MATH 561: Numerical Analysis I

Instructor: Prof. Wolfgang Bangerth
bangerth@colostate.edu

Answers for homework assignment 4

Problem 1 (Conjugate Gradient iteration). Compared to programs that use the Jacobi or Gauss-Seidel method, writing a program from scratch that uses the Conjugate Gradient iteration is a bit more involved because it requires numerous temporary vectors, vector copies, dot products, etc. If one were to try, it would be appropriate to encapsulate these kinds of operations into vector and matrix classes, but then a program would immediately be significantly larger than what is useful to show here. Rather, I would advise to just google for implementations of CG in Matlab or a number of other programming environments that have such vector and matrix classes.

That said, without much attempt at explanation, here is an implementation in the `deal.II` system that I happen to know well and for which there is already a CG class. The majority of the code here is devoted to evaluating the error in each step, by (ab)using the `print_vectors` function:

```
#include <deal.II/lac/full_matrix.h>
#include <deal.II/lac/vector.h>
#include <deal.II/lac/solver_cg.h>
#include <deal.II/lac/precondition.h>
#include <iostream>

using namespace dealii;

const unsigned int N=100;
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 (const Vector<double> &x)
{
  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);
}


class MySolverCG : public SolverCG<>
{
public:
  MySolverCG (SolverControl &control)
  :
  SolverCG<>(control)
    {}

  virtual void print_vectors(const unsigned int step,
                             const Vector<double>   &x,
                             const Vector<double>   &,
                             const Vector<double>   &) const
    {
      std::cout << step << ' ' << evaluate_error(x) << std::endl;
    }
};


int main ()
{
  FullMatrix<double> A(N,N);
  for (unsigned int i=0; i<N; ++i)
    for (unsigned int j=0; j<N; ++j)
      if (i==j)
        A(i,j) = 2.01;
      else if (i+1==j)
        A(i,j) = -1;
      else if (j+1==i)
        A(i,j) = -1;

  Vector<double> b(N);
  for (unsigned int i=0; i<N; ++i)
    b(i) = 1./100 * std::sin(2.*M_PI*i/50.);

  Vector<double> x(N);
  for (unsigned int i=0; i<N; ++i)
    x[i] = 1.*rand() / RAND_MAX;
```

```
  SolverControl control (100);
  MySolverCG     cg(control);

  cg.solve (A, x, b, PreconditionIdentity());

}
```

It produces the following output, similar to the programs in the previous homework:

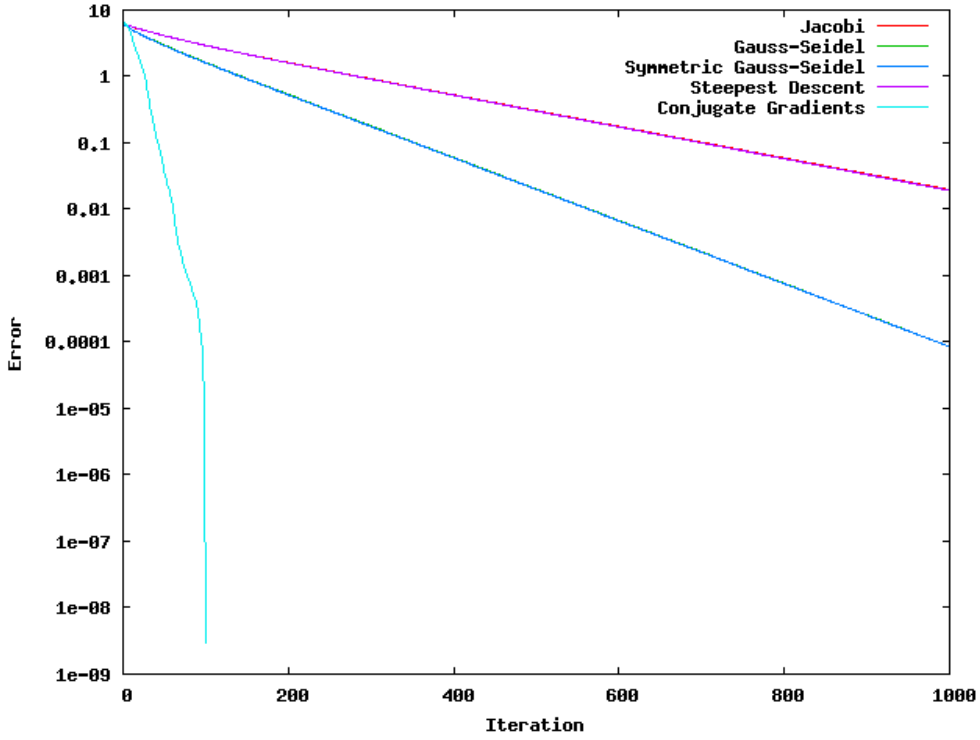
```
1 6.38679
2 6.22214
3 6.06544
4 5.915
5 5.74731
6 5.50458
7 5.21522
8 4.96438
9 4.64302
10 4.12201
11 3.67867
12 3.32237
13 3.07851
14 2.86559
15 2.57217
16 2.40848
17 2.24732
18 2.06427
19 1.90954
20 1.75201
21 1.56885
22 1.43034
23 1.31334
24 1.22302
25 1.11926
26 1.02216
27 0.916024
28 0.812003
29 0.680413
30 0.565884
31 0.495084
32 0.431357
33 0.360707
34 0.285168
35 0.244079
36 0.212231
37 0.177302
38 0.154958
39 0.135408
40 0.119771
41 0.108518
42 0.100897
43 0.0910954
```

