Numerical Analysis for Computer Scientists
FMN011, Lund University 2012
Project #1
Solving huge systems of equations

Erik Westrup, <ada09ewe@student.lu.se>

1 Introduction and Problem Background

This project is about solving systems of very huge matrices. More specifically the matrices are not only huge but also almost banded and sparse which have important consequences and applications. A sparse quadratic $n \times n$ matrix is a matrix where the number of non-zero elements is, using big O notation, only $O(n)$ compared to a "full" matrix which has $O(n^2)$ non-zero elements. This is actually a common situation when we have measured information from the real world where not all the variables are depended on each other. To further explore the sparse system two types of methods for solving will be used: direct and iterative. Direct methods finds exact solution in a finite number of steps and iterative approaches the solution and finds and approximation only. The direct methods are useful when an exact solution is required (and exists) while iterative methods suits real time problem where data must be processed and close but not exact solution is sufficient. It is not always certain that iterative methods actually will converge towards the correct solution. However we will here only work with one specific sparse matrix for which we know an iterative method will converge.

In theory we can solve any large system (if it converge in the iterative case of course) but that requires equally much time. Unlimited computational power is (at present to be optimistic) not available and we have to deal with constraints in both time and memory. That is why we need to look at this problem from a computational point of view. Of course it's possible to theoretically consider the hardware specifications and analyze the execution of machine code to find out how large system we can solve. However it is much pragmatic and simpler to just test it on the hardware directly. So that is what will be done in this project.

The sparse $n \times n$ matrix A that should be used in this project is defined by:

$$\begin{aligned} A(i, i-1) &= -1, \quad i \in [2, n] \\ A(i, i) &= 3, \quad i \in [1, n] \\ A(i, i+1) &= -1, \quad i \in [1, n-1] \\ A(i, n+1-i) &= \tfrac{1}{2}, \quad i \in [1, n-1] \backslash \frac{n}{2} \wedge \frac{n}{2} + 1 \end{aligned} \tag{1}$$

With $n = 8$ it looks like this:

$$A = \begin{pmatrix} 3 & -1 & 0 & 0 & 0 & 0 & 0 & 0.5 \\ -1 & 3 & -1 & 0 & 0 & 0 & 0.5 & 0 \\ 0 & -1 & 3 & -1 & 0 & 0.5 & 0 & 0 \\ 0 & 0 & -1 & 3 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 3 & -1 & 0 & 0 \\ 0 & 0 & 0.5 & 0 & -1 & 3 & -1 & 0 \\ 0 & 0.5 & 0 & 0 & 0 & -1 & 3 & -1 \\ 0.5 & 0 & 0 & 0 & 0 & 0 & -1 & 3 \end{pmatrix}$$

2 Numerical Considerations

Exploiting the vacuity of a matrix in when it's used in numerical calculations gives great benefits since we only need to store the non-zero elements which saves

us a factor of $O(n)$ in space which is important since a computer is limited in memory. Also matrix operations can be specialized that takes advantage of this structure to do less operations [1].

The most common direct method is the *Gaussian elimination*[1] and therefor it will be tested here [2]. There are many iterative methods like Jacobi or Gauss-Seidel but we will here use a method called *Successive Over-Relacation*[2] that is a modification of Gauss-Seidel that iterates more aggressively to achieve faster convergence. When the convergence is guaranteed we can iterative a given number of times, let it run for a given time or if we know the real solution until the maximum error is below some small ϵ. [3].

To solve this any programming language could have been used but using tool and language developed for numerical computation makes the task much simpler. Thus I've used *MATLAB*® to implement and test the algorithms. *MATLAB*® have built in support for dealing with sparse matrices and most matrix operations are overloaded to exploit this as well. I've used many other features *MATLAB*® offers that does not affect the complexity but makes the code easier or more reusable including functions, packages, exceptions, formatted strings, default arguments and time measurements.

2.1 Time out

I wanted to run the script in a predetermined time since my computer is a shared resource that I need for a lot of other things. Also it is good to be able to run different tests the same amount of time one can make comparisons. It's not as simple as just adding time checks before or after a call to a function that is tested since the next invocation could the one that hits the wall of computational infeasibility. You can't know that before calling the method because then there would really not be anything to test. Since I did not want to make it too complicated by using threads (I guess it's possible but I have not investigated it) continuous time checks has to be done inside the functions. Therefore I made a function *time_out()* to solve this problem. In the simplest case the function is called with the number of seconds the caller is prepared to wait as an argument, typically *time_out(maxtime)*. If no time out is desired the paramter can be set to ∞ explicity at the caller or as a default paramter in a function. The function then records this data in static variables. Then calls to the function without any arguments are places inside loops in the implementation algorithm. The argument-less call to *time_out()* will check if the time out is reached. If it is an exception is thrown which has to be caught some where in the call chain.

If a function wants to support time outs it can do so in two ways. In the first situation the function will take the maximum time to wait as a parameter and self initialize the time out function. In the second case it will only make the argument-less calls and thus the caller must handle the time out initialization. This latter case is used when a script makes many calls to this function but wants to time out not on individual calls but since the start of the script. The first situation was used by me when I implemented the functions and the second is used in the final version of the scripts.

