Discrete Optimization of The Sparse QR Factorization

Using Heuristic Branch & Bound search

Jakob Østergaard
5th of October 1998

IMM – DTU[1]
UNI•C[2]

[1]Institute of Mathematical Modelling, Technical University of Denmark
[2]Danish Computing Center for Research and Education

This project was conducted at the Institute of Mathematical Modelling, at the Technical University of Denmark, and at UNI•C, Danish Computing Center for Research and Education, during the late spring, summer, and autumn of 1998.

The original idea behind this project, came from Claus Bendtsen, of UNI•C. In short, he was interested in investigating whether discrete optimization of QR factorization would be beneficial, if at all possible, in relation to another project he was involved in.

During the time of the project, I have had much good advice, and some interesting and most enlightening conversations with Professor Jens Clausen, of IMM, DTU, who supervised this project.

Abstract

When using certain mathematical models for simulating aerial pollution, it becomes interesting to be able to solve some specific system of linear equations as fast as possible. This report describes an attempt made, to optimize the performance of a solver using sparse QR factorization, and back-substitution.

The optimization problem is complex, and it quickly turns out that attempting an exhaustive search thru the solution space is absurd. A branch-and-bound algorithm is developed, and using different search strategies and heuristics, solutions of varying quality are found.

I will present the methods I used for finding "good" solutions, as well as some of the approaches that failed. This is by no means a thorough mathematical treatment of the problem. It is a document describing a number of approaches taken, and their usefulness in actual applications.

Jakob Østergaard, October 1998, Lyngby

Contents

Chapter 1

Introduction

As explained briefly in the abstract, we are faced with the problem of solving a system of linear equations. This itself is usually not a hard problem. Solving it efficiently however, proves to be a much greater challenge.

1.1 The origin of the system

The system we are facing, is used in the process of solving a set of partial differential equations, describing how pollutants react with each other in the atmosphere.

It should be of no surprise to the reader, that chemical reactions between a number of species can be written in differential form. Let's look at a little example of this. Consider a system consisting of four species, each with concentrations c_1, c_2, c_3 and c_4. 1 and 2 react with each other, producing 3, while 4 decays producing 1. The concentration of species 1, c_1 is thus given as:

$$\dot{c}_1 = -c_1 c_2 + \alpha c_4 \tag{1.1}$$

where α indicates how fast c_4 is decaying. This of course, is a very simple model of a reaction.

When modelling aerial pollution, we work with a fairly large number of species. This number N depends on the model in use, but is often between 36 and 56, but can go as high as more than 100 species for the most complex models.

Working with many species, invites us to write this reaction scheme on a more compact form. For every species in our model, we are given the equation:

$$\dot{c}_i = \sum_{j,k} \beta_{ijk} c_j c_k + \sum_j \alpha_{ij} c_j \,. \tag{1.2}$$

In real applications however, the coefficients α and β are often zero, since most chemicals only react (notably) with a small subset of the others.

The *implicit Euler* method is used for solving the resulting system of differential equations. The system of equations is written short as $\dot{c} = f(t, c)$. The Euler method lets us take small steps h in time, thereby finding the concentrations of the species at any given time. A step in time, from some given initial condition c_0 is given by

$$c = c_0 + hf(t, c). \tag{1.3}$$

Or, put in another way, we find the concentrations after a time-step by finding a root in the equation

$$g(c) = c_0 - c - hf(t, c). \tag{1.4}$$

Simple Newton iteration yields

$$c^{i+1} = c^i - \left(\frac{\partial g}{\partial c}(c^i)\right)^{-1} g(c^i). \tag{1.5}$$

The Jacobi matrix $\mathbf{J}$ for f is given by $\partial f/\partial c$. Thus,

$$\frac{\partial g}{\partial c} = -\mathbf{I} + h\frac{\partial f}{\partial c} = -\mathbf{I} + h\mathbf{J}. \tag{1.6}$$

Finding the concentrations of the species seems to be the "simple" matter of iterating thru equation 1.5. That is in a sense true, except from the fact, that iterating thru that equation involves *inverting* $-\mathbf{I}+h\mathbf{J}$. This is the reason we actually bother to discuss the efficient solution of a linear system of equations in the first place. It should also be clear to the reader, why it is so important that the solution of that system of equations run efficiently.

The reaction schemes doesn't change often. They are usually developed and implemented, and then used for years, for research and forecasts etc. Therefore, if we can only solve the model in an efficient way, it is perfectly reasonable to spend much time optimizing the solver, since an improvement there will benefit the users of the model for years.

1.2 Properties from a numerical point of view

The system of linear equations

$$\mathbf{A}x = b$$

can be solved in numerous ways. However, in our specific application, we can exploit the following properties:

- $\mathbf{A}$ is sparse
- Although the elements of $\mathbf{A}$ change, the sparsity *pattern* of $\mathbf{A}$ is invariant.

We suggest, that QR factorization followed by simple back-substitution may be used in solving the above system, even though QR factorization is usually considered more expensive than *Householder* and other factorizations. The QR factorization however, is insensitive to changes in the elements in $\mathbf{A}$. This means, that if we find *one* good row/column ordering, and one good sequence of Givens Rotations to complete the QR factorization, that single ordering can be used for all occurring values of $\mathbf{A}$. If we used a non-orthogonal factorization, we would have to create a new factorization for every new value of $\mathbf{A}$.

The idea in this project is to use one good ordering of rows and columns, and one good sequence of Givens rotations, for all values of $\mathbf{A}$ (as long as $\mathbf{A}$ maintains the same structure). All in all, they together should minimize the computational cost of the factorization and the following back-substitution, by minimizing the number of Givens rotations needed, and minimizing the number of fill-ins produced during the factorization.

Chapter 2

The Sparse QR Factorization

The QR Factorization will factorize our $\mathbf{A}$ matrix into a $\mathbf{Q}$ and an $\mathbf{R}$ matrix, such that $\mathbf{QR} = \mathbf{A}$. The $\mathbf{Q}$ matrix is orthogonal, and the $\mathbf{R}$ matrix is upper triangular.

One can get an impression of the interesting properties of orthogonal factorizations by considering the following trivial — but nonetheless interesting property:

$$\mathbf{QR}x = b \quad \text{and} \quad \mathbf{R}x = \mathbf{Q}^T b$$

Once we know $\mathbf{Q}$, solving the original system is a simple matter of back-substitution, since $\mathbf{R}$ is upper-triangular.

2.1 The Factorization

A Givens rotation can zero out one element in a row, using some other row. The rotation can be seen as:

$$A(:,[i,j]) \leftarrow \begin{pmatrix} c & s \\ -s & c \end{pmatrix}^T A(:,[i,k]) \tag{2.1}$$

where $c = cos(\theta)$ and $s = sin(\theta)$, for some θ, which clearly guarantees orthogonality.

Contrary to eg. the simple Gauss elimination, we will not only introduce non-zeros (called fill-ins) in the row holding the element we are eliminating, but also in the eliminating row. The orthogonality of the transform should hopefully make up for this, by not requiring us to pivot rows in order to guarantee numerical stability.