44	0.0814086
45	0.0718391
46	0.0643351
47	0.0576503
48	0.0505103
49	0.0435575
50	0.0370632
51	0.0316098
52	0.0272282
53	0.0237624
54	0.0216385
55	0.0202608
56	0.0184558
57	0.0162288
58	0.0147887
59	0.0135204
60	0.0113032
61	0.00944535
62	0.00760725
63	0.00597306
64	0.00518262
65	0.0044312
66	0.00372427
67	0.00303265
68	0.00259011
69	0.00225884
70	0.00198251
71	0.00179096
72	0.00154374
73	0.00142663
74	0.00128418
75	0.00118779
76	0.00109729
77	0.00104149
78	0.000955988
79	0.000890067
80	0.000830362
81	0.00079017
82	0.000719525
83	0.000647478
84	0.000595777
85	0.000544643
86	0.000492648
87	0.000441175
88	0.000401053
89	0.000363637
90	0.000310447
91	0.000254872
92	0.000185902
93	0.000150941
94	0.000123376

```
95  0.000101565
96  7.66458e-05
97  3.15404e-05
98  2.08336e-05
99  3.02281e-06
100  2.97862e-09
```

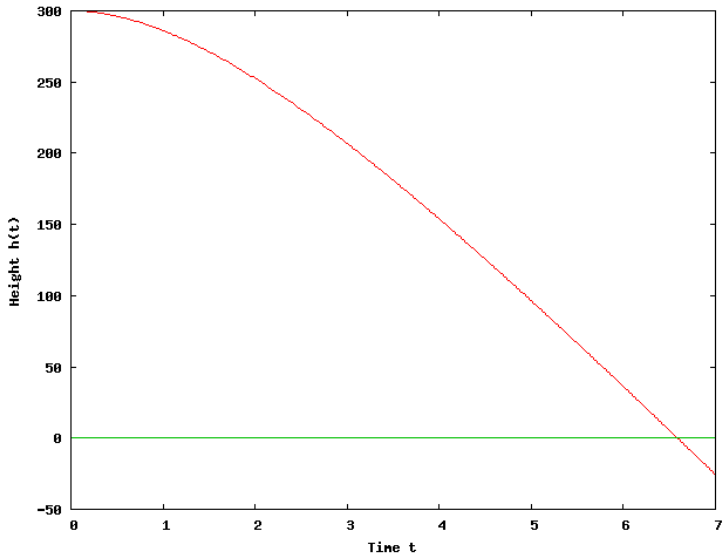
After 100 iterations, the error is in essence zero, and that's how it should be because we know that the CG method yields the exact solution of systems of size N after N iterations. (That is because in iteration k, CG searches the solution in a subspace of dimension k. In iterarion N, this subspace is the entire space, so the iterate $x^{(k)}$ must equal the exact solution of the linear system.)