3 Results & Analysis

In table 3 a concise presentation of the results can be found. The following subsections will comment the results if necessary.

[1] GE from now on.
[2] SOR from now on.

Task #	Result
1	$4 \times n - 6$ elements
2	n/a
3	n/a
4	$k = 10$
5	$k = 22$
6	$\omega \in [1.105500, 1.212400]$
7	$k = 13$
8	$k = 22$
9	n/a
10	Orig: 0.008563s, Pert: 0.008500s, $\frac{Pert}{Orig} = 99.25\%$
11	Orig: 0.003601s, Pert: 0.001449s, $\frac{Pert}{Orig} = 40.23\%$
12	GE, $k = 7 \Rightarrow$ Orig: 13.95055s, Pert: 13.94964s, $\frac{Pert}{Orig} = 99.99\%$
12	SOR, $k = 21 \Rightarrow$ Orig: 14.77553s, Pert: 7.76254s, $\frac{Pert}{Orig} = 52.54\%$
12	$k_{\text{best}} = 7$
13	GE, $k = 7$, Total average: 13.8909941537s
13	SOR, $k = 7$, Total average: 0.0032668375s

Table 1: A concise presentation of the results.

3.1 Task #1

To calculate the number of elements $a \in A, a \neq 0$ we can simply sum up the elements from the definition of A (1).

$$\sum_{i=2}^{n} + \sum_{i=1}^{n} + \sum_{i=1}^{n-1} + \sum_{i=1, i \neq \frac{n}{2}, \frac{n}{2}+1}^{n} \tag{2}$$

$$= (n-1) + n + (n-1) + (n-2) = 4 \times n - 4 \tag{3}$$

This indeed is a sparse matrix since $O(4 \times n - 4) = n$ which is the definition of a sparse matrix [1].

3.2 Task #2

See program listings in appendix A.

3.3 Task #3

The matrix A is strictly diagonally dominant since the main diagonal entries at each row is greater in magnitude than the summation of the absolute values the other entries at that row. Looking at A with $n = 8$ we see that three (satisfied) situations exists:

$$\begin{aligned} 3 &> |-1| + |\tfrac{1}{2}| &&= 1.5 \\ 3 &> |-1| + |-1| + |\tfrac{1}{2}| &&= 2.5 \\ 3 &> |-1| + |-1| &&= 2 \end{aligned}$$

For implementation details; see program listings in appendix A.

3.4 Task #4

Running the script on my desktop computer[3] for 5h blank the naive GE manages to solve systems of 2^{10} unknowns. I did not run the script longer because I needed my computer for other school work but the exact time here is not of importance. The runtime is long enough tho hit the wall of computational infeasibility for the naive GE i.e. $k \approx 11$. This runtime of $5h$ is used as a reference time in the following tasks since it seems to be long enough for finding computability borders when stepping with $n = 2^k$.

3.5 Task #5

With $MATLAB^{®}$s built in backslash operator and a time limit of 5 hours systems with 2^{22} unknowns was solved. It is obvious that the backslash operator is far more efficient than my own GE implementation since it solved systems with $k = 13$ in the matter of seconds using the same hardware as task 4. This is because it because the operator is overloaed with to make exploit structures like sparse and banded matrices. What is interesting here though is that the computations here was not limited in time but in space because at $k = 23$ my computer ran out of free memory and $MATLAB^{®}$ terminated with a "Out of Memory Exception". Is this reasonable?

My computer have $2\text{GiB} = 2 * 1024^3\text{B}$ of memory. The largest data structure in use is the matrix A which is a sparse matrix with $4 \times n - 6$ elements each represented by a 3-tuple $(x, y, data)$. The data field is 64 bits in $MATLAB^{®}$ and lets assume that the index variables x and y is just large enough to address everything in memory which is 4 bytes on my 32 bits computer. Summing up (in bytes) the used memory and dividing with the available memory we get:

$$\text{Usage}(k) = \frac{\text{sizeof}(A(k))}{\text{Available memory}} = \frac{(4 \times 2^k - 6) \times (2 \times 32 + 64)}{2 \times 1024^3}$$

$$\text{Usage}(22) \approx 1.0$$
$$\text{Usage}(23) \approx 2.0$$

Of course the Operating System and the other (idle) programs running needs a share of the memory and I also have a swap partition of 4GiB that I don't really know how it is utilized here. But still we see that even when using the sparse matrix structure there is a limit and it is reachable.

3.6 Task #6

The relative error at each step in SOR is equal to the absolute error since the correct solution x has ones in all places:

$$\text{Relative forward error} = \frac{||x - x_c||_\infty}{||x||_\infty} = \frac{||x - x_c||_\infty}{1} = \text{Forward error}$$

We should use the infinity norm to measure the maximum error. This because we are interested in knowing the maximum error for any individual element in the approximate solution, which is the forward error, and from above we see that it is equal to just the infinity norm of the difference between the real and computed x [4].

