

C1 - Assignment 1 Report: Sparse Matrices.

Student Number: 1894945

November 6, 2018

C1 - Assignment 1 Report

Student Number: 1894945

Contents

1	Introduction	2
1.1	Well-posed, direct problems	2
1.2	Numerical Methods	2
1.3	Linear systems	3
1.3.1	Splitting Methods	3
1.3.2	Gauss-Seidel Method	4
2	Implementation	4
2.1	Construction of the Gauss-Seidel function	5
2.2	Testing correctness	5
2.3	Running the program	6
2.4	Results of the tests	6
2.4.1	Varying the matrix size	6
2.4.2	Varying δ	7
2.4.3	Varying λ	8
3	Conclusive remarks	9
3.1	Memory	9
3.2	Performances	9
3.2.1	Possible solutions	10

1 Introduction

1.1 Well-posed, direct problems

The problems that have been addressed in the present assignment are always representable in the form:

$$F(x, d) = 0 \quad (1)$$

where x represents the unknown, d the set of data from which the solution depends on and F the functional relation between x and d . Such types of problem are called *direct problems* ([1]).

Definition 1. Let D be the set of admissible data, i.e. the set of data for which problem (1) admits a unique solution x . Let $d \in D$ and denote by δd a perturbation such that $d + \delta d \in D$ and by δx the corresponding change in the solution, in such a way that:

$$F(d + \delta d, x + \delta x) = 0$$

Then the solution x depends continuously on the data d if

$$\exists \eta_0(d), \exists K_0(d)$$

such that:

$$\|\delta d\| \leq \eta_0(d) \implies \|\delta x\| \leq K_0(d) \|\delta d\|$$

If x is continuously dependent on the data d , then the problem is said to be *well-posed* or *stable*. Whenever such a property is not satisfied, the problem is said to be *ill-posed*.

1.2 Numerical Methods

In the following, it will always be assumed that problem (1) is well-posed. A numerical method for problem (1) consists in a sequence of approximate problems:

$$F_n(x_n, d_n) = 0 \quad n \geq 1 \quad (2)$$

with the underlying expectation that $x_n \rightarrow x$ as $n \rightarrow \infty$, i.e. the approximate solution converges to the exact one.

Definition 2. Consider the problem

$$F_n(x_n, d_n) = 0, \quad n \in \mathbb{N}$$

and denote D_n the set of admissible data for this problem. Then, the numerical method F_n is stable if its solution x_n depends continuously on the data d_n , for all admissible data $d_n \in D_n$.

Definition 3. The numerical method (2) is convergent iff

$$\forall \epsilon > 0, \exists n_\epsilon, \exists \delta(n_\epsilon) \mid \forall n > n_\epsilon, \forall \delta d_n : \|x(d) - x_n(d + \delta d_n)\| < \epsilon$$

where δd_n a perturbation of d_n , d_n is an admissible datum for the n^{th} approximate problem, $x_n(d + \delta d_n)$ the corresponding solution of it and $x(d)$ the solution of the exact problem.

1.3 Linear systems

Consider the following linear system:

$$Ax = b$$

where $A \in \mathbb{R}^{n \times n}$, $x \in \mathbb{R}^n$ and $b \in \mathbb{R}^n$. It is evident that such a problem can be expressed in the form (1) as follows:

$$F(x, (A, b)) = 0$$

In order to obtain the solution x of the given problem, one should generally invert A (assuming A is non-singular). One gets:

$$x = A^{-1}b$$

The standard *direct* method for inverting a generic dense matrix is given by computing its LU decomposition (see [2]). The general computational cost for such a procedure is $\mathcal{O}(n^3)$, so that the method becomes impractical if A is large or sparse, as it happens in the present assignment.

A naive implementation of a direct method to invert a sparse matrix is a computational waste, both in memory and compute time. One would rather neither store the zeros in a sparse matrix, nor multiply by them. Certainly, a direct method can be adapted to do neither of these things, leading to the *sparse direct methods*. Alternatively, one can rely on the so called *iterative methods*. Iterative methods are a class of matrix inversion techniques depending only on the calculation of matrix–vector multiplications, which are typically relatively easy to implement in the case of sparse matrices (see [2]).