An example where "x" is used to represent some arbitrary non-zero, and "0" is used to represent the zero elements, could be:

$$\begin{pmatrix} x & x & x & 0 \\ x & x & 0 & 0 \\ 0 & 0 & x & x \\ 0 & x & x & x \end{pmatrix} \begin{pmatrix} c & s & 0 & 0 \\ -s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} x & x & x & 0 \\ 0 & x & f & 0 \\ 0 & 0 & x & x \\ 0 & x & x & x \end{pmatrix}$$

The "f" denotes a "fill-in". In order to completely factorize the system, we will ofcourse have to apply more Givens Rotations. But the above example should demonstrate how one Givens Rotation affects the system.

Whenever a fill-in is introduced below the diagonal, it will require one extra Givens rotation in order to eliminate it. A fill-in above the diagonal, will require more work in the final back-substitution.

2.2 The Back Substitution

When our system is all factorized, we perform the back-substitution. This is trivial, but included for the sake of completeness. Back-substitution of one row in a dense system can be written as

$$x(i) \leftarrow (x(i) - \mathbf{R}(i, i+1:n)b(i+1:n))/\mathbf{R}(i,i) \tag{2.2}$$

where $x(n) \leftarrow b(n)/\mathbf{R}(n,n)$ is the special case for the last element in x, eg. the one we solve for first. Obviously, equation 2.2 requires $2(n-i)+2$ FLOPS, for every element in x. This is back-substitution on a dense system. If b or $\mathbf{R}$ turns out to be sparse, we will save some work.

2.3 Computational costs

Obviously we are not very keen of the thought of using trigonometric functions throughout the factorization. Fortunately, there exists an efficient formula for calculating the rotation coefficients c and s from above. The following is taken from [MC]. If we want to eliminate element b using element a, assuming both are non-zero, the coefficients are given by:

$$|b| \geq |a| \quad : \quad \tau = -a/b; \quad s = 1/\sqrt{1+\tau^2}; \quad c = s\tau \tag{2.3}$$

$$|b| \leq |a| \quad : \quad \tau = -b/a; \quad c = 1/\sqrt{1+\tau^2}; \quad s = c\tau. \tag{2.4}$$

This formula costs us 5 FLOPS plus a square-root. (For the pedantic, it should be mentioned, that the case $|a| = |b|$ will probably never occur, and which one of the two formulas above are used in this case does not matter).

The actual rotation of two rows, given by repeating equation 2.1, will cost us $6p+2q$ FLOPS, if p is the number of elements in the rows rotated where both rows have non-zeros, and q is the number of elements where only one row has a non-zero.

Finally, the back-substitution of one row costs us approximately $2p+2$ FLOPS, where p is the number of non-zeros in the row currently being worked on.

Depending on the hardware on which we run the actual factorization, the square root used in equations 2.3 and 2.4, can cost anything from one FLOP to "many" FLOPS. We take the liberty to account one FLOP for every square-root used, but this is easily changed in the actual optimizer.

Chapter 3

The Optimization

In short, we wish to find a row- and column ordering, and a sequence of Givens Rotations, that together will produce a QR factorization of our system of equations with a minimal computational cost.

Now, there are $(2n)!$ combinations of row- and column orderings, if our system has n equations with n unknowns. And if we have k non-zeros in the system, we will need $\mathcal{O}(k)$ Givens Transforms, giving us $\mathcal{O}(k!)$ combinations there. All in all, we have $\mathcal{O}((2n)!k!)$ possible combinations. For the 56×56 system with roughly 12% non-zeros, this amounts to a number of combinations in the order of 10^{185}.

Clearly, exhaustive search thru the solution space is not an option. We will have to apply some strategy, in the hope that we can somewhat limit out search to some small number of small sub-spaces of the solution space.

3.1 Defining the problem

In order to be able to apply any of the widely used search strategies we will be considering for use in the project, we will need to define some way of performing an "incremental search" on our problem. A search where we can take steps, and decide which way to go after every step. This is necessary in order to limit the size of the subspace we seek thru.

Now, what is our solution space ? It could be seen as many things. We could search in one space first, then in another, eg. ordering the rows first, then the columns and finally the rotation sequence. Or perhaps we should decide upon the rotation sequence first ?

Some time was spend in the very beginning of this project, experimenting with different strategies, and ways of searching. I came to several conclusions doing this:

1. Attempting to find the optimal rotation sequence on a system is hard
2. Until we have a complete ordering of the rows and columns, we can say little or nothing about an optimal rotation sequence

With this in mind, I decided upon the strategy that is used in the final optimizer.

The strategy is: First, decide which row should be the upper-most row, and which column should be the left-most column. Calculate or estimate the cost of this

partial solution. Then, decide on the second-upper-most row, and the second-left-most column, calculate or estimate again, and so on. For each of these partial orderings, we find a suitable sequence of Givens Rotations in order to be able to estimate a lower bound on the cost of the current solution. More on this later.

As one notices, we do not at all consider the rotation sequence in this ordering strategy. In fact, the actual rotation sequence is found from a very simple algorithm, which has some roots in theory and some roots simply in experience gathered:

- We always use diagonal-elements for eliminating non-zeros below the diagonal.
- We eliminate non-zeros column-wise, from the left to the right.
- If there is more than one row with a non-zero in the current column, we choose the row with the smallest weighted sparsity distance to our row holding the diagonal-element.

3.1.1 Sparsity distance

The word "sparsity distance" was mentioned above. This is a metric for which I could find no existing name. If one exists I may be confusing things, if not, I invented a metric.

Consider two rows in a sparse matrix. If the two rows are identical, eg. they have zeros and non-zeros at the same column positions, the sparsity distance is 0. If their patterns differ by one non-zero somewhere, the distance is 1, and so on.

The weighted sparsity distance, is a variant of the simple metric from above. The weighted metric gives different weights to pattern differences at different locations. A pattern difference which is located below the diagonal in the system matrix is given a higher weight than a difference located above the diagonal. This is usable in relation to QR factorization, because introducing non-zeros below the diagonal is more expensive than introducing them above the diagonal (below the diagonal will require Givens rotation, above the diagonal requires only back-substitution).

Both distance measures can be shown to be true metrics, eg. they obey the following:

$$
\begin{aligned}
&d(a,b) \geq 0 \\
&d(a,b) + d(b,c) \leq d(a,c) \\
&d(a,b) = d(b,a)
\end{aligned}
$$

This was a very loose description of the metrics. They are however not important for the understanding of the optimizer, and they are only used in a small isolated part of the program. I felt however, that they deserved to be mentioned.

3.2 The choice of a search strategy

Many strategies are available, when one wants to search a solution space for some feasible solution. The only one we have out ruled so far, is exhaustive search.

Several strategies where considered:

- Simulated annealing
- Branch and Bound

- Tabu search

A description of these and other strategies can be found in [JC].

Both simulated annealing and tabu search rely somewhat on the fact that once a variable has been altered, we can be fairly sure, that next time we alter that variable, it will influence the result in nearly the same way. This is not at all the fact for our problem. The generation of fill-ins completely spoil that property. Placing one row/column in some position will have completely different impacts on our result, depending on the position of all other rows/columns.