With a limit of 1024 iterations and w-resolution of 0.0001 chosen in the interval $[1, 2]$, the best relaxation parameter is $\omega \in [1.105500, 1.212400]$, which gives a solution

[3] A standard iMac 7,1 from 2007 with 2GiB of RAM and a processor specification as follows: Intel(R) Core(TM)2 Duo CPU T7700 @ 2.40GHz

after 9 iterations. Since any ω in this rage I will, for simplicity, use $\omega = 1.2$ in the following tasks. By plotting the number of iterations as a function of the relaxation parameter (figure (3.6)) we see that SOR is very bad for this system when it approaches 2. Also note that SOR is better than Gauss-Seidel for this systems since the latter corresponds to $\omega = 1.0$ which is not in the found optimal interval.

How come this iterative method is so much better than the exact GE used before? It turns out that the sparse property of the system is not preserved during the calculations in GE. Running the script for task #4 again with the debug option for printing the number of non-zero elements before and after elimination we get the result in table 3.6. The number of non-zero elements increases rapidly with k and thus the problem becomes approaches a $O(n^2)$ with increasing values of k. This problem is called *fill-in* [1]

k	Before	After
1	28	28
4	60	96
5	124	342
6	252	1261
7	508	4820
8	1020	18945

Table 2: Number of non-zero elements in A before and after elimination in GE.

The iterative SOR method on the other hand does not need to change the structure of A (or the L and U parts of it) and the sparseness is exploited all the way.

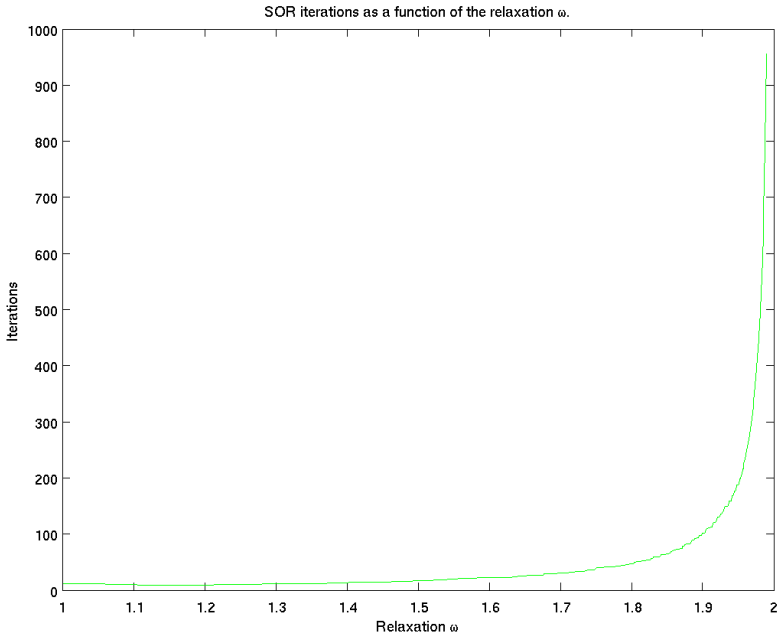

Figure 1:

3.7 Task #7

Running for 5 hours SOR was able to solve systems with 2^{13} unknowns. This is better then the naive GE probalby since it finds solution that are only correct to the 4th decimal. However it is much slower than GE using *MATLAB*®s backslash operator. This hints about the sprase structure not being fully exploited here.

3.8 Task #8

Mathematically it is the same equation being solved but using the backslash operator is more efficient than multiplying with the inverse because at each iteration step we have the matrix $\omega L + D$ which is a lower triangluar sprase matrix and the backslash operator sees this and can optimize using only forward substitution instread of the costly matrix multiplication.

After this change was made the script for task #7 was now showed that SOR could solve systems with 2^{22} unknowns. This is the same number as in task #5 where the backslash operator also was used. Thus the SOR implementation is now more efficient!

3.9 Task #10

With $n = 8$ and 100 time measurements, the mean time for solving the original system ($A \times x = b$) with naive Gaussian elimination is 0.008563 seconds and for the perturbed system ($A_p \times x = b_p$) 0.008500 seconds (i.e. 99.26% of the original mean time). We see that there is almost no difference in solving a slightly different system. This is because a perturbation does not matter when we solve for exact solutions — the number of iterations is independent of the actual values in the system which we easily see by looking at the implementation. The slight difference is probably noise in the computation caused by other processes in the system.

3.10 Task #11

With $n = 8$ and 100 time measurements, the mean time for solving the original system ($Ax = b$) with SOR (in Fixed-Point Iteration mode) is 0.003601 seconds and for the perturbed system ($A_p \times x = b_p$) 0.001449 seconds (i.e. 40.23% of the original mean time). We see that it is possible to solve slight different system iteratively with SOR very efficient if we can exploit solutions to an slight different system. This is very useful in real time applications such as signal processing because the sample rate is so high that not much will change from each sample to the next.

