(k)} = b - Ax^{(k)}$; (ii) computing the step length $\alpha^{(k)} = \frac{r^{(k)} \cdot r^{(k)}}{r^{(k)} \cdot (Ar^{(k)})}$, and (iii) computing the update $x^{(k+1)} = x^{(k)} - \alpha^{(k)} r^{(k)}$.

Step (i) can be done as before, by utilizing that the matrix A has only 2 or 3 non-zero entries per row. Step (ii) requires the computation of $r^{(k)} \cdot r^{(k)}$, which is easy to do, and $r^{(k)} \cdot (Ar^{(k)})$ which is also easy to do if you realize that this term is equal to $\sum_i \sum_j a_{ij} r_i^{(k)} r_j^{(k)}$ and then realizing that for each i, only 3 of the a_{ij} are nonzero (and two for $i = 1, N$). In other words, we can rewrite this as $r^{(k)} \cdot (Ar^{(k)}) = \sum_i a_{ii} r_i^{(k)} r_i^{(k)} + a_{i,i-1} r_i^{(k)} r_{i-1}^{(k)} + a_{i,i+1} r_i^{(k)} r_{i+1}^{(k)}$, plus modifications for the first and last row. As before, that means that we don't actually have to store the matrix, and we also don't have to store the vector $y = Ar^{(k)}$. With this, step (iii) is then obvious.

A code that does this is here:

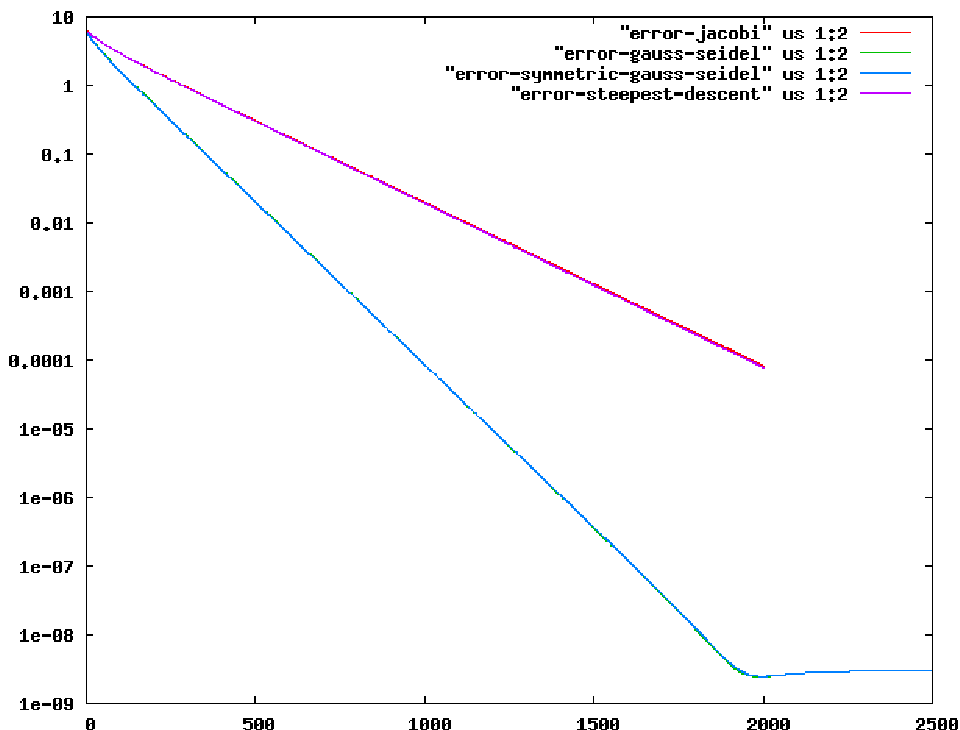
```
#include <iostream>
#include <cmath>
#include <stdlib.h>

const unsigned int N = 100;

double x[N];
double residual[N];

double b[N];

const double exact_solution[N] =
{
      0.0440078737,  0.0884558262,  0.132535005,  0.175452634,  0.216443544,
      0.254781354,   0.289789124,   0.320849314,  0.347412865,  0.369007265,
      0.385243468,   0.39582154,    0.400534955,  0.399273452,  0.392024416,
```

```
        0.378872752,    0.359999251,     0.335677471,    0.306269187,    0.272218462,
        0.234044451,    0.192333032,     0.147727406,    0.100917809,    0.0526304912,
        0.00361614596, -0.0453620379, -0.0935405097, -0.140167488,   -0.184514895,
       -0.225889915,   -0.263645981,   -0.297193036,   -0.326006889,   -0.349637532,
       -0.367716279,   -0.379961624,   -0.386183713,   -0.386287372,   -0.380273637,
       -0.368239766,   -0.350377727,   -0.326971195,   -0.298391096,   -0.265089776,
       -0.227593882,   -0.186496074,   -0.142445691,   -0.0961385186, -0.0483058328,
        0.000297127082, 0.0489030582,  0.0967446876,   0.143066865,    0.187138465,
        0.228263914,     0.265794149,    0.299136854,    0.327765796,    0.351229116,
        0.369156457,     0.381264797,    0.387362913,    0.387354391,    0.381239145,
        0.369113418,     0.351168261,    0.327686515,    0.299038355,    0.265675446,
        0.228123821,     0.186975581,    0.142879561,    0.0965310903,   0.0486610319,
        2.42514473e-05, -0.0486122865, -0.096481615, -0.142828861,   -0.18692315,
       -0.228069133,   -0.265617956,   -0.298977486,   -0.32762166,    -0.35109877,
       -0.369038598,   -0.381158247,   -0.387266606,   -0.387267363,   -0.381160527,
       -0.369042423,   -0.351104179,   -0.327628706,   -0.29898624,    -0.265628505,
       -0.228081584,   -0.186937626,   -0.142845507,   -0.0965005981, -0.0486337963
};


// compute the l2-norm difference between the current iterate 'x'
// and the exact solution of which we have the values above
double evaluate_error ()
{
  double sum_of_squares = 0;

  for (unsigned int i=0; i<N; ++i)
    sum_of_squares += std::pow((x[i]-exact_solution[i]), 2.0);

  return std::sqrt (sum_of_squares);
}



int main ()
{
  // initialize x with random values between
  // 0 and 1. set b to the values given in
  // the problem description
  for (unsigned int i=0; i<N; ++i)
    {
      x[i] = 1.*rand() / RAND_MAX;
      b[i] = std::sin(2.*3.14159265358*i/50)/100;
    }

  // do the iterations
  for (unsigned int it = 0; it<=2000; ++it)
    {
      // first compute the residual using the sparsity of the matrix
      double residual[N];
      residual[0]   = b[0] - (2.01*x[0] - 1. * x[1]);
```

```
      for (unsigned int i=1; i<N-1; ++i)
        residual[i] = b[i] - (2.01*x[i] - 1. * x[i-1] - 1. * x[i+1]);
      residual[N-1] = b[N-1] - (2.01*x[N-1] - 1. * x[N-2]);

      // for the step length, first compute the enumerator
      // residual*residual:
      double alpha_enumerator = 0;
      for (unsigned int i=0; i<N; ++i)
        alpha_enumerator += residual[i] * residual[i];

      // then compute the denominator using the trick mentioned in the
      // answers:
      double alpha_denominator = 0;
      for (unsigned int i=0; i<N; ++i)
        {
          alpha_denominator += 2.01 * residual[i] * residual[i];
          if (i>0)
            alpha_denominator += -1. * residual[i-1] * residual[i];
          if (i<N-1)
            alpha_denominator += -1. * residual[i+1] * residual[i];
        }

      double alpha = alpha_enumerator/alpha_denominator;

      // finally do the update
      for (unsigned int i=0; i<N; ++i)
        x[i] = x[i] + alpha * residual[i];

      // at the end of each iteration output the error
      std::cout << it << ' ' << alpha << ' ' << evaluate_error() << std::endl;
    }
}
```