From the simple analysis above, I decided to focus my efforts on branch and bound search. Early in the project, tabu search was implemented to some extent. But it was rather unsuccessful, even compared to the likewise very early implementation of branch and bound I had running. Therefore, I sticked with branch and bound, and decided to only work with the other two, if they suddenly became interesting due to some change of (my understanding of) the properties of the problem. That never occurred.

3.3 Branch and Bound

In the field of Discrete Optimization, Branch and Bound is a popular way of limiting the solution space to include only potentially relevant subspaces.

In short, the B&B algorithm begins in the root of the search tree. A number of branches is generated, and depending on the variant of the search (depth first, breadth first, best first, or hybrids), some branch is elected for exploration.

If we're able to predict what the smallest possible value we can ever get from our target function will be (the cost of factorizing the system given some row/column ordering), from descending some given branch, we are able to discard all branches we encounter which does not yield any possibility for an improvement (eg. a lower value of the target function). This is why the algorithm – quite intuitively – is called branch and bound.

3.3.1 Branching and Bounding

Branch and Bound algorithms produce some number of possible new branches, from some starting point. On these branches, a cost function (called $f(s)$) representing the current cost of the solution, plus some penalty for not having finished the task yet is defined. Furthermore, we define a lower-bound (called $g(s)$) on the cost, which is used for bounding our search.

When we have some number of possible new branches to search thru, we can ignore all branches having a $g(s) \leq f(\text{best})$, and their entire sub-trees.

If the $g(s)$ function is close to the true cost of completing the search along some branch, we will be able to cut away most of the unnecessary branches. If the function is not very close, however, we will still have to search thru some very large domain.

Often the quality of $g(s)$ and the time it takes to compute is a compromise, and one will have to experiment to find a viable routine. However, if the quality of $g(s)$ is low, nothing else in the search algorithm will be able to compensate for this.

3.3.2 Defining the branch

What exactly is it we are branching ? In order to apply B&B search, we need to perform some sort of incremental solution of our optimization problem.

The approach I chose, is to fix some row and some column at the left-most upper-most position. Since many rows and columns may yield a valid sub-solution (eg. place a non-zero in the diagonal), we are often able to generate multiple branches.

For every branch we are again able to fix some other row and some other column in the second-left-most and second-upper-most positions. Again we probably have more than one choice, and therefore a number of sub-branches. At all times, the active branches are kept in the list of "live nodes".

The process of fixing a column and a row can be visualized as below:

$$\left(\begin{array}{c|ccc} x & x & x & x \\ \hline 0 & 0 & x & x \\ x & x & 0 & 0 \\ 0 & x & x & 0 \end{array}\right) \Longrightarrow \left(\begin{array}{cc|cc} x & x & x & x \\ 0 & x & x & 0 \\ \hline x & x & 0 & 0 \\ 0 & 0 & x & x \end{array}\right) \tag{3.1}$$

This example only changes the row-ordering, but column orderings could just as well have changed simultaneously.

When we reach the point where we have fixed all columns and all rows, we have reached a final solution. The cost of this solution (found simply by factorizing it and recording the number of FLOPS required), will determine whether it will be the new best-solution, or simply discarded.

3.3.3 Routine overview

The B&B routine is fairly modular. It consists of four main parts:

- The main B&B strategy routine
- The Branch generator
- A branch sorting routine
- The heuristic calculation routine

The blocks interact as sketched in figure 3.1. Of course this presentation of the B&B algorithm is overly simple. Many other routines are involved, some of the major ones are memory handling and target function ($f(\cdot)$) calculation.

The responsibilities of the four major blocks are:

B&B strategy This routine traverses the list of live nodes, calling the branch generator for every live node it encounters. Occasionally it will call the sort algorithm. When this routine runs out of live nodes, the search is complete, and the current best solution is returned as the overall best solution.

Branch generator The branch generator generates branches for some given live node. If one of the branches turns out to be a final solution, and that solution is better than the currently best, the best solution is updated, and the sort algorithm invoked. The branches which seem promising (based upon the heuristic) are inserted in a pseudo-sorted manner into the list of live nodes. If a branch has a lower estimated cost than the first live node, it's inserted

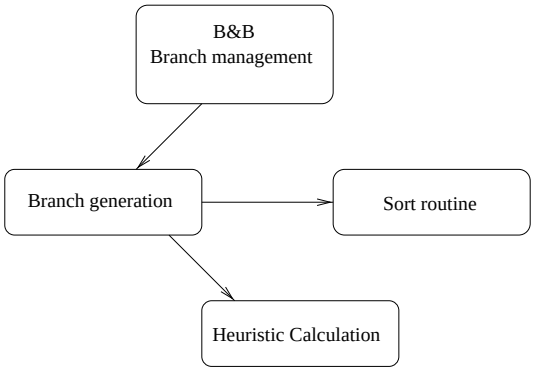

Figure 3.1: The branch and bound routine

first. Otherwise, the branch is simply inserted as number two in the list of live nodes. This special ordering of new live nodes contributes to the hybrid nature of the search, and places it somewhere between best-first and depth-first. The ordering was experimentally found to be good.

Sorting routine The sorting routine sorts the list of live nodes after their $h(s) + f(s)$ values, eg. the estimate of what a possible completion of the search thru the live node may cost. If this was called every time a new live node was to be investigated, the search would be a best-first. The fact that the sort algorithm is only called occasionally, makes the search a hybrid between depth-first and best-first.

Heuristic calculation Calculation of the heuristic $h(s)$. This is the routine that ultimately has required the most time and efforts. Providing a good heuristic in realistic time is not at all simple. Many approaches where tried, some of which are described in chapter 4. The one that made it into the final optimizer, calculates a possible ordering of rows, given some column ordering, and returns the resulting cost as the heuristic.

3.3.4 Why Heuristics ?

Searching thru the search-tree involves computation of lower bounds, where some are required, but most usually belong in branches which are discarded. The better the bounding function, the fewer unnecessary lower bounds will have to be computed.

It is not simple to calculate a good lower bound. Usually, improvements in the lower bound function yields much higher computational costs of the function. However, no matter how fast the lower bound is to calculate, it will never make it up for a bad bound. The choice between a slow but good, and a faster but worse bounding function remains a compromise.

In order to find a reasonable minimal cost for evaluating the equation system using some ordering (every branch represents a possible ordering), we must go thru the QR factorization process. The ways the different rows of the system affects each other, is simply too complex to easily approximate using some other function. At least, I completely failed to find an easier way, not that I didn't try.

Thus, in order to guarantee that the lower bound is really a lower bound, we should optimize the system before calculating the target function. This is a little too

recursive to do us any good. Instead, a simple ordering of the system is made, and that system is factorized. This means, that our lower bound is not necessarily a lower bound anymore. It is a heuristic.

Using heuristics means, that a solution to our problem may no longer be the optimal solution. This is bad if we end up far from the optimum. However, a well-behaved heuristic may well yield results that are "good" if not optimal. It truly worried me not being able to guarantee the quality of the solution. However, the heuristic used seems to behave well, and the solutions returned absolutely seems to make sense. Furthermore, it did not seem to be a viable solution to recursively order the system every time a bound calculation is made. Note, that bound calculations for the 56×56 system some times take place more than a billion times (for some ugly systems). If every calculation would involve a B&B search with an average depth of perhaps 30, the result – however good – would be of use to no one by the time it would finish.