Lets compare the results with the one found previously for GE.

$$\frac{\text{mean}_{\text{SOR,orig}}}{\text{mean}_{\text{GE,orig}}} = \frac{0.003601}{0.008563} = 42.053\% \tag{4}$$

$$\frac{\text{mean}_{\text{SOR,pert}}}{\text{mean}_{\text{GE,pert}}} = \frac{0.001449}{0.008500} = 17.047\% \tag{5}$$

We see that SOR is much better; especially for a perturbed system.

3.11 Task #12

When redoing the previous two tasks but increasing k as much as posible it turns out that for GE 2^7 systems could be measured 100 times to get average times for the original as 13.95055 seconds and for the perturbed 13.94964 seconds (i.e $\frac{Pert}{Orig} = 99.99\%$ of the original mean.) For SOR it was 2^{21} and 14.77553 seconds for the original system and 7.76254 seconds for the perturbed (i.e. $\frac{Pert}{Orig} = 52.54\%$ of the original mean). These

k:s ($log_2()$) of the system size) represent how large systems we can have when we want to measure the average solve time 100 times for GE and SOR as will be done in the next exercise. The best n, the one that works for both GE and SOR, is $n = 2^7$.

3.12 Task #13

The results from running both GE and SOR 100 times for 8 perturbed systems are shown in table 3.12. We see that the total means does not deviate so much from the mean times found in task #12. The only thing I notes is the 7th perturbed system solved with SOR that had an average notworty lower than the rests. I belive that the discrepancy is caused by the fact that the *MATLAB*® process is running on a shared system where the OS decides what processes will get execution time.

	GE [s]	SOR [s]
	13.9126260200	0.0034260700
	13.8981966600	0.0032473300
	13.8661602900	0.0034195100
	13.9028463100	0.0030180800
	13.8716484400	0.0032055200
	13.9065633500	0.0032118300
	13.8556906300	0.0034037200
	13.9142215300	0.0032026400
Total mean	13.8909941537	0.0032668375

Table 3: Average times when solving 8 perturbed systems 100 times each.

Redoing the calculations from task #11 gives:

$$\frac{\text{mean}_{\text{SOR,pert}}}{\text{mean}_{\text{GE,pert}}} = \frac{0.0032668375}{13.8909941537} = 0.023518\% \tag{6}$$

This quota (6) is significantly lower than (5). This proves the efficiency of using the iterative method SOR with initial guess being the solution (or approximation) to a known system. This is the final and most important point in the report.

4 Lessons Learned

From this project several theoretical understandings are gained.

- Successive Over-Relaxation
- Sparse matrices and how it affects numerical analysis.
- Benefits of using iterative methods on large systems that does not deviate much from a system with a known solution.
-

In this project I've learned a lot about *MATLAB*®. It has been used in numerous coursed for Computer Scientist at LTH but not in any course have we really learned how to use it. After this project I really feel that I know how to use. This course or one including learning *MATLAB*® should have been included early in the education and not in the third year. More specifically:

- I feel more comfortable in using the data types.
- Language control structures.

- How to use and write functions properly.
- Exception handling.
- Formatted strings.
- Execution time measurements.
- Running scripts as batch jobs.
- Using sparse matrices.
- Global and persistent variables.
- Many other details ...

5 Acknowledgments

I first implemented all tasks my self. Then I discussed the results with friends and adjusted some small things.

In exercise 3 in this course I, Gustaf Waldemarson and Erik Jansson implemented a simpler version of SOR. I used this as the basis for the SOR implementation in this project.

With Tommy Ivarsson, Oscar Olsson and Tommy Olsson I discussed the stopping criteria for SOR. With the both Tommys and Gustaf Waldemarson I also discussed the results found in task #13.

References

[1] T. Sauer, "Numerical analysis," pp. 119–120, 2006. First edition.

[2] T. Sauer, "Numerical analysis," p. 76, 2006. First edition.

[3] T. Sauer, "Numerical analysis," p. 114, 2006. First edition.

[4] T. Sauer, "Numerical analysis," pp. 91,93, 2006. First edition.

Appendix

A Program listings

Here the *MATLAB*® functions and scripts used to achieve the results above are listed.

A.1 Scripts

The following scrips are the drivers for producing the results found in this report.

src/task2.m

```
pr1.init_task(2)
format short

n = 8;
dia_elems = [-1 3 -1 0.5];
 % A is the sought sparse matrix. b=A[1...1]'
[A, b] = pr1.make_mat(n, dia_elems);
spy(A)
full(A)
b
```

src/task3.m

```
pr1.init_task(3)

n = 8;
dia_elems = [-1 3 -1 0.5];
maxtime = Inf;
[A, b] = pr1.make_mat(n, dia_elems);

% x is the solution to the system Ax=b found by naive Gaussian
    elimination.
% with back-substitution.
x = pr1.naive_gauss(A, b, maxtime)
```

