Problem 1 (Steepest descent iteration). When plotted, the contour lines and surface plots for the functions \(q(y)\) show an elliptic trough that becomes more and more elongated as we increase \(a_{11}\). For example, for \(a_{11} = 1, 10, 100\), we get the following three plots:

![Contour plots](image)

To run the steepest descent iteration on this, let us use the following program that implements the algorithm:

```cpp
#include <iostream>
#include <iomanip>
#include <cmath>

const double a11 = 100;

int main ()
{
    std::cout << std::setprecision(16);
    double x[2] = {2, a11};
    double g[2];
    double alpha;
    unsigned int iteration = 0;
    while (true)
    {
        std::cout << iteration << ' ' << x[0] << ' ' << x[1] << std::endl;
        // then do the iteration
    }
}
```

1
// end the iteration once we’ve achieved the desired accuracy
if (std::sqrt((x[0]-1.)*(x[0]-1.) + x[1]*x[1]) < 1e-4)
    break;

// compute the gradient as Ax-b

    g[0] = a11*x[0] - a11;
    g[1] = x[1];

// compute the step length

    alpha = (g[0]*g[0] + g[1]*g[1]) / (g[0]*a11*0 + g[1]*1*1);

// form the next iterate

    x[0] = x[0] - alpha*g[0];
    x[1] = x[1] - alpha*g[1];

++iteration;

}
}

Running this program generate a list of the iterates. Let us plot them as a red line on top of the surface plot for the case that $a_{11} = 10$:

![Surface plot with iterates](image)

It is obvious that the solution does indeed converge to the minimum of the surface, but it wiggles back and forth. We can visualize this in a different way if we just plot the path of the iteration together with the contours of the function $q(y)$ at levels corresponding to where iterates lie:
In each iterate, the direction we choose is orthogonal to the isolines shown (if this doesn’t seem to be exactly the case, then it is due to an unequal ratio of side lengths in the plot). We take a step in this direction that shoots over the middle line of the valley and has to go back and forth between the two sides of the valley. The more elongated the valley becomes, the worse this effect is getting. For example, here are the iterates for $a_{11} = 100$: 
In effect, the larger \( a_{11} \) is (or, put differently: the larger the condition number of the matrix), the more iterations we will need. Running the program for a number of different values of \( a_{11} \) yields the following results:

<table>
<thead>
<tr>
<th>( a_{11} )</th>
<th>condition number ( \kappa(A) )</th>
<th>number of iterations ( N )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>58</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>691</td>
</tr>
<tr>
<td>1000</td>
<td>1000</td>
<td>8060</td>
</tr>
<tr>
<td>10000</td>
<td>10000</td>
<td>92104</td>
</tr>
<tr>
<td>100000</td>
<td>100000</td>
<td>1036164</td>
</tr>
</tbody>
</table>

This behavior is almost linear (a perfectly linear relationship would increase the number of iterations ten fold each time we increase \( a_{11} \) by a factor of 10). A better approximation would be \( N = \mathcal{O}(N \log(\log N)) \), but for practical purposes, factors of \( \log(\log N) \) are negligible (since for example \( \log(\log 10^6) \approx 2.5 \), i.e. a small factor in the grand scheme of things).

**Problem 2 (CG iteration).** A small program that implements the CG method for this particular matrix and right hand side is quickly written:

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

const unsigned int N = 100;

double x[N], r[N], b[N], v[N], z[N];