3.4 The Heuristic

The heuristic that is used in the final optimizer, uses a fixed column ordering and orders the non-fixed rows to form a legal system. The cost of factorizing this resulting system is then used as the heuristic.

The non-fixed columns are ordered so that the columns with the most non-zeros are positioned right-most in the system. They are sorted so that the number of non-zeros are descending as we move to the left. This is not necessarily optimal, but it proves experimentally to be "good".

Ordering the rows, given the column-ordering, is done using bipartite matching. Any matrix holding some number of non-zeros can be seen as a bipartite graph. Finding a bipartite matching on this graph, will again yield an ordering of the rows guaranteeing non-zeros in all of the diagonal elements. This is illustrated in figure 3.2.

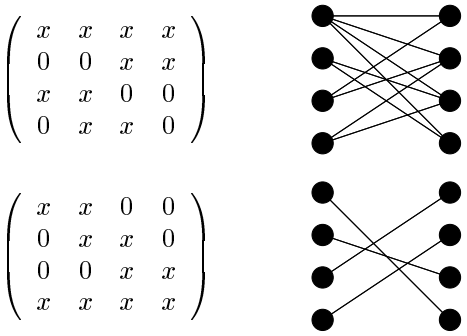

Figure 3.2: Matrix and bipartite graph before and after matching

Luckily, bipartite matching runs in close-to $\mathcal{O}(n)$ time and memory, given n columns and rows.

The drawback of bipartite matching is, that we do not necessarily find an ordering of the rows which is anywhere near optimal (wrt. minimizing fill-ins in the following QR factorization). However, this will prove to be of little importance. The quality of the ordering is still good enough, and the run-time efficiency seems to make it up for any extra branches descended due to the worse heuristic.

3.4.1 Measuring the Quality of Heuristics

The fact that we do not examine all possible row-orderings when calculating our heuristic suggests that we may be getting costs far away from the realistic optimal costs. For some reason (which I do not entirely understand, but do appreciate), the heuristic based on a simple and fast bipartite matching yields results pretty close to the true cost.

Looking at the values of the actual cost, the heuristic, and the sum of these, as we progress down the branches to a final solution, will show how well the heuristic matches the expectations. (Ideally the $g(s) + h(s)$ function would be a horizontal line). Such a sampling of the heuristic and cost values, is found in figure 3.3. This plot is taken from the best solution to the system describing the chemical reactions used in the aerial pollution model.

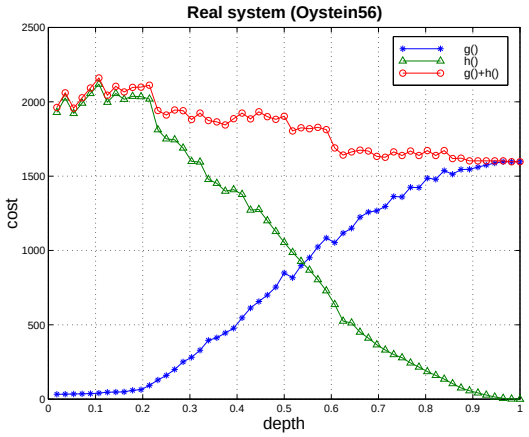

Figure 3.3: $h(s), g(s)$ and $h(s)+g(s)$ down the search tree. Depth 0 is the beginning (top) of the search tree, and depth 1 is the end (where final solutions are)

As can be seen in the figure, the heuristic somewhat overshoots the actual cost. This is compensated for, by multiplying the heuristic with some "acceptance" factor. Per default, the optimizer uses an acceptance factor of 0.68. This value was found to work well in a wide range of systems. Experimenting with this value will sometimes yield better results, but often at the expense of a severely increased execution time of the optimizer (of course due to the much larger number of accepted branches in the search).

Chapter 4

Approaches that failed

As indicated earlier, it soon came clear that the calculation of a reasonable heuristic in reasonable time would be the greatest challenge in the optimization problem. This section will outline some of the approaches I followed and later abandoned.

Some of the ideas that I describe here may seem naive, but at the time they where explored, they seemed as viable approaches that at least was worth some attention. I hope that this chapter will be – if nothing else – a help to anyone who might continue this work or work with similar problems. Perhaps some of the approaches could actually be turned into usable algorithms, given more thought and work.

4.1 Using dynamic programming

The heuristic calculation basically consists of two steps: The first finds a suitable ordering of the rows and columns not yet fixed by the B&B search tree descent. The second step factorizes the system and returns the cost as the heuristic.

A simple algorithm for finding a suitable ordering of rows given a fixed column ordering, simply fixes one suitable row, then recursively attempts to fix the remaining rows. If this fails, another row is fixed as the first, and the recursion attempted again. If everything fails, no valid ordering (an ordering with non-zero diagonal) exists. If we succeed, we can either just use this ordering, or generate all possible orderings. Either way, this problem is $\mathcal{NP}$ hard.

I liked this approach because of its obvious simplicity. It was easy to generate all possible orderings, or just some. But the factorial run-time of the algorithm made it virtually unusable on systems of dimensions above $n = 30$, or just more dense systems. It can easily be seen, that the algorithm will explode in run-time as either the dimension or the density of the system increases.

One way to work around the non-polynomial run-time of this algorithm would be to apply dynamic programming, or "memorization". If we remembered how some sub-set of rows could be ordered, chances seemed to be good, that this sub-set would need to be ordered very often, and thus if it was memorized and the ordering could be retrieved fast, we would save a recursive descent saving us from some of the $\mathcal{NP}$ unpleasantness.

Storing and retrieving the ordering of some sub-set of the rows needs to be done as fast as possible. I chose a trie as the data structure for storing the solutions. Tries yield a search time which is constant with respect to the number of stored

solutions, and linear with respect to the dimension of the system. The overall algorithm seemed viable to this point.

During a series of experiments, I found two major flaws in this design. The first was, that the "hit-rate" (the ratio between hits and misses) in the search trie was around 1%. No matter how fast I could retrieve the orderings, it was only a minor part of the overall orderings that where to be found that I could actually retrieve from the trie. Thus, the memorization did not improve run-time significantly.

The second pitfall was, that search tries have exorbitant memory requirements. Even for smaller systems ($n \approx 20$) the memory consumption of the program exceeded half a gigabyte. This made the algorithm unsuitable for execution on commodity hardware, and suggested that it would be impossible to treat large systems even on supercomputers.

A third observation is, that even if we could have dealt with the memory consumption and if the hit-rate had been more significant, the algorithm would still have run in non-polynomial time. The need for ordering a large number of sets of rows recursively would still exist. The run-time might have been better measured in real time, but the complexity would still have been non-polynomial, and we would have met the wall of combinatorial explosion for some not-too-large size of system anyway.

These sad observations led to the abandoning of this approach. Perhaps storing the orderings in a balanced tree (eg. a red-black tree) would solve the memory consumption problem. Perhaps a more sophisticated search logic would be able to find more usable memorized solutions, and thus improve the hit-rate. I doubt this approach will lead to anything as good as what I use in the final optimizer, but still I felt it deserved to be mentioned.