The performance of an iterative method is generally evaluated by the number of iterations required to converge to a sufficiently accurate approximation of the actual solution. A generic sparse-matrix–vector multiplication costs $\mathcal{O}(kn)$, k being the average number of non-zero elements in each row. Hence, ideally, one desires an iterative method whose computational cost is either independent of n or scales sublinearly with respect to n , in order to be competitive with direct methods without relying on the sparsity of the matrix.

1.3.1 Splitting Methods

In order to implement an iterative method, it is necessary to generate a sequence of approximations $\{x^{(k)}\}$ to the solution. In the case of a splitting method for the problem $Ax = b$, the sequence is obtained as follows:

$$Px^{(k+1)} = Nx^{(k)} + b$$

or, equivalently:

$$x^{(k+1)} = x^{(k)} + P^{-1}r^{(k)}$$

where $A = P - N$ is the *splitting matrix*, P is the preconditioner matrix and $r^{(k)} = b - Ax^{(k)}$. In particular, the inversion of P , which has to be non singular, has not to cost more than $\mathcal{O}(n^2)$ operations, so that the whole method consists in $\mathcal{O}(n^2)$ operations (the left multiply with N is always this much).

1.3.2 Gauss-Seidel Method

Let the decomposition of A be redefined as $A = D - (E + F)$, where D is the diagonal of A , $-E$ and $-F$ are the upper-triangular and lower-triangular components of $A - D$.

The Gauss-Seidel method is defined by setting $P_G = D - E$ and $N_G = F$. Clearly, inverting P_G costs $\mathcal{O}(n^2)$.

It can be proven (see [2], [3]) that, if $A \in \mathbb{R}^{n \times n}$ is symmetric and positive definite, then the Gauss-Seidel method converges for any $x^{(0)}$.¹ As a matter of fact, the convergence of the iterations depends on $\rho(P^{-1}N)$, $\rho(B)$ being the spectral radius of the matrix B . More precisely one can show the following to hold true:

Theorem 1. *For $b \in \mathbb{R}^n$ and $A = P - N \in \mathbb{R}^{n \times n}$, if P is non-singular and $\rho(P^{-1}N) < 1$, then the iterates of the splitting scheme $Px^{(k+1)} = Nx^{(k)} + b$ converge to $x = A^{-1}b$ for any $x^{(0)}$.*

Proof. Let $e^{(k)} = x^{(k)} - x$ be the error at the k^{th} iterate. Since $P(x^{(k+1)} - x) = N(x^{(k)} - x)$ one has:

$$e^{(k+1)} = (P^{-1}N)^k e^{(0)}$$

which concludes the proof. □

Taking advantage of the triangular form of P , the updating of the i^{th} component of x at the $(k+1)^{th}$ iteration can be written in the following form:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left[b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^n a_{ij}x_j^{(k)} \right] \quad i = 1, \dots, n \quad (3)$$

Hence, in the Gauss-Seidel method, at the $(k+1)^{th}$ step, the available values of $x^{(k+1)}$ are used to update the solution.

The algorithm is terminated on the residual $r^{(k)}$, meaning that the iterations stop when

$$\|r^{(k)}\| < TOL$$

TOL being some error tolerance. It is, however, not clear with respect to which norm the previous condition has to be satisfied: even though the choice a norm is completely subjective, L_∞ and L_2 norms are the ones most commonly used (see [2]). In the following, the L_∞ one will be used.

2 Implementation

The program has been built with the aim to minimise the memory used to store the matrices. Hence, following the instructions given in the assignment, the STL container *vector* has been used to store both the matrix entries and the corresponding column indexes. However, instead of using simple vectors, vectors of pointers have been used. Such a solution provides a more efficient storing of both entries and indexes, as it can be directly seen by comparing the memory allocated by the program with the case in which simple vectors are used.

¹Clearly, in order to improve convergence of the algorithm, an optimal choice of the *initial guess* $x^{(0)}$ can to be made.

The initialisation of an instance `SparseMatrix` is implemented in the following way: the constructor builds two vectors of vectors of pointers, setting the length of the first containers to the size of the matrix, but not initialising the content of the nested containers and setting the size of the rows and columns of the matrix. When a value and an index need to be stored, the *addEntry* function is called and if the row at which the value has to be inserted is non-existing, it is dynamically created: both the value and the index are pushed in. If the row exists already, a simple `push_back` is performed.