src/task4.m

```
pr1.init_task(4)

% How to batch-run:
%j = batch('task4');
%wait(j);
%diary(j);
%load(j);

n = 8;
maxtime = 5*60*60; % In seconds.
global debug_nnz;
debug_nnz = true;
pr1.time_out(maxtime);

can_solve = true;
while can_solve
    try
        [A, b] = pr1.make_mat(n);
        x = pr1.naive_gauss(A, b);
        fprintf(1, 'Solved for k=%i.\n', log2(n));
        n = bitshift(n, 1);
```

```
    catch ex
        fprintf(1, '%s %s\n', 'Exception:', ex.message);
        can_solve = false;
    end
end
fprintf(1, 'Under %i seconds sparse systems of 2^%i unknowns can be solved with naive Gaussian elimination.\n', maxtime, (log2(n)-1));
```

src/task5.m

```
pr1.init_task(5)

n = 8;
maxtime = 5*60*60; % In seconds.
pr1.time_out(maxtime);

can_solve = true;
while can_solve
    try
        [A, b] = pr1.make_mat(n);
        x = A\b;
        fprintf(1, 'Solved for k=%i.\n', log2(n));
        n = bitshift(n, 1);
        pr1.time_out();
    catch ex
        fprintf(2, '%s %s\n', 'Exception:', ex.message);
        can_solve = false;
    end
end

fprintf(1, 'Under %i seconds sparse systems of 2^%i unknowns can be solved with MATLABs backslash operator.\n', maxtime, (log2(n)-1));
```

src/task6.m

```
pr1.init_task(6)

n = 8;
[A, b] = pr1.make_mat(n);
x_cor = ones(n,1);
w = 1.2;
x0 = zeros(n,1);
tol = 1e-4;
maxit = 2^10;
maxtime = Inf;

it_step = 1e-4;
lowest_iter = Inf;

result = [];
for w = 1:it_step:2
    try
        [x, iters] = pr1.sor(A, b, x_cor, w, x0, tol, maxit, maxtime);
        result = [result; w iters];
        if iters < lowest_iter
            lowest_iter = iters;
            clear best_w
            best_w = w;
        elseif iters == lowest_iter
```

```
            best_w = [best_w w];
        end
    catch ex % maxit reached.
        %fprintf(2, '%s %s', 'Exception:', ex.getReport());
    end
end

if length(best_w) == 1
    ws_str = sprintf('=%f', best_w);
else
    ws_str = sprintf('∈[%f, %f],', min(best_w), max(best_w));
end

fprintf(1, 'With a limit of %i iterations and w-resolution of %f
    chosen in the interval [1,2], the best relaxation parameter is
    w%s which gives a solution after %i iterations.\n', maxit,
    it_step, ws_str, lowest_iter);

fig = figure('visible','off'); % Don't display the plot.
plt = plot(result(:,1), result(:,2), 'g');
%set(fig ,'visible','on') % Enable plots again.

xlabel('Relaxation \omega')
ylabel('Iterations')
title('SOR iterations as a function of the relaxation \omega.')
saveas(plt, '../img/task6_plot.eps', 'eps')
saveas(plt, '../img/task6_plot.png', 'png')
```

src/task7.m

```
pr1.init_task(7)

n = 2^3;
w = 1.2;
tol = 1e-4;
maxit = Inf;
maxtime = 5*60*60; % In seconds
pr1.time_out(maxtime);

can_solve = true;
while can_solve
  try
    [A, b] = pr1.make_mat(n);
    x_cor = ones(n,1);
    x0 = zeros(n,1);
    [x, iters] = pr1.sor(A, b, x_cor, w, x0, tol, maxit);
    fprintf(1, 'Solved for k=%i in %i iterations.\n', log2(n),
        iters);
    n = bitshift(n, 1);
  catch ex
    if strcmp(ex.identifier, 'algo:timeout')
      can_solve = false;
    elseif strcmp(ex.identifier, 'algo:maxit')
      can_solve = false;
    end
  end
end
fprintf(1, 'With w=%.3i systems of 2^%i unknowns can be solved with
    SOR under %i seconds with an iteration limit of %i.\n', w, (
    log2(n)-1), maxtime, maxit);
```

src/task9.m

```
pr1.init_task(9);

n = 8;
dia_elems = [-1 3 -1 0.5];
 % A and b are perturbed from the original system. x is the
     solution to Ax=b.
[A, b, x] = pr1.make_perturbed_mat(n, dia_elems);
full(A)
x
b
```

src/task10.m

```
pr1.init_task(10)

n = 8;
solve_times = 100;
maxtime = Inf;

[original_mean, perturbed_mean] = pr1.duosolve_gauss(n, solve_times
    , maxtime);
percent = (perturbed_mean/original_mean) * 100;

fprintf(1,'With n=%i and %i time measurements, the mean time for
    solving the original system (Ax=b) with naive Gaussian
    elimination is %fs and for the perturbed system (Ap*x=bp) %fs (
    i.e. %.2f%% of the original mean time).\n', n, solve_times,
    original_mean, perturbed_mean, percent);
```

src/task11.m

```
pr1.init_task(11)

n = 8;
solve_times = 100;
maxit = Inf;
maxtime = Inf;

[original_mean, perturbed_mean] = pr1.duosolve_sor(n, solve_times,
    maxit, maxtime);
percent = (perturbed_mean/original_mean) * 100;

fprintf(1,'With n=%i and %i time measurements, the mean time for
    solving the original system (Ax=b) with SOR (in Fixed-Point
    Iteration mode) is %fs and for the perturbed system (Ap*x=bp) %
    fs (i.e. %.2f%% of the original mean time).\n', n, solve_times,
     original_mean, perturbed_mean, percent);
```