double c, d, t;
```
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;
    }

    // compute the initial residual and copy it into v
    r[0] = - (2.01*x[0] - x[1] - b[0]);
    for (unsigned int i=1; i<N-1; ++i)
        r[i] = - (2.01*x[i] - x[i-1] - x[i+1] - b[i]);
    r[N-1] = - (2.01*x[N-1] - x[N-2] - b[N-1]);

    for (unsigned int i=0; i<N; ++i)
        v[i] = r[i];

    c = 0;
    for (unsigned int i=0; i<N; ++i)
        c += r[i] * r[i];

    // do the iterations
    for (unsigned int it = 0; it<=200; ++it)
    {
        // in each iteration, produce some output
        for (unsigned int i=0; i<N; ++i)
            std::cout << it << ' ' << i << ' ' << x[i] << std::endl;

        // compute 'z' as A*v
        z[0] = (2.01*v[0] - v[1]);
        for (unsigned int i=1; i<N-1; ++i)
            z[i] = (2.01*v[i] - v[i-1] - v[i+1]);
        z[N-1] = (2.01*v[N-1] - v[N-2]);

        // compute <v,z>
        double v_times_z = 0;
        for (unsigned int i=0; i<N; ++i)
            v_times_z += v[i] * z[i];

        // update v
        for (unsigned int i=0; i<N; ++i)
            v[i] -= c * z[i];

        // compute c
        c = 0;
        for (unsigned int i=0; i<N; ++i)
            c += r[i] * r[i];

        // compute new residual
        r[0] = - (2.01*v[0] - v[1] - b[0]);
        for (unsigned int i=1; i<N-1; ++i)
            r[i] = - (2.01*v[i] - v[i-1] - v[i+1] - b[i]);
        r[N-1] = - (2.01*v[N-1] - v[N-2] - b[N-1]);
    }
}
v_times_z += v[i] * z[i];

    // then t and update x

t = c / v_times_z;

    for (unsigned int i=0; i<N; ++i)
        x[i] += t * v[i];

        // update the residual

for (unsigned int i=0; i<N; ++i)
    r[i] -= t * z[i];

        // compute 'd'

    d = 0;
    for (unsigned int i=0; i<N; ++i)
        d += r[i] * r[i];

    // in practice, we would check whether
    // the norm of the residual,
    // i.e. sqrt(d) is smaller than some
    // tolerance, here, we simply keep
    // going by updating 'v' since we are
    // interested in what happens for large
    // numbers of iterations

for (unsigned int i=0; i<N; ++i)
    v[i] = r[i] + d/c * v[i];

    c = d;
}
}

It should be noted that you should, in general, not do this: Matlab has much better ways to handle vectors so that you don’t have to write all these loops. The code would therefore have been a lot shorter in Matlab than shown here. On the other hand, writing code such as this is a good exercise and is also something one might do when developing larger software packages.

If I run the code, and plot the results, I can graph the iterates as follows:
Likewise, I can graph the $l_2$-norm of the difference between $x_k$ and $x_{100}$ (both graphs show the same data, but the left one is linear and the right one logarithmic):

From these results, it is obvious that the CG iteration converges relatively quickly compared to the Steepest Descent algorithm or any of the Defect Correction methods we’ve seen in class. In particular, we essentially have convergence in only 40-50 iterations. As a remark on the side: one can see that the CG iteration shares one trait with the defect correction schemes for which we have created similar graphs before: it smooths the solution vector (i.e., even if you start with a wildly oscillating initial vector made up from random numbers, one gets a relatively smooth vector after only a few iterations, and the remaining iterations are used to reduce the long-range error components). The smoothing property is something that is crucial for multigrid methods that are among the most efficient solvers for linear systems arising from partial differential equations (the topic of MATH 610).

After 100 iterations, there is only marginal progress (not even visible in the graph). This shouldn’t surprise us: in the CG iteration, we construct a basis
In which we develop our iterate $x_k$. However, for a 100-dimensional vector space, there can be only 100 basis vectors, and the CG method should find the exact solution after at most 100 iterations. After that, the residual $r_k$ should be zero, and the iteration should in theory start to fail due to divisions by zero. In practice, however, it doesn’t: over the course of iterations, we accumulate some round-off that makes sure that the residual is non-zero, even if extremely small. We can therefore continue computing update directions, though it should be noted that such computations are pretty pointless and should be avoided: whatever accuracy you have after $N$ iterations can essentially not be improved upon.