2.1 Construction of the Gauss-Seidel function

The core of the whole program is clearly represented by the function performing the Gauss-Seidel algorithm. Such a function has been chosen to be a member function and has been built to accept 7 parameters: the initial guess x_0 , the vector to invert against b , the tolerance required by the method (*tol*), the number of iterations after which a check for stagnation has to be made (*itCheck*), two files on which the residual error at every iteration and the solution are printed respectively, the maximum number of iteration the user is willing to wait for the algorithm to converge (*MaxIter*). Looking at Eqn. (3), it is clear that the elements of x_0 can be overwritten as they are computed in the algorithm and only one storage vector is needed: x_0 has then been passed by reference to the function, so that no copy of it has to be produced.

It is worth pointing out that the matrix entries needed for overwriting the elements of x_0 are obtained through the member function *getValue*.²

In case the matrix is not a square matrix and the sizes of x_0 and b do not equal the row and column size of the matrix, the program exits with an appropriate warning to the user. If this is not the case, the iterations start. The program runs until either the residual error becomes smaller than the given tolerance or the maximum number of iterations *MaxIter* has been reached. In order to make sure that the program has not stagnated, every *itCheck* iterations, the current residual error, in the L_∞ norm, is compared with the one computed and stored *itCheck* iterations before: if the residual has not decreased, the program exits with an appropriate stagnation warning to the user. An appropriate warning is printed on the terminal if the program exist due to having reached the maximum number of iterations the user has fixed.

The sudo code for the implementation of the algorithm can be found either in Ref.[2] or on Wikipedia and will not be reported here.

2.2 Testing correctness

In order to check the correctness of the implementation of the Gauss-Seidel algorithm, the tests requested by the assignment have been performed: for this reason, a non-member function `Gauss_Seidel_test` has been built. Such a function takes 6 arguments: the size of the matrix A that has to be initialised to perform the test, the parameters δ and λ necessary to the implementation of the same tests, the fixed tolerance (*tol*), the number of iterations after which a check for stagnation has to be made (*itCheck*), the maximum number of iteration the user is willing to wait for the algorithm to converge (*MaxIter*). The last 3 parameters will be used by the member function `Gauss_Seidel` previously introduced, within the function `Gauss_Seidel_test`.

The vectors x_0 , b , ω and D and the parameter $a = 4(\delta - 1)$ necessary to construct the matrix A are initialised by this function. The entries of A are added by means of the member function *addEntry*.

²More comments on this choice can be found in §3.2

The files on which the residual error at every iteration and the corresponding final solution are printed, are identified by the value of δ , the value of λ , by the strings *Residual* or *solution* and the size of the matrix with on the output files.

The function prints on the terminal the error between the exact and the found solution, indexing it through the value of λ , δ and the size N of matrix A .

2.3 Running the program

The program is run through the Makefile provided. The current optimisation is set to -Ofast and all following tests have been performed with this optimisation choice. Every possible error or warning has been explicitly checked by means of the instruction -Wall -Wfatal-errors -pedantic (that can still be found commented out in the Makefile) before proceeding to performing the different tests: no warnings appeared.

The program generates a fixed number of output files, in which the results of the tests have been stored. Some of these files are used to produce plots by means of two different plotscripts.

2.4 Results of the tests

In the following, the results of the more significant tests performed will be reported and briefly discussed. The tolerance has been set to 10^{-6} for each of the tests.

2.4.1 Varying the matrix size

The Gauss_Seidel_test function has been shown to work for the case of $N = 100$, $N = 1000$ and $N = 10000$, N being the size of the matrix, as required by the assignment. The time T_{10000} required to reach convergence in the case $N = 10000$ has been explicitly computed through the following instructions:

```
auto start = std::chrono::high_resolution_clock::now();

auto finish = std::chrono::high_resolution_clock::now();
std::chrono::duration<double> elapsed = finish - start;
std::cout<< "Elapsed time: " << elapsed.count() << "s\n";
```

and found to be $T_{10000} \approx 83$ seconds. The algorithm has thus been shown to work in quite a short time, so that the program can be regarded as being reasonably efficient.³ The solutions x_0 obtained for the three different cases can be found in the files *d_1.000000_L0.000000_GSsolution_100.txt*, *d_1.000000_L0.000000_GSsolution_1000.txt* and *d_1.000000_L0.000000_GSsolution_10000.txt*. The residual error at every iteration has been printed in corresponding *Residual* files.