4.2 Finding the complete set of possible orderings

Another approach came from the observation, that often the actual number of possible orderings was small, although it could seem huge if only a small sub-set of the rows where considered. This is because some rows put severe restrictions on the positioning of other rows, thus limiting the number of orderings significantly. If all possible orderings of some sub-set could be generate at a moderate cost, all these orderings could be factorized, and we would be able to choose the minimal cost as the heuristic, thus producing a better (closer to lower-bound) heuristic.

This approach builds upon the idea of fixing one row position, but remembering what rows where candidates for that position. When fixing the next row, we hold the candidates together with the previous set of candidates, thus producing a set of possible orderings for these two rows. We continue this way, until we end up holding the set of possible orderings for the sub-system we where to calculate the heuristic for.

For the system

$$\begin{pmatrix} x & 0 & x & x \\ 0 & x & x & 0 \\ x & 0 & 0 & x \\ 0 & x & x & 0 \end{pmatrix}$$

this can be illustrated as

$$\begin{aligned} 1: &\quad \{[r_1 \to p_1], [r_1 \to p_3], [r_1 \to p_4]\} \\ 2: &\quad \{[r_1 \to p_1, r_2 \to p_2], [r_1 \to p_1, r_2 \to p_3], \end{aligned}$$

$$
\begin{aligned}
& [r_1 \to p_3, r_2 \to p_2], \\
& [r_1 \to p_4, r_2 \to p_2], [r_1 \to p_4, r_2 \to p_3]\} \\
3: \quad & \ldots
\end{aligned}
$$

where the mapping $[r_1 \to p_2]$ is interpreted as "row one is fixed in position two". The number to the left of the mappings represent the current "depth", eg. the number of rows we have fixed.

As is already seen in the mappings written before, the number of possible orderings easily explode. It is obviously a $\mathcal{NP}$ hard problem finding all orderings this way. Although the problem is complex, we may apply a little trick, to minimize the complexity. (The problem is still $\mathcal{NP}$ hard, but the actual run-time we see, will improve much with the little trick described below).

Remember that we order the columns ascending from left to right by the number of non-zeros they hold. Also, we should keep in mind that any valid ordering has strictly non-zeros in the diagonal. Thus, the top-most rows must have non-zeros to the left. Further on, we have the least number of choices for placing rows in the top, because the left-most columns hold the least number of non-zeros.

With the above observations, it should be clear, that if we start by fixing rows in the top locations, moving downwards, our algorithm will at all times consider the least number of possible orderings.

This approach produced a good heuristic on systems of sizes up to $n = 40$. The run-time of a system of this size with density around 12% was approximately 6 days on a standard PC. Much of the time was spent in actually calculating the cost of some ordering, because the number of orderings actually found was often large (in the order of 10^5) and the run-time of the ordering itself was small compared to the number of orderings returned.

Limiting the number of orderings factorized to eg. 100, chosen randomly from the set of valid orderings, improved the run-time quite a lot (this was done in the 6 day run mentioned before). Still, the heuristic was good, even though it was only based on a hundred randomly chosen orderings out of several hundreds of thousands possible ones.

Still, the complexity of this approach killed it, when we where considering systems of the more realistic size above $n = 40$. Even though the $\mathcal{NP}$ complexity was limited thru the trick described earlier, the algorithm still was $\mathcal{O}(n!)$ hard. On the system with $n = 56$ and a density of approximately 12% this algorithm did not complete in reasonable time (eg. the author's lack of patience preempted it after a week or so).

Still, based on numerous experiments with large more sparse systems, this algorithm behaves very well and produces good heuristics in reasonable time if only the sparsity can be kept well below 10%. It is absolutely a viable approach if the systems considered are very sparse.

Chapter 5

Implementation

While this project in a sense is quite academic, it is also a project in which the research will lead to no results what so ever, if the algorithms and ideas are not implemented, tested, run, and their results interpreted.

Thus, no matter how terse and thorough the theory is described and treated, a large amount of the time and effort spend in this project has been spent on implementing and testing new ideas.

5.1 An overview of the software package

The software developed can generate unrolled Fortran code which performs the sparse QR factorization and back-substitution in a "hopefully close to optimal" way, on some system of linear equations, given the structure of the system.

Clearly, if one has run an optimization of some system for days, and then decides to change something in the way the resulting Fortran code is generated, it would be rather inconvenient and somewhat redundant to have to re-run the entire optimization. Thus, the software package was split into several programs:

- One program for optimizing the system, the optimizer
- And one program for generating Fortran code from the more abstract description generated by the optimizer.

To give some assurance that the solver generated is correct, a regression test tool was developed. It's a very simple program that solves a large number of random systems (obeying the given structure), keeping track of the maximal relative error, eg. $||Ax - b||/||b||$. This tool should be run whenever a solver has been generated.

5.2 Target platforms

The production code resulting from this project is to be used in super-computing environments, which means the code should run on a vast number of different more or less UNIX like operating systems. Of course, I should use tools readily available on these platforms.

Most of the development was done using the GNU/Linux operating system, running on the author's PC, because of the vast number of development tools of all kinds

available on this platform. But from the very beginning of the project, attention was paid to issues involving porting to different systems offering different subsets of tools and features.

It is my belief, that the tools developed can easily be ported to most UNIX variants. Current requirements from the target platforms are a C compiler and a Perl interpreter. But the tools could be run on one system, and the resulting Fortran code used on a completely different system. Of course, the end-system will need a Fortran compiler.

5.3 Choice of languages

I chose C as the implementation language for the optimizer. This language is readily available on any platform I have ever heard of, and most of the platforms support the standardized ANSI C.

The C programming language allows for sophisticated data structures like trees etc. that would be either very hard or perhaps infeasible to implement in a language like Fortran. Because of the run-time efficiency needed in the optimizer, only languages with Fortran- or C like performance could be considered. I chose C because, I at the beginning of this project felt it would meet my demands. Had I known at that time, what I know now, I would not for a second have doubted that C++ would be the language of choice. C++ offers C or Fortran run-time efficiency, but especially the specialized memory handling I implemented would have been a lot cleaner done in C++ with eg. templates.

The Fortran code generator was written in Perl, since this language offers great string-handling capabilities, and completely frees the programmer from even considering allocation of data. Performance is of no importance in this tool, so the fact that Perl is interpreted (rather than compiled) doesn't matter. The Fortran code should only be generated once for every new optimized system, and even though I didn't give performance a second of thought implementing this tool, it easily generates the tens of thousands of lines of Fortran code needed to solve the 56×56 system in a matter of seconds on a standard PC.

The regression test tool was written in Fortran. This choice was made mostly because the automatically generated solver is also a Fortran program. It's a very simple (read: Spartan) piece of software, and I saw no need to complicate matters further by implementing this tool in eg. C.

5.4 Implementing Branch and Bound

An initial implementation of B&B is fairly easily done. It's first later on, when concerns such as memory utilization, run-time efficiency, heuristics etc. etc. occur, that the implementation becomes a real challenge. This happened indeed.