With this, we can update the picture that shows the error as a function of the iteration number for the various methods:



As before, the symmetric and non-symmetric Gauss-Seidel variants have essentially the same values, and so do the Jacobi and Steepest Descent methods. On the other hand, Conjugate Gradients converges *dramatically faster*, being the vastly superior one of all of the methods in question.

Problem 2 (Bisection method by hand). Before showing the steps, it is worth illustrating the behavior of the function for which we seek the root. This is done in the following graph:



We can see that the ball hits the ground (i.e., height zero) somewhere between 6 and 7 seconds. So let us start with this interval. We then have to do the following steps:

- Iteration 1: We have $a_1 = 6, b_1 = 7$ and evaluating $h(t)$ for these two values, we see that $h(a_1) > 0$ and $h(b_1) < 0$. So the root must be within this interval. We then try $c_1 = (a_1 + b_1)/2 = 6.5$, i.e., the midpoint and find that $h(c_1) > 0$. So the root must lie between c_1 and b_1 and we set $a_2 = c_1, b_2 = b_1$.
- Iteration 2: We repeat and compute $c_2 = (a_2 + b_2)/2 = 6.75$, and find that $h(c_2) < 0$. So the root must lie between a_2 and c_2 and we set $a_3 = a_2, b_3 = c_2$.
- Iteration 3: We repeat and compute $c_3 = (a_3 + b_3)/2 = 6.625$, and find that $h(c_4) < 0$. So the root must lie between a_3 and c_3 and we set $a_4 = a_3, b_4 = c_3$.

We are now done because the interval within which the root must lie is $[6.5, 6.625]$. Our best guess for where the root is is the midpoint of the interval, $x_4 = (a_4 + b_4)/2$, and the distance of the root from this point (i.e., the error $e_4 = |x_4 - x^*|$) must clearly be less than the distance from this point to the end points of the interval. This distance is $(b_4 - a_4)/2 = 0.06125$, i.e., less than the desired error level.

Problem 3 (Bisection method with a computer). A code that does this is here:

```
#include <iostream>
#include <cmath>

double h_of_t (const double t)
{
  const double h0 = 300;
  const double m  = 0.5;
  const double g  = 32.17;
  const double k  = 0.25;

  return h0 - m*g/k*t + m*m*g/k/k*(1-std::exp(-k*t/m));
}


int main ()
```

```
{
  std::cout.precision(12);

  double left  = 6;
  double right = 7;

  double h_left = h_of_t(left);
  double h_right = h_of_t(right);

  for (unsigned int iteration=1; iteration<=30; ++iteration)
    {
      // compute new midpoint and evaluate h(t) there
      const double midpoint = (left + right) / 2;
      const double h_midpoint = h_of_t(midpoint);

      // check whether the sign change is in the left half of the
      // interval. if so, make the left half the current interval
      if (h_left * h_midpoint < 0)
        {
          right   = midpoint;
          h_right = h_midpoint;
        }
      else
        {
          left   = midpoint;
          h_left = h_midpoint;
        }

      std::cout << "Iteration=" << iteration
                << ", best guess=" << (left+right)/2
                << ", estimated error=" <<  (right-left)/2
                << std::endl;
    }
}
```

The code of course needs to start from somewhere, and we have chosen the interval $[6, 7]$ as the first guess, based on the picture shown in the previous function.