Finally, the distance between the exact solution and the approximate one is printed on the terminal. Unsurprisingly, the error becomes bigger as the size of the matrix grows.

³More comments on the efficiency of the code can be found in §3.2.

2.4.2 Varying δ

The performance of the algorithm has been tested for different values of δ , as required by the assignment. Having shown that the algorithm converges in a reasonably short time even in the case of $N = 10000$, the size of the matrix for this series of tests has been fixed to $N = 100$.

In order to investigate the change in performance for different order of magnitudes, 10 different values of δ , ranging from 1.0 to 10^5 have been used.

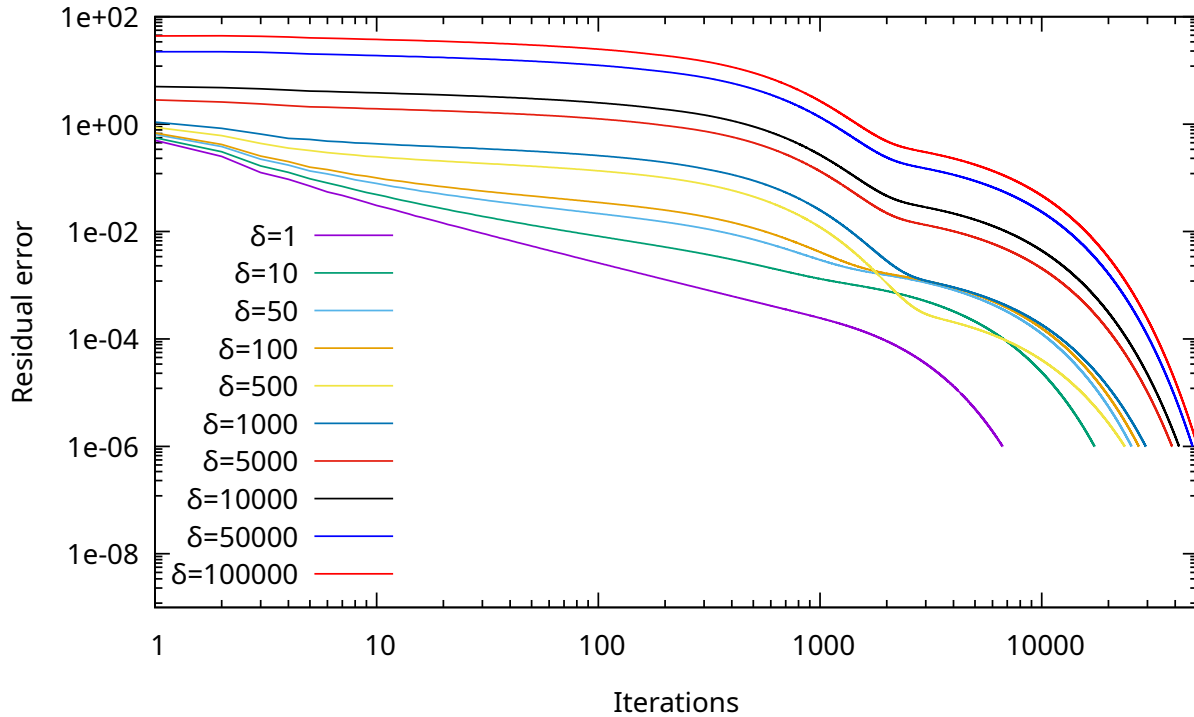


Figure 1: Graphical representation of the performance of the algorithm for different values of δ . To point out the differences between the the results at different orders of magnitude, both the iterations and the residual error in the L_∞ norm have been plotted on a logarithmic scale.

It is clear that the desired closeness to the exact solution, measured in terms of the residual error, is reached independently of how large the initial residual is. This is not surprising: in §1.3.2 it has been pointed out that for symmetric and positive definite matrices the method converges for every initial guess x_0 . As expected, the residual error is a monotone decreasing function of the iterations. For $\delta \geq 50$, a concave region in the residual error profile is noticeable, signalling a change in the “velocity” of convergence. Interestingly, such a region is particularly steep for $\delta = 500$ and an overlap with the curves associated with smaller δ is present.

Clearly, the number of iterations required to have a residual error smaller than the fixed tolerance grows with δ , i.e. the code becomes less efficient when δ grows. In other words, the spectral radius of the matrix $P^{-1}N$ (see §1.3.2) gets smaller for increasing δ .

2.4.3 Varying λ

The performance of the algorithm has been tested for different values of λ , as required by the assignment. Having shown that the algorithm converges in a reasonably short time even in the case of $N = 10000$, the size of the matrix for this series of tests has been fixed to $N = 100$.

In order to investigate the change in performance for different order of magnitudes, 10 different values of λ , ranging from 1.0 to 10^5 have been used.

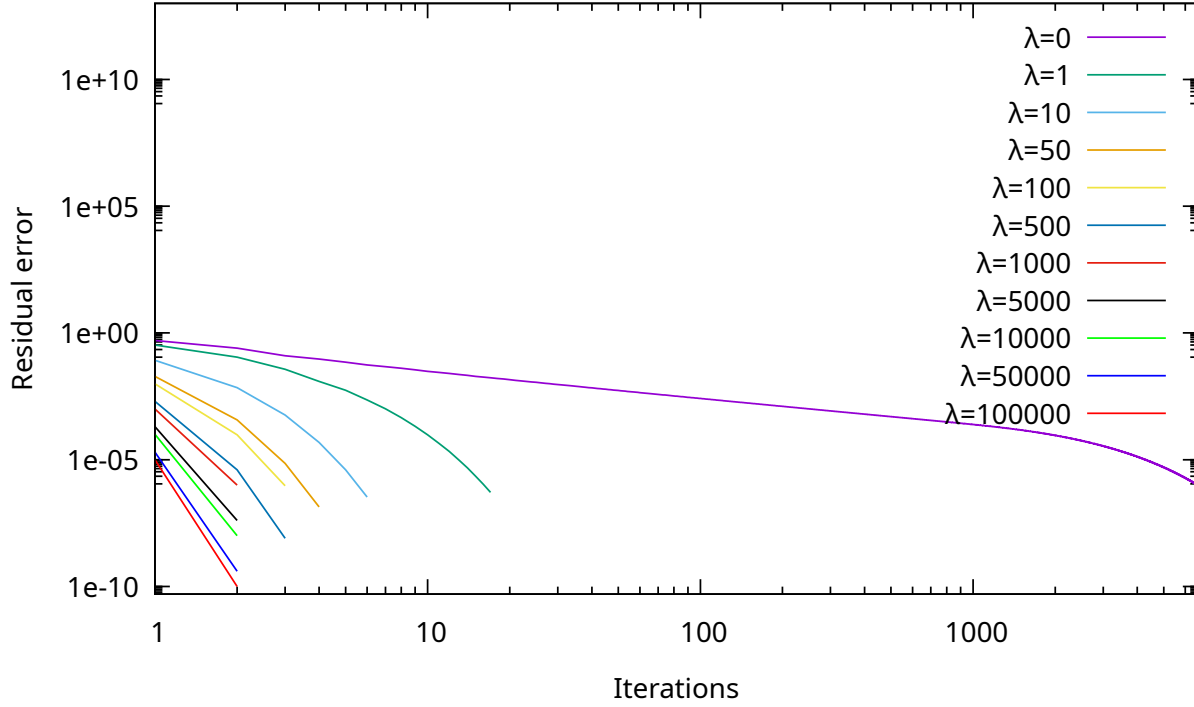


Figure 2: Graphical representation of the performance of the algorithm for different values of λ . To point out the differences between the the results at different orders of magnitude, both the iterations and the residual error in the L_∞ norm have been plotted on a logarithmic scale.

It is clear that the presence of λ changes the efficiency of the algorithm. In particular, the plot shows that for $\lambda \geq 10^3$, the desired convergence is reached in just two iterations.

This can be understood by recalling what has been mentioned in §1.3.1 and §1.3.2 about the decomposition of A into $P_G = D - E$ and $N_G = F$. The presence of $\lambda \in \mathbb{R}^+$ modifies the diagonal component of A and consequently the matrix D . The preconditioner matrix P thereby obtained is different from the one of the initial problem and this results in a faster convergence of the method. As a matter of fact, for $\lambda \in \mathbb{R}^+$ the matrix becomes strictly diagonal dominant. Differently from the case of δ , the spectral radius of $P^{-1}N$ becomes smaller and a faster convergence is obtained.

3 Conclusive remarks

In this section some additional comments and observation are presented.

3.1 Memory

When dynamically allocating memory, one has always to make sure to free the reserved locations as soon as they are no longer needed. With the help of the software *valgrind*, possible memory leaks have been checked. To do this, is it necessary to compile with *-g* and *-O1* optimisation. After having compiled the program, the following instruction has been used:

```
$ valgrind --leak-check=yes ./sparsematrix
```

The following output has been produced on the terminal:

```
==4200== Memcheck, a memory error detector
==4200== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==4200== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==4200== Command: ./sparsematrix
==4200==
==4200==
==4200== HEAP SUMMARY:
==4200==      in use at exit: 0 bytes in 0 blocks
==4200==    total heap usage: 1,252,428 allocs, 1,252,428 frees, 989,616,540 bytes allocated
==4200==
==4200== All heap blocks were freed -- no leaks are possible
==4200==
==4200== For counts of detected and suppressed errors, rerun with: -v
==4200== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

One can clearly see that no memory has been lost, so that everything has been deleted correctly. It may seem a little strange at fist that all the heap blocks were freed, even though no explicit call to the destructor has been made. However, all instances are built by the function *Gauss_Seidel_test*, so that the deletion is automatically performed every time the programs exits the function. The absence of memory leaks is however non trivial, since it proves that the destructor has been constructed correctly. This can be explicitly checked by means of commenting out the last two lines of the destructor: if *valgrind* is called again, some memory leak will be found.

3.2 Performances

In order to test performance, the following instructions have been used⁴:

```
$ valgrind --tool=callgrind ./sparsematrix
$ kcache-grind
```

⁴The test has been performed only on the part of the code concerned with the test over different λ and δ , commenting out the part on different N in the main file.

The first command analyses the performances in the terms of load distribution, while *kcachegrind* is used to visualise the load distribution results. Results are reported in the following screenshot.

















Incl.	Self	Called	Function
 100.00	0.00	(0)	 0x00000000000001090
 99.99	0.00	1	 _start
 99.99	0.00	1	 (below main)
 99.99	0.00	1	 main
 99.99	0.01	20	 Gauss_Seidel_test(unsigned int, double, double, double, int, int)
 99.96	14.37	20	 SparseMatrix::Gauss_Seidel(std::vector<double, std::allocator<double>>, std::vector<int, std::allocator<int>>>
 59.74	59.74	215 049 776	 SparseMatrix::getValue(int, int) const
 41.57	15.91	308 991	 SparseMatrix::operator*(std::vector<double, std::allocator<double>>, std::vector<int, std::allocator<int>>>

Figure 3: Load distribution of the program.

It is clear that a better efficiency could have been obtained if the program had not relied so much on the *getValue* function, which is not particularly efficient.

3.2.1 Possible solutions

A possible way to optimise the code is to sort the indexes and values as soon as they are entered, so that each index can be associated to the corresponding value in a unique way and without the necessity to call a the *getValue*. Such a solution, however, has not been implemented, since even though it is clearly more efficient for the matrix given in the assignment, it is not immediately clear how faster and optimised the code could become in the case of denser sparse matrix. On the other hand, it is still reasonable to expect that such a choice will make the program faster, since the sorting takes place only at the beginning of the program and *getValue* is called a large number of times during the iterations.

Possibly, the best solution is to store the values and the indexes by means of the *insert* function instead that through a *push_back*: this way, the values will be automatically sorted.

References

- [1] A. Quateroni, R. Sacco, F. Saleri; *Numerical Mathematics*, Vol.37, Springer Verlag, (2007).
- [2] T. Grafke; *Scientific Computing*, Lecture Notes, University of Warwick, (2018).
- [3] W. Hackbush; *Iterative Solution of Large Sparse Systems of Equations*, Springer-Verlag, New York, (1994).