Results are again shown for Problem 8 below.

Problem 8 (Comparing methods). The following figure shows how the error $\|x - x^{(k)}\|_{l_2}$ decreases with the number of iterations:



Here, the symmetric and non-symmetric Gauss-Seidel variants have essentially the same values, and so do the Jacobi and Steepest Descent methods.

It is clear that the (symmetric) Gauss-Seidel iteration is much faster to converge than the Jacobi iteration. For example, after 1000 iterations, the former method has produced a solution with an error that is $8.5 \cdot 10^{-5}$, whereas the approximate solution of the Jacobi method after this many iterations still has an error of 0.020 – some 230 times larger. Maybe more importantly, if you strive for an error of 10^{-6}, you will need 2808 iterations with the Jacobi method, but only 1407 with the Gauss-Seidel method. It is important to also remember, if you look at the two algorithms, that both methods actually do the same amount of work; in fact, the Jacobi method does slightly more because it needs to copy the solution vector at the beginning of the iteration.

As a final thought, take a look at the Gauss-Seidel iteration: it seems to level off after some 1900 iterations, and the error no longer decreases. Why is that? It is because I have only written the exact solution into the source files to 8 or 9 digits of accuracy. For example, the first value x_1 in the source file is 0.0440078737, but that is almost certainly not the *exact* value of the first solution component – the exact value would have several more digits (maybe infinitely many), and has only been rounded to the value shown. The difference will therefore be within the interval $[-0.00000000005, 0.00000000005]$, and statistically speaking the difference from the shown value will be around 0.000000000025 in magnitude on average. This is true for every component. In other words, even if we have a method that computes the *exact* solution, the `evaluate_error()` function will compute an "error" because what we store in the program is not, in fact, the exact solution – it is just an approximation. Since we output only 8 digits, it is easy to see that the l_2 norm difference between anything that is really close to the exact solution and the `evaluate_solution` variable in the code will be around 10^{-8} – as is indeed the case in the figure.