src/task12.m

```
pr1.init_task(12)

solve_times = 100;
maxit = Inf;
maxtime = 5*60*60; % In seconds. This is for each test i.e. the
    scripts total max time is 2*maxtime.

% Find largest k for Gauss.
n_ge = 8;
pr1.time_out(maxtime);
```

```
can_solve = true;
while can_solve
  try
    [orig_mean, pert_mean] = pr1.duosolve_gauss(n_ge, solve_times);
    percent = (pert_mean/orig_mean) * 100;
    fprintf(1, 'Solved duo Gauss for k=%i, orig_mean=%.5fs and
        pert_mean=%.5fs (i.e. %.2f%% of the original mean time).\n'
        , log2(n_ge), orig_mean, pert_mean, percent);
    n_ge = bitshift(n_ge, 1);
  catch ex
    can_solve = false;
    fprintf(2, 'Can''t solve more. Time out reached. \n');
  end
end
fprintf(1, 'In %i seconds both the original and perturbed system
    could be solved with Gaussian elimination with a system 2^%i
    unknowns. For the largest system solved; orig_mean=%.5fs and
    pert_mean=%.5fs (i.e. %.2f%% of the original mean time).\n\n',
    maxtime, (log2(n_ge)-1), orig_mean, pert_mean, percent);

% Find largest k for SOR.
n_sor = 8;
pr1.time_out(maxtime);
can_solve = true;
while can_solve
  try
    [orig_mean, pert_mean] = pr1.duosolve_sor(n_sor, solve_times,
        maxit);
    percent = (pert_mean/orig_mean) * 100;
    fprintf(1, 'Solved duo SOR for k=%i, orig_mean=%.5fs and
        pert_mean=%.5fs (i.e. %.2f%% of the original mean time).\n'
        , log2(n_sor), orig_mean, pert_mean, percent);
    n_sor = bitshift(n_sor, 1);
  catch ex
    can_solve = false;
        fprintf(2, 'Can''t solve more. Time out reached. \n');
  end
end
fprintf(1, 'In %i seconds both the original and perturbed system
    could be solved with SOR with a system 2^%i unknowns. For the
    largest system solved; orig_mean=%.5fs and pert_mean=%.5fs (i.e
    . %.2f%% of the original mean time).\n', maxtime, (log2(n_sor)
    -1), orig_mean, pert_mean, percent);

k_best = log2(min(n_ge, n_sor)) - 1; % Subscract one since the n at
     termination is allready doubled.
fprintf(1, '\nThe best n, the one that works for both GE and SOR,
    is n=2^%i\n', k_best);
```

src/task13.m

```
pr1.init_task(13)

nbr_systems = 8;
solve_times = 100;
maxit = Inf;
maxtime = Inf;
n = 2^7; % From task 12.

% GE
means = pr1.multi_solve_gauss(nbr_systems, n, solve_times, maxtime)
    ;
```

```
mean_ge = mean(means);
means_str =  sprintf('%.10fs\n', means);
fprintf(1, 'Solving  %i perturbed systems with GE with 2^%i
    unknowns %i times each gave the following mean solve times with
     a total mean of %.10fs:\n%s\n', nbr_systems, log2(n),
    solve_times, mean_ge, means_str);

% SOR
means = pr1.multi_solve_sor(nbr_systems, n, solve_times, maxit,
    maxtime);
mean_sor = mean(means);
means_str =  sprintf('%.10fs\n', means);
fprintf(1, 'Solving  %i perturbed systems with SOR with 2^%i
    unknowns %i times each gave the following mean solve times with
     a total mean of %.10fs:\n%s', nbr_systems, log2(n),
    solve_times, mean_sor, means_str);
```

A.2 Functions

The following functions implements the algorithms and the rest serves as helper functions to these algorithms and the scripts. To distinguish these from other *MATLAB*®-functions in the global namespace these reside in a own package called *pr1*.

src/+pr1/init_task.m

```
function []= init_task( task_number )
% Initialize a task with the number task_number by clearing
    variables, globals functions etc. in the workspace.
clc % Clear command screen.
format long % Format of floating point numbers.
close all  % Close all figures.
fprintf(1, '-->Task #%i.\n', task_number);
evalin('caller', 'clear all'); % Clear workspace at the caller.
end
```

src/+pr1/make_mat.m

```
function [ A, b ] = make_mat( n, dia_elems )
% Makes a n by n sparse matrix according to the assignment. The
    diagonal
% elements are provided in order of enumeration in the assignment.
    b is
% chosen as b = Ax where x is [1 ... 1]'. Default argument for
    dia_elems
% exists.
if nargin == 1
  dia_elems = [-1 3 -1 0.5];
end

e = ones(n, 1);
elems = zeros(n, length(dia_elems));
for i=1:length(dia_elems)
    elems(:, i) = dia_elems(i) * e;
end

% First create a sparse matrix with dia_elems(end) along the
    diagonal.
A = sparse(n,n);
A = spdiags(elems(:,end), 0, A);