If you run this code, you'll get the following output:

```
Iteration=1, best guess=6.75, estimated error=0.25
Iteration=2, best guess=6.625, estimated error=0.125
Iteration=3, best guess=6.5625, estimated error=0.0625
Iteration=4, best guess=6.59375, estimated error=0.03125
Iteration=5, best guess=6.578125, estimated error=0.015625
Iteration=6, best guess=6.5859375, estimated error=0.0078125
Iteration=7, best guess=6.58984375, estimated error=0.00390625
Iteration=8, best guess=6.587890625, estimated error=0.001953125
Iteration=9, best guess=6.5888671875, estimated error=0.0009765625
Iteration=10, best guess=6.58837890625, estimated error=0.00048828125
Iteration=11, best guess=6.58862304688, estimated error=0.000244140625
Iteration=12, best guess=6.58850097656, estimated error=0.0001220703125
Iteration=13, best guess=6.58856201172, estimated error=6.103515625e-05
```

```
Iteration=14, best guess=6.58853149414, estimated error=3.0517578125e-05
Iteration=15, best guess=6.58854675293, estimated error=1.52587890625e-05
Iteration=16, best guess=6.58853912354, estimated error=7.62939453125e-06
Iteration=17, best guess=6.58853530884, estimated error=3.81469726562e-06
Iteration=18, best guess=6.58853721619, estimated error=1.90734863281e-06
Iteration=19, best guess=6.58853816986, estimated error=9.53674316406e-07
Iteration=20, best guess=6.5885386467, estimated error=4.76837158203e-07
Iteration=21, best guess=6.58853888512, estimated error=2.38418579102e-07
Iteration=22, best guess=6.58853900433, estimated error=1.19209289551e-07
Iteration=23, best guess=6.58853894472, estimated error=5.96046447754e-08
Iteration=24, best guess=6.58853897452, estimated error=2.98023223877e-08
Iteration=25, best guess=6.58853898942, estimated error=1.49011611938e-08
Iteration=26, best guess=6.58853898197, estimated error=7.45058059692e-09
Iteration=27, best guess=6.5885389857, estimated error=3.72529029846e-09
Iteration=28, best guess=6.58853898384, estimated error=1.86264514923e-09
Iteration=29, best guess=6.58853898477, estimated error=9.31322574615e-10
Iteration=30, best guess=6.5885389843, estimated error=4.65661287308e-10
```

From this, we can conclude that the correct answer is $t^* = 6.58853898$, to the given digits.

To determine the error, it is important to realize that for complicated problems, we of course don't know the exact solution. Rather, all we know is that if the interval in iteration k is $[a_k, b_k]$, the best guess we have for the location is the solution is the midpoint of the interval, $x_k = (a_k + b_k)/2$. Furthermore, because the exact solution must lie somewhere in the interval, its distance from the best guess x_k can not be more than the distance from x_k to the end points of the interval, i.e., we know that $|x_k - x^*| \leq (b_k - a_k)/2$. This is what the last column shows.

In other words, to get to an error of less than 10^{-2} seconds, and starting with the interval stated above, we need 6 iterations. To 10^{-4}, it requires 13. To 10^{-6}, it takes 19, and to 10^{-8} 26. I didn't run the program long enough for 10^{-12}, but you can guess how many iterations that would take.

Problem 4 (Newton's method). Newton's method finds the next iterate x_{k+1} by taking the tangent to the function $f(x)$ at $x = x_k$ and seeing where that intersects the x-axis. In general, this may be close to a root x^* of the function, but will not exactly be where f is zero. However, it actually *is* a root for f if the function is linear, because then the function and its tangent will be the same. In that case, following the tangent – from wherever we start – will yield the exact solution in one step.

Problem 5 (Newton's method). A code that does this is here:

```
#include <iostream>
#include <cmath>

double f (const double x)
{
  return std::pow(x,25)-1;
}

double f_prime (const double x)
{
  return 25*std::pow(x,24);
}


int main ()
```

```
{
  std::cout.precision(12);

  double x_k = 20;

  for (unsigned int iteration=0; iteration<=30; ++iteration)
    {
      std::cout << "Iteration=" << iteration
                << ", current_iterate=" << x_k
                << std::endl;

      x_k = x_k - f(x_k)/f_prime(x_k);
    }
}
```

The code of course needs to start from somewhere, and we have chosen the interval $[6, 7]$ as the first guess, based on the picture shown in the previous function.