5.4.1 $\mathcal{O}(1)$ memory handling

Obviously, the B&B routine needs to store every live node, during the search. This itself leads to a large number of required allocations and de-allocations of memory area. Further more, the heuristic in the final implementation require a large number of similar memory operations.

Deallocation is usually efficient, but allocation can require many clock-cycles, and can even have a run-time which is dependent on the number of allocated memory areas, depending on the platform on which the optimizer is run of course. Since both the number of allocated areas, and the number of allocations is mind-boggling, this itself could prove a significant bottleneck in the overall run-time of the optimizer.

Therefore, an efficient memory sub-system was implemented. This sub-system allocates a large (and hopefully large enough) memory area, which is then split in parts of the size needed by the B&B routine and the heuristic. Allocation can then be done by a special function call, and will always run in $\mathcal{O}(1)$ time. De-allocation has the same complexity. If the memory area is exhausted, the system will fail (simply abort the program with an error message). Of course, a more dynamical memory handler could have been implemented, but I found that it wasn't worth the effort, since just allocating a large enough area initially was practically possible. Besides, the page handling in modern operating systems lets us allocate much more memory than we actually use, without taking performance hits.

Chapter 6

Tools

Several tools where developed, during the time of the project. Tools for creating test-systems, for generating actual Fortran code, and for regression testing.

The tools described in the following are only the tools which are of actual use for an end user of the optimizing software. These tools will allow someone to generate Fortran code for solving the actual system optimized, and for testing whether the solver is consistent with the actual system (a simple test of $\mathbf{A}x \doteq b$).

6.1 Fortran Code Generation

A simple tool for generating a solver written in Fortran, from the data returned by the optimizer was built. The tool reads the structure of the system, the ordering of rows and columns, the sequence of Givens rotations applied to solve the system, and generates Fortran code that does just this.

The Fortran routine uses the algorithm described in [MC] section 5.1.8 and 5.1.9. The back-substitution that is finally applied should be trivial.

The fact, that the subroutine takes an unordered system as argument, and returns a x vector corresponding to this system (so that the optimized solver can directly replace any previously used solver with no change of the main program) makes the generated code virtually unreadable. This suggests use of the test utility to verify the correctness of the code.

The code generator generates unrolled code. It is fairly trivial, and will probably work best on cache architectures, since no effort has been put into making the code behave well on deep pipelined architectures. Some effort could be put into making the code generator optimize the code for either cache or vector architectures. This would probably not be too difficult, yet it remains as an exercise for the end user.

6.2 Regression test utility

A generic test program was written, that generates a system holding pseudo-random numbers, solves it using the optimized code, and finally checks the $\mathbf{A}x - b$ residual. The test utility is trivial but should catch most errors anyway.

I have of course tested the generated solvers for numerous systems, and it is my belief, that the optimizer and code generator behaves correctly for any given (valid) system.

Chapter 7

Results

This chapter seeks to document the results achieved with the optimization software developed. A simple quality-of-solution estimate is done based on a comparison between the number of FLOPS used to solve the system of equations in the automatically generated Fortran code, and in MatLab respectively.

Unfortunately, the sparse solvers provided by MatLab are all based on iterative methods. It would have been nice to compare the QR factorization with another direct method. When comparing the results achieved by the generated Fortran code, and the MatLab routines, we should keep this major difference in the underlying methods in mind. This is nothing but a bold comparison of what two fundamentally different methods can achieve. The comparisons are done anyway, in order to give the reader some idea of the quality of the optimized solutions.

In MatLab, I used the `bicg()` routine, which is an iterative solver based on the bi-conjugate gradients method. In all comparisons, I let this method iterate until the default accuracy of $\varepsilon \approx 10^{-6}$ is reached. The typical accuracy of the QR factorization lies around $\varepsilon \approx 10^{-10}$ (this is found by numerous runs of the regression tester on a wide variety of systems). This may be considered in favor of the iterative method, since it would have needed much more iterations to achieve the higher accuracy, but I consider $\varepsilon \approx 10^{-6}$ a sufficient accuracy, and the fact that the QR factorization is better is considered only unnecessarily better.

7.1 Sanity testing on sample systems

In order to document that the optimizer (and the related tools) work as intended on other systems than the single system this project was initiated for, I have run tests on a number of differently structured systems. This indicates, that the optimizer and tools could be applied to a broad range of sparse systems of linear equations.

7.1.1 The completely random system

This is a system with non-zeros spread uniformly all over. The system is quite sparse (only 17.38% non-zeros), but there is no obvious "better ordering". The result from the optimization is expected to show little improvement, but is nonetheless interesting due to the difficult nature of the system.

As expected, the optimization didn't improve the structure of the system much. In figure 7.2, the optimized system is shown, along with the resulting R matrix.

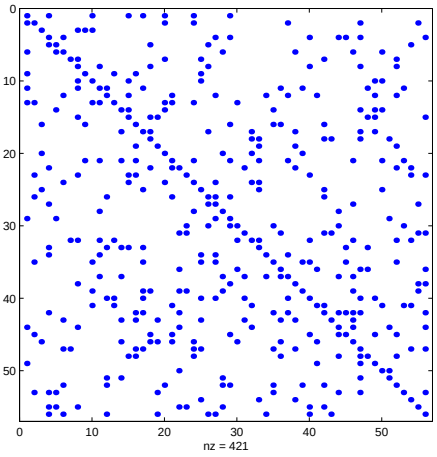

Figure 7.1: The completely random system. $nz = 17.38\%$

The R matrix is almost dense, indicating that fill-ins have been generated heavily throughout the system.

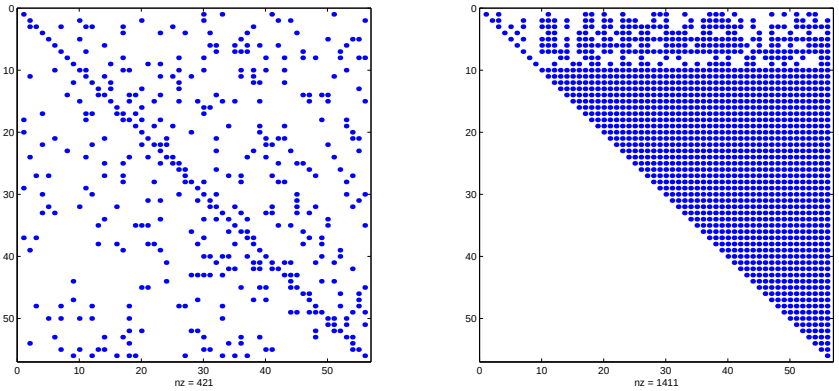

Figure 7.2: Ordered "random" system before and after QR

Solver	KFLOPS
MatLab	351.2
QR	140.8

7.1.2 The more dense random system

Like the previous system, non-zeros are distributed randomly all over the system. This system has a non-zero density of 42.36%, which will cause the B&B routine to generate a much larger number of branches. The run-time of the optimization of this system was roughly two days, while the previous system completed in less than an hour. A glance of the system is found in figure 7.3.

As one might expect, the optimization of this system is close to hopeless. Some non-zeros may have been relocated, yet, the R matrix is dense. Although this system shows a limitation of what we can possibly optimize, and what we cannot,

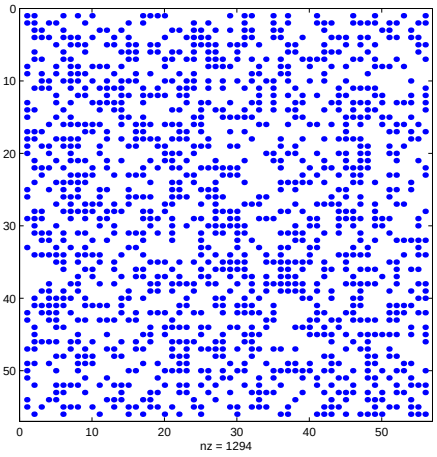

Figure 7.3: The more dense random system. $nz = 42.36\%$

it is also in a way malplaced. We wouldn't consider using a direct sparse method on this system in the first place. I kept the example anyway, because it's a nice demonstration of something inappropriate for optimization.

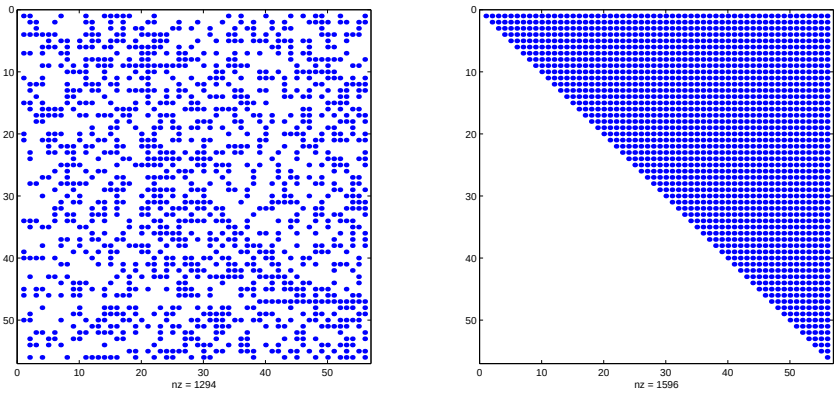

Figure 7.4: Ordered "dense random" system before and after QR

Solver	KFLOPS
MatLab	722.5
QR	293.7

7.1.3 The double-diagonal system

Having seen the impacts of optimization on systems of randomly distributed non-zeros, it is also interesting to look at optimization of not-so-randomly distributed non-zero systems. Two different systems will be considered here. The first one is a system with two diagonals, one from the upper-left to the lower-right corner, and one orthogonal to this. If one thinks about how this could possibly be optimized, one may realize the interesting property of this system: It is not possible to simply move non-zeros from below the main diagonal, without moving another non-zero below it again. The system is shown in figure 7.5.

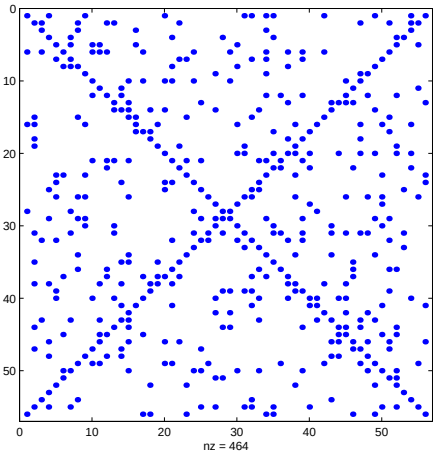

Figure 7.5: The double-diagonal system. $ns = 14.80\%$

This system is the last in the series of hardly-improving systems. Because of the ugly property, that we cannot relocate any of the other-diagonal non-zeros to above the first diagonal, without relocating a non-zero below that diagonal too, we shouldn't expect too much of an optimization. As seen in figure 7.6, the optimization didn't yield much of an improvement. Although the R matrix is still somewhat sparse, the original structure of the system is a "QR-killer". This system demonstrates very well, that although a system may be sparse, it is not necessarily well suited at all, for QR factorization. Not even when put thru the optimizer.

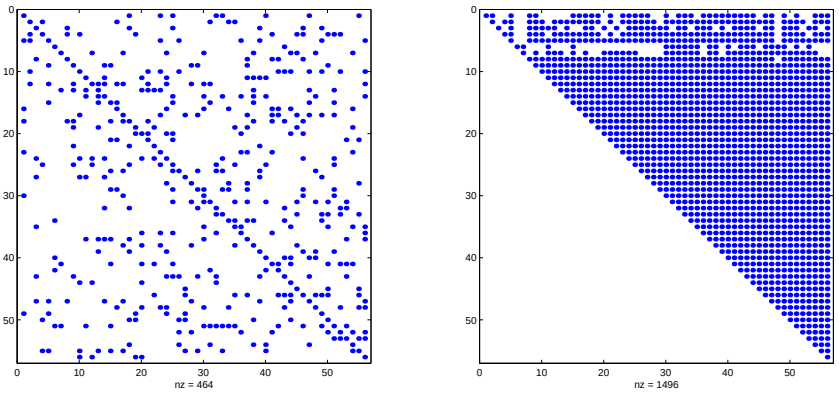

Figure 7.6: Ordered "double-diagonal" system before and after QR

Solver	KFLOPS
MatLab	313.6
QR	160.6

7.1.4 The upper-triangular system

The title of this section is somewhat misleading. This system does indeed have an upper-triangular structure, but unlike the usual upper-triangle, this upper triangle

is located in the upper *left* corner of the system. Thinking of the way QR works, this system seems unsuitable for QR factorization, at least if the rows and columns are not re-ordered. Fortunately, the high concentration of non-zeros in one isolated part of the system poses good opportunities for relocating these non-zeros to somewhere else, where their impact on the total cost of the solution will not be so severe. The system can be seen in figure 7.7.

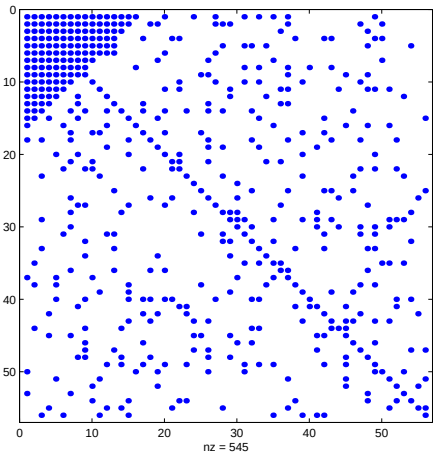

Figure 7.7: The alternative upper-triangular system. $nz = 17.38\%$

As expected, this system did benefit from the optimization. It is clearly seen in figure 7.8, that the high concentration of non-zeros in the most unfortunate (from a QR point of view) position in the upper left corner, has been relocated to positions mostly above the diagonal, and the ones which are below the diagonal are located near the very bottom of the system, to minimize their impact on the rows below them (by minimizing the number of rows below). The R matrix is fairly dense though. This implies, that the back-substitution will be fairly expensive, comparable to back substitution on a dense system. The QR factorization however, will be fairly cheap (again compared to a dense factorization), because of the large number of non-zeros below the diagonal.