```

```
% Then flip the matrix and add the rest of the elements in
    dia_elems along
% the new main diagonal.
A = fliplr(A);
A = spdiags(elems(:,1:end-1), -1:1, A);

% Define b as the solution to A[1 1 1..]'.
x = ones(n, 1);
b = A*x;
end
```

src/+pr1/make_perturbed_mat.m

```
function [ A, b x ] = make_perturbed_mat( n, dia_elems )
% Will construct a perturbation system of the original Ax=b. Random
    quantities
% in the range [-e^-4, e^4] will be added to the diagonal elements
    of A and
% to the elements of b. The new system is A_p*x=b_p which is
    returned by
% the function.
import pr1.* % Behave like packages in other languages :@
if nargin == 1
  dia_elems = [-1 3 -1 0.5];
end

quantities = rand(1,5)*2e-4 - 1e-4;
dia_elems = dia_elems + quantities(1:end-1);
[A, b] = make_mat(n, dia_elems);
b = b + quantities(end);
x = A\b;
end
```

src/+pr1/naive_gauss.m

```
function [ x ] = naive_gauss( A, b, maxtime )
% A naive Gaussian elimination for the system Ax=b. maxwait it the
    number
% of seconds the caller is prepared to wait. If maxtime is not
    given it is
% assumed that the caller handles the time out self (by self
    calling
% time_out(maxwait)). If debug_nnz defined as a global variable (
    and set to
% true) the number of
% non-zero elements will be printed before and after the
    elimination part.
import pr1.*
if nargin == 3
    time_out(maxtime);
end
global debug_nnz

if ~exist('debug_nnz', 'var')
    debug_nnz = false;
end

n = length(A(:,1));

if debug_nnz
    fprintf(1, 'Before elimination A have %i non-zero elements',
        nnz(A));
```

```
end

% Elimination.
for j = 1:n-1 % For each column (of interest).
    for i = j+1:n % For each row.
        fac = A(i,j)/A(j,j); % Factor to multiply row elements with
            .
        b(i) = b(i) - fac * b(j);
        for t = j:n % For non-eliminated element in row i.
            A(i,t) = A(i,t) - fac * A(j,t);
        end
        time_out();
    end
end
if debug_nnz
    fprintf(1, 'and after %i.\n', nnz(A));
end

% Back-substitution.
x = zeros(n, 1);
for i = n : -1 : 1 % For each row in reverse order.
    for j = i+1:n % For each element to the right of the pivot.
        b(i) = b(i) - x(j) * A(i,j);
    end
    x(i) = b(i)/A(i,i);
  time_out();
end
end
```

src/+pr1/duosolve_gauss.m

```
function [ orig_mean, pert_mean ] = duosolve_gauss( n, solve_times,
     maxtime )
% Solves two systems of n unknowns with Gaussian elimination: Ax=b
    and a
% perturbed
% version A_p*x=b_p solve_times times and returns the mean time of
    solving
% these. A maximum waiting time can be specified in seconds.
%
% Returns the mean times of the original and perturbed systems.
import pr1.*
if nargin == 3
  time_out(maxtime);
end

[A, b] = make_mat(n);
[Ap, bp] = make_perturbed_mat(n); % Ignore the correct solution x.

times = zeros(solve_times,2);
for i = 1:solve_times
  % Original system.
  tic_id = tic();
  naive_gauss(A, b);
  times(i,1) = toc(tic_id);

  % Perturbed system.
  tic_id = tic();
  naive_gauss(Ap, bp);
  times(i,2) = toc(tic_id);
end
orig_mean = mean(times(:,1));
```

```
pert_mean = mean(times(:,2));
end
```

src/+pr1/multi_solve_gauss.m

```
function [ means ] = multi_solve_gauss( nbr_systems, n, solve_times
    , maxtime )
% Solves nbr_systems perturbed systems of size nxn solve_times
    times with GE
% with a script timeout of maxtime.
%
% A vector of length nbr_systems of mean solve times is returned.
import pr1.*
if nargin == 4
    time_out(maxtime);
end

means = zeros(nbr_systems, 1);  % The resulting mean times.

for s = 1:nbr_systems
    [Ap, bp] = make_perturbed_mat(n); % Ignore the correct solution
        x.
    times = zeros(solve_times,1);

    for i = 1:solve_times
        tic_id = tic();
        naive_gauss(Ap, bp);
        times(i) = toc(tic_id);
    end
    means(s) = mean(times);
end
end
```

src/+pr1/sor.m

```
function [ x, iters] = sor( A, b, x_cor, w, x0, tol, maxit, maxtime
    )
% To the system Ax=b, this function will iteratively find the
    solution x
% given a tolerance of error to the given known solution x_cor.
% A - n by n matrix.
% b - Column vector of n elements.
% x_cor - The correct solution to the system used with tol.
% w - Relaxation parameter typically > 1.
% tol - The error tolerance.
% maxit - Maximum number of iterations. Default is infinity;
% maxtime - Maximum number of seconds to wait for finish. If
    exceeded an
% exception is thrown. If not given the caller is assumed to handle
     the
% time out.
%
% Returns the approximate solution x and the number of iterations
    needed.
% Exception is thrown if the number of iterations exceeds maxit.
import pr1.*
if nargin == 6
  maxit = Inf;
elseif nargin == 8
    time_out(maxtime);
end