If you run this code, you'll get the following output:

```
Iteration=0, current_iterate=20
Iteration=1, current_iterate=19.2
Iteration=2, current_iterate=18.432
Iteration=3, current_iterate=17.69472
Iteration=4, current_iterate=16.9869312
Iteration=5, current_iterate=16.307453952
Iteration=6, current_iterate=15.6551557939
Iteration=7, current_iterate=15.0289495622
Iteration=8, current_iterate=14.4277915797
Iteration=9, current_iterate=13.8506799165
Iteration=10, current_iterate=13.2966527198
Iteration=11, current_iterate=12.764786611
Iteration=12, current_iterate=12.2541951466
Iteration=13, current_iterate=11.7640273407
Iteration=14, current_iterate=11.2934662471
Iteration=15, current_iterate=10.8417275972
Iteration=16, current_iterate=10.4080584933
Iteration=17, current_iterate=9.9917361536
Iteration=18, current_iterate=9.59206670745
Iteration=19, current_iterate=9.20838403915
Iteration=20, current_iterate=8.84004867759
Iteration=21, current_iterate=8.48644673048
Iteration=22, current_iterate=8.14698886127
Iteration=23, current_iterate=7.82110930681
Iteration=24, current_iterate=7.50826493454
Iteration=25, current_iterate=7.20793433716
...
```

This is not exactly rapid progress towards the solution $x^* = 1$. However, if we wait for a while, it looks like the convergence eventually becomes quite rapid after all:

```
Iteration=65, current_iterate=1.40819274254
Iteration=66, current_iterate=1.3518758514
Iteration=67, current_iterate=1.29782962956
```

```
Iteration=68, current_iterate=1.24599315147
Iteration=69, current_iterate=1.19635745083
Iteration=70, current_iterate=1.14904440164
Iteration=71, current_iterate=1.10450815611
Iteration=72, current_iterate=1.06400916824
Iteration=73, current_iterate=1.03047222016
Iteration=74, current_iterate=1.00871540312
Iteration=75, current_iterate=1.00084651096
Iteration=76, current_iterate=1.00000853624
Iteration=77, current_iterate=1.00000000087
Iteration=78, current_iterate=1
```

From this output, we can see that it takes 77 iterations to reach an accuracy of 10^{-8}, but at that point we gain several correct digits in each iteration already.

The explanation for this behavior is that we know that the error $e_k = x_k - x^*$ satisfies the formula $e_{k+1} = Ce_k^2$, which implies rapid progress if e_k is already small and C is small as well. But for Newton's method, we know that $C \approx \frac{1}{2}\frac{f''(x^*)}{f'(x^*)}$. In general, if a function is linear (see also problem 4), then $C = 0$ and we converge in one step; if f is mostly linear, i.e., its second derivative is small compared to its first, then C is small and we converge rapidly from the beginning. But here, $f''(x^*) = 25 \cdot 24 = 600$ and $f'(x^*) = 25$, so $C = 12$ which is quite large and the formula above only predicts that e_{k+1} is smaller than e_k if $e_k < \frac{1}{12}$, i.e., it is already quite small. (If e_k is larger than that, the formula does not guarantee convergence at all, but the fact that the function is convex does so.) It is easy to verify, however, that once we get close to the solution, we still have quadratic convergence.

Finally, if we applied the program to the function $f(x) = x^3 - 1$ instead, we would get the following output, demonstrating very rapid convergence:

```
Iteration=0, current_iterate=20
Iteration=1, current_iterate=13.3341666667
Iteration=2, current_iterate=8.89131921009
Iteration=3, current_iterate=5.9317625841
Iteration=4, current_iterate=3.96398190609
Iteration=5, current_iterate=2.663868255
Iteration=6, current_iterate=1.82288570674
Iteration=7, current_iterate=1.31557074928
Iteration=8, current_iterate=1.06964452533
Iteration=9, current_iterate=1.00443613823
Iteration=10, current_iterate=1.00001956356
Iteration=11, current_iterate=1.00000000038
Iteration=12, current_iterate=1
```

Problem 6 (Convergence order for sequences). Postponed to the next homework.