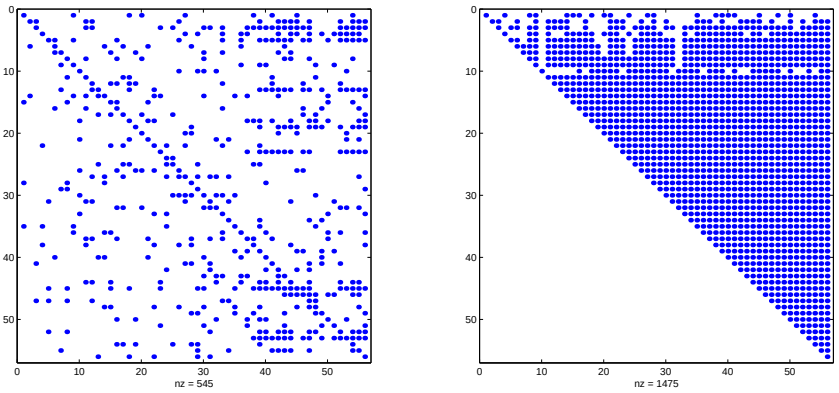

Figure 7.8: Ordered "upper triangular" system before and after QR

Solver	KFLOPS
MatLab	357.0
QR	146.2

7.2 The real system

Finally, we will see how well the optimizer behaves when given the system for which it was originally created. This system was given to me by Ph.D. Zahari Zlatev of DMU[1]. It describes the structure of the system derived from a scheme of chemical reactions, as described in section 1.1. The system treats 56 species, and is thus of dimension 56×56 as the systems previously examined in this chapter. The system holds 11.74% non-zeros, where some are located near the top and near the left side of the system. Others are distributed fairly randomly (from a statistical point of view – I'm sure the researchers behind the system will disagree that the distribution is anywhere near random).

This initial distribution of the non-zeros will behave very bad with QR factorization. We will generate a large number of fill-ins, and besides that, a large number of Givens rotations will be needed in order to factorize the system. Fortunately, it looks as though there are good opportunities for re-arranging some rows and columns to make the system better suited for sparse QR. Let's see. The initial system can be seen in figure 7.7.

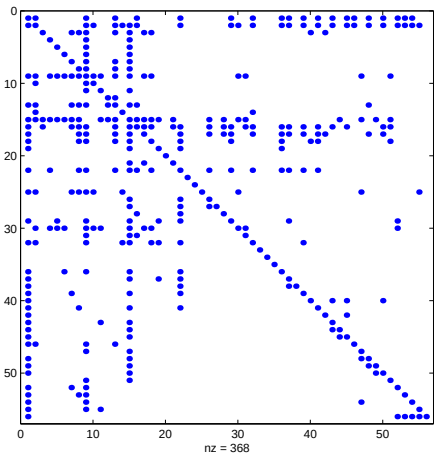

Figure 7.9: The real system system. $nz = 11.74\%$

The optimization on this system proved to be beneficial. The optimized system truly shows a structure much better suited for sparse QR, than the original system did. The R matrix is still somewhat sparse, although it is clearly seen, that the big savings – compared to a dense factorization – doesn't come from the back-substitution.

Solver	KFLOPS
MatLab	367.3
QR	25.9

[1] Ministry of Environment and Energy, National Environmental Research Institute, Department of Atmospheric Environment

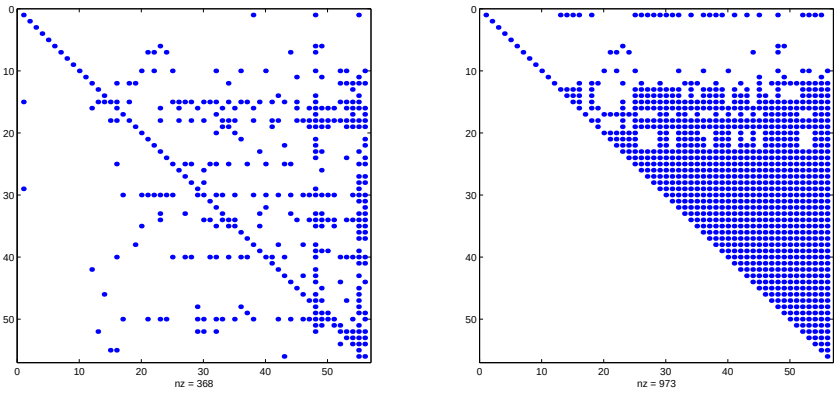

Figure 7.10: Ordered system before and after QR

It is however interesting to see, how horrible the MatLab performance on this system is. MatLab requires an order of magnitude more FLOPS to solve this system to some reasonable accuracy. Had the system been better conditioned, the iterative method would have behaved better.

Chapter 8

Conclusion

The original goal of this project was to investigate whether it was possible to optimize a QR factorization – by means of discrete optimization of various properties of the factorization – so that it would be a viable alternative to some existing iterative solvers for sparse systems.

We ended up with an optimizer and some related tools for code-generation and testing, that optimizes the row- and column ordering of a system of linear equations, and finds a suitable sequence of Givens rotations to apply in order to completely factorize the given system. Plain Fortran code can be automatically generated, to solve the system specified.

It seems from the results in chapter 7, and results gathered from numerous test-runs not documented here, that the row- and column ordering yields substantial improvements on *some* systems, whereas other systems may have a structure not suitable for optimization (several examples are given in the results chapter).

Even if a system of linear equations does not have a structure which is obviously well suited for optimization, a solver generated by means of the tools developed during this project, may still be very efficient compared to iterative alternatives. The only real demand from the tools is, that the system must be sparse. The system on which the tools are used, should not hold more than 15 – 20% non-zeros.

Only time will tell, whether the solvers generated from the software developed here, will find use in the simulation software for which it was originally developed. Unfortunately this project ends here, and the comparison with the currently used iterative solver – and eventually other solvers – will not make it into this report.

From where I stand, it looks as though we are going to incorporate code generated by these tools, into the aerial pollution simulation software. At least to see if it yields the performance gain it seems to promise in this report.

Bibliography

[ITA] **Introduction to Algorithms**, *Thomas H. Cormen, Charles E. Leiserson and Ronald L. Rivest*, The MIT Press, Cambridge, Massachusetts London, England

[MC] **Matrix Computations**, *Gene H. Golub and Charles F. Van Loan*, The Johns Hopkins University Press.

[JC] **Branch and Bound Algorithms – Principles and Examples**, *Jens Clausen*, Department of Computer Science, University of Copenhagen.

[RS] **Row Ordering for a Sparse QR Decomposition**, *Thomas H. Robey and Deborah L. Sulsky*, Society for Industrial and Applied Mathematics, Matrix Anal. Appl. October 1994.

[PL] **Generelle Optimeringsheuristikker**, *Per S. Laursen*, Datalogisk Institut (DIKU), Københavns Universitet, Marts 1994.

[GO] **Ordering Givens Rotations for Sparse QR Factorization**, *M. I. Gillespie and D. D. Olesky*, Society for Industrial and Applied Mathematics, Matrix Anal. Appl. July 1995.