Les Cahiers du GERAD ISSN: 0711–2440

Sparse solutions of linear systems via maximum feasible subsets

J.W. Chinneck

G–2018–104

December 2018

La publication de ces rapports de recherche est rendue possible grâce au soutien de HEC Montréal, Polytechnique Montréal, Université McGill, Université du Québec à Montréal, ainsi que du Fonds de recherche du Québec – Nature et technologies.

Dépôt légal – Bibliothèque et Archives nationales du Québec, 2018
– Bibliothèque et Archives Canada, 2018

The publication of these research reports is made possible thanks to the support of HEC Montréal, Polytechnique Montréal, McGill University, Université du Québec à Montréal, as well as the Fonds de recherche du Québec – Nature et technologies.

Legal deposit – Bibliothèque et Archives nationales du Québec, 2018
– Library and Archives Canada, 2018

Sparse solutions of linear systems via maximum feasible subsets

John W. Chinneck

GERAD & Department of Systems and Computer Engineering, Carleton University, Ottawa (Ontario), Canada K1S 5B6

chinneck@sce.carleton.ca

December 2018
Les Cahiers du GERAD
G–2018–104

Abstract: Algorithms for finding sparse solutions of underdetermined systems of linear equations have been the subject of intense interest in recent years, sparked by the development of compressed sensing, where this is a central problem. There are also reasons for interest in finding sparse solutions for general linear systems of equations and inequalities. This paper describes several new and effective linear-programming-based algorithms for both underdetermined sets of linear equations, and general linear systems, derived from algorithms for solving the maximum feasible subset problem. The algorithms allow greater compression and potentially better denoising in the compressed sensing application, and are the first methods applied to general linear systems. Extensive experimental results are reported.

Acknowledgments: This research was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

1 Introduction

A sparse solution is one in which only a small fraction of the variables take values other than zero; these nonzero variables are called the *supports*. A solution having k supports is called k-sparse. Algorithms for finding sparse solutions of linear systems have been the subject of intense research since the development of compressed sensing in the 2000s (e.g. (Donoho 2006)), which permits the compression of a sparse signal into a space much smaller than otherwise expected. A critical step in compressed sensing is the recovery of a sparse solution from a set of linear equations. Standard methods in compressed sensing such as Basis Pursuit (Chen et al 2001) require that the input signal be very sparse. This paper develops improved algorithms for the recovery of sparse input signals that have more supports than can be handled by methods such as Basis Pursuit. This permits greater signal compression. Variants also offer promise of better denoising.

Sparse solutions of general linear systems that include inequality constraints and variable bounds as well as equalities are also useful, but this is a much less studied topic. This paper develops a number of methods for finding sparse solutions for general linear systems as well.

The algorithms developed here are based on methods for solving the *maximum feasible subset* (*maxFS*) problem, which is this: given an infeasible set of linear constraints, find the largest cardinality feasible subset. The sparse solution problem can be formulated as an instance of *maxFS*. Heuristic solutions for the NP-hard *maxFS* problem date to the early 1980s (e.g. Bordetskii and Kazarinov 1981), well before the explosion of interest in compressed sensing. A series of very effective linear programming based methods were developed beginning in the 1990s (Chinneck 1996, 2001). Surprisingly, the *maxFS* solution approach has not been taken up by the compressed sensing community: a single paper (Jokar and Pfetsch 2008) explores the possibility of using known *maxFS* algorithms to solve the central problem posed by compressed sensing.

Jokar and Pfetsch (2008) compare seven methods for solving sparse systems of underdetermined linear equations in a compressed sensing scenario:

1. Three methods from the compressed sensing literature: LP-based Basis Pursuit, Orthogonal Matching Pursuit (Pati et al. 1993) which includes a nonlinear step, and a linearized version of Orthogonal Matching Pursuit,
2. Four methods from the *maxFS* solution literature: an exact branch-and-cut method (Pfetsch 2008), a nonlinear method due to Mangasarian (1999), a nonlinear bilinear formulation (Bennett and Bredensteiner 1997), and an LP-based heuristic by Chinneck (2001).