```

```
n = length(A);
D = spdiags(diag(A),0,n,n);
L = tril(A, -1);
U = triu(A, 1);
iters = 0;
x = sparse(x0);
wLD = w*L + D;
% wLDinv = wLD\eye(n);

while (norm(x - x_cor, Inf) > tol) && (iters < maxit)
%     x = wLDinv*(((1-w)*D*x - w*U*x) + w*b);
    x = wLD\(((1-w)*D*x - w*U*x) + w*b);
    iters = iters + 1;
    time_out();
end

if iters == maxit
    err_str = sprintf('Maximum number of iterations exceeded; iters
        =%i', iters);
    exception = MException('algo:maxit', err_str);
    throw(exception);
end
end
```

src/+pr1/duosolve_sor.m

```
function [ orig_mean, pert_mean ] = duosolve_sor( n, solve_times,
    maxit, maxtime )
% Solves two systems of n unknowns with SOR: Ax=b and a perturbed
% version A_p*x=b_p solve_times times and returns the mean time of
    solving
% these. The solution to the normal system will be the initial
    start values
% for the perturbed system.  Maximum number of iterations and time
    to wait
% can be specified. If not, it is assumed that the caller handles
    the time out.
%
% Returns the mean times of the original and perturbed systems.
import pr1.*
if nargin == 2
  maxit = Inf;
elseif nargin == 4
    time_out(maxtime);
end

w = 1.2;
tol = 1e-8; % TODO what to use?
x0 = zeros(n,1);

[A, b] = make_mat(n);
x_cor = ones(n,1);
[Ap, bp, xp_cor] = make_perturbed_mat(n);

times = zeros(solve_times,2); % Time measurements.
for i = 1:solve_times
  % Original system.
  tic_id = tic();
  [x, iters] = sor(A, b, x_cor, w, x0, tol, maxit);
  times(i,1) = toc(tic_id);
  %fprintf(1, 'Solved original in %i iterations and %.5f seconds.\n
      ', iters, times(i,1));
```

```
31
32    % Perturbed system.
33    tic_id = tic();
34    [xp, itersp] = sor(Ap, bp, xp_cor, w, x, tol, maxit); % Initial
          guess is solution from above.
35    times(i,2) = toc(tic_id);
36    %fprintf(1, 'Solved perturbed in %i iterations and %.5f seconds.\
          n', itersp, times(i,2));
37  end
38  orig_mean = mean(times(:,1));
39  pert_mean = mean(times(:,2));
40  end
```

src/+pr1/multi_solve_sor.m

```
1  function [ means ] = multi_solve_sor( nbr_systems, n, solve_times,
       maxit, maxtime )
2  % Solves nbr_systems perturbed systems of size nxn solve_times
       times with SOR
3  % with a script timeout of maxtime and each system executing maxit
       times.
4  %
5  % A vector of length nbr_systems of mean solve times is returned.
6  import pr1.*
7  if nargin == 3
8      maxit = Inf;
9  elseif nargin == 5
10     time_out(maxtime);
11 end
12
13 means = zeros(nbr_systems, 1); % The resulting mean times.
14
15 w = 1.2;
16 tol = 1e-8; % TODO what to use?
17
18 [A, b] = make_mat(n);
19 x0 = zeros(n,1);
20 x_cor = ones(n,1);
21 % Initial guess that is close to real solution of the perturbed
       matrices.
22 x_init = sor(A, b, x_cor, w, x0, tol, maxit);
23
24 for s = 1:nbr_systems
25     [Ap, bp, xp_cor] = make_perturbed_mat(n);
26      times = zeros(solve_times,1); % Time measurements.
27
28      for i = 1:solve_times
29          tic_id = tic();
30              x = sor(Ap, bp, xp_cor, w, x_init, tol, maxit);
31          times(i) = toc(tic_id);
32      end
33     x_init = x;
34     means(s) = mean(times);
35 end
36 end
```

src/+pr1/time_out.m

```
1  function [] = time_out( mtime )
2  % Will throw an exception when a specified max time has elapsed
3  % since the first call to this function with a parameter. First
       call this
```

```
% function with a max time to initialize the function. Then
    subsequent calls
% should be argument-less and will throw an exception when the
    maximum wait
% time is exceeded.
import pr1.*
persistent maxtime t_start;
if nargin == 1
    maxtime = mtime;
    t_start = clock();
else
    t_now = clock();
    diff = etime(t_now, t_start);
    if  diff >= maxtime
        err_str = sprintf('Execution time exeeded; %is', diff);
        exception = MException('algo:timeout', err_str);
        throw(exception);
    end
end
end
```