Their results show that the *maxFS* heuristics by Mangasarian and Chinneck provide the best results, with Chinneck's method the best overall. This motivates the development here of new *maxF*S algorithm variants of Chinneck's method, and their application to both underdetermined linear equations, and general linear systems that include inequalities and bounded variables . These new methods have advantages over existing methods for compressed sensing, and, as far as we are aware, provide the first results for sparse solutions of general linear systems.

1.1 Compressed sensing

The first step in compressed sensing is to generate a *measurement vector* of size m from a sparse input vector of size n, where $m << n$. The small measurement vector can be stored or transmitted in place of the much larger original vector. The trick is to be able to recover the original sparse input vector from the much smaller measurement vector: this is where sparse solutions for sets of underdetermined linear equations come into play.

The small $m \times 1$ measurement vector $\boldsymbol{b}$ is generated from the large sparse $n \times 1$ input vector x by multiplying it by the $m \times n$ *encoding matrix* $\boldsymbol{A}$, that is $\boldsymbol{Ax} = \boldsymbol{b}$. There are some restrictions on $\boldsymbol{A}$, chiefly the Restricted Isometry Property (Candes 2008), but this is satisfied by random matrices or random Gaussian matrices for example. We wish to recover the original large sparse input vector at some later time or at the signal receiver. Assuming that we know $\boldsymbol{b}$ and the encoding matrix, the unknown original sparse vector $\boldsymbol{x}$ is recovered by finding a sparse solution to the underdetermined linear system of equations $\boldsymbol{Ax} = \boldsymbol{b}$ where all variables are unbounded in both directions.

An intuitive understanding of compressed sensing is given by analogy to identifying counterfeit coins with a small number of weighings of different subsets of the coins: see Bryan and Leise (2013).

An $m \times n$ underdetermined linear system of equations $\boldsymbol{Ax} = \boldsymbol{b}$, where $m << n$ and $\boldsymbol{A}$ is random, has the property that any random subset of m columns will admit a solution, with probability very close to 1.0. Thus there always exists a solution that is at most m-sparse. For most compressed sensing signal recovery algorithms to correctly return the original input vector, the input vector must have a k-sparsity in which $k << m$. This affects m: the smaller the maximum ratio k/m for effective recovery, the larger m must be, and hence the less the compression. Thus a goal of compressed sensing algorithms is be effective at as high a ratio of k/m as possible in order to maximize the compression.

In the experiments by Jokar and Pfetsch (2008) with exact recovery using random unit matrices of size 128×256 for encoding, the largest k/m ratios at around 0.5 are obtained by the methods of Mangasarian and Chinneck. One goal of any new methods is to obtain higher k/m ratios.

1.2 Basis pursuit and variants

Let us define $\| \boldsymbol{x} \|_0$ to mean the number of supports in a vector $\boldsymbol{x}$. The sparse solution goal is to find $min \| \boldsymbol{x} \|_0$ while satisfying the constraints. Because this is NP-hard, many methods instead try to minimize a different vector norm. One of the original and most-used methods in compressed sensing minimizes $\| \boldsymbol{x} \|_1$, i.e. the sum of the magnitudes of the variables; this is known as *Basis Pursuit* (Chen et al. 2001). Basis pursuit is easily implemented as a linear program (LP) by a variable substitution: each x_j is replaced by the difference of two nonnegative variables $u_j - v_j$, with the objective function $min \sum_j (u_j + v_j)$. The solution is obtained by solving a single LP.

In *Matching Pursuit* algorithms (Mallatt and Zhang 1993), supports are chosen one at a time. At any intermediate iteration h there is a current solution $\boldsymbol{x}^h$ in which only the supports chosen thus far are nonzero, and the residual error $\boldsymbol{r}^h = \boldsymbol{b} - \boldsymbol{Ax}^h$. The next support chosen is the one that maximizes $\left| \boldsymbol{A}_j^T \boldsymbol{r}^h \right|$ where $\boldsymbol{A}_j^T$ is the jth column of $\boldsymbol{A}$. After adding a support, the approximate solution $\boldsymbol{x}^{h+1}$ is found to minimize the residual error $\mathbf{r}^{h+1}$; this is found in various ways, allowing only the supports selected so far to take nonzero values. In *Orthogonal Matching Pursuit* (Pati et al. 1993) $\boldsymbol{x}^h$ is found by minimizing the residual error in least squares sense. There are numerous matching pursuit variants.

1.3 Chinneck's original maxFS heuristic

The new algorithm variants developed here are based on Chinneck's original *maxFS* algorithm for infeasible general sets of linear constraints (Chinneck 1996, 2001). This algorithm elasticizes the input constraints by adding nonnegative elastic variables as follows:

nonelastic constraint	*elastic version*
$\boldsymbol{a}_i\boldsymbol{x} \geq b_i$	$\boldsymbol{a}_i\boldsymbol{x} + e_i \geq b_i$
$\boldsymbol{a}_i\boldsymbol{x} \leq b_i$	$\boldsymbol{a}_i\boldsymbol{x} - e_i \leq b_i$
$\boldsymbol{a}_i\boldsymbol{x} = b_i$	$\boldsymbol{a}_i\boldsymbol{x} + e_i' - e_i'' = b_i$

This allows constraints to be "stretched" as needed to provide a solution. An *elastic objective function, min* $Z = \sum_i (e_i + e_i' + e_i'')$, works to minimize the sum of the constraint violations. The original model is feasible if the minimum elastic Z is zero.

The algorithm solves the dual of the *maxFS* problem: it finds the smallest set of constraints which, when removed from the infeasible set, renders the remainder of the set feasible. The small removed set is called the *cover set*; its complement is the *maxFS* solution.

Chinneck's original algorithm is shown in Algorithm 1. *NINF* is the number of constraints that are violated (by having an associated elastic variable that is nonzero) after an elastic LP is solved. The algorithm tests the effect on elastic Z of temporarily removing each member of a set of candidate constraints, one at a time: the candidate constraint whose temporary removal most reduces elastic Z is removed permanently. The algorithm terminates when enough constraints are permanently removed that elastic Z becomes zero; the permanently removed constraints constitute the cover set, and the *maxFS* solution is the complement of this.

The list of candidate variables can be constructed several ways. The longest list, shown in Algorithm 1, consists of all constraints having a nonzero dual price at the LP solution in which the last permanently removed constraint was dropped. This list can be shortened by sorting it in decreasing order by the absolute value of the dual price associated with each constraint, and then selecting only the top k of these.

It is also possible to shorten the list of candidates by dividing the complete list of candidates into two sub-lists: those having a nonzero elastic variable ("violated"), and those whose elastic variables are zero, but who have a nonzero dual price ("satisfied"). The violated set is sorted in decreasing order of the product of the nonzero elastic variable and the absolute dual price (an estimate of the expected reduction in elastic Z were this constraint to be removed), and the satisfied set is sorted in decreasing order by the absolute dual price. Now take the top k elements from each list. Values of k in the range of 1–7 provide significant speed-ups with relatively little loss of accuracy.

Algorithm 1 Chinneck (1996) heuristic for *maxFS*

```
Input: Constraints defining an infeasible set of linear constraints.
0. $CoverSet \leftarrow \emptyset$. Set up elastic LP.
1. Solve elastic LP.
   if $NINF = 1$ then add the single violated constraint to $CoverSet$ and exit.
   $HoldSet \leftarrow$ {constraints having nonzero dual prices}.
2. $MinZ \leftarrow \infty$.
   $CandidateSet \leftarrow HoldSet$.
   for each constraint in $CandidateSet$:
       Delete the constraint.
       Solve elastic LP.
       if $elastic\ Z = 0$ then
           Add constraint to $CoverSet$ and exit.
       if $elastic\ Z < MinZ$ then
           $Winner \leftarrow$ currently deleted constraint.
           $MinZ \leftarrow elastic\ Z$.
           $HoldSet \leftarrow$ {all constraints having nonzero dual prices}.
           if $NINF = 1$ then $NextWinner \leftarrow$ single violated constraint.
           else $NextWinner \leftarrow \emptyset$.
       Reinstate the constraint.
3. Add $Winner$ to $CoverSet$.
   if $NextWinner \neq \emptyset$ then add $NextWinner$ to $CoverSet$ and exit.
   Delete the $Winner$ constraint permanently.
   Go to Step 2.
Output: The maxFS solution is the complement of $CoverSet$.
```

2 MaxFS-based algorithm options

There are numerous ways to formulate the linearly-constrained sparse solution problem for an LP-based *maxFS* solution derived from Chinneck's methods. The original formulation (Chinneck 2008) adds explicit non-elastic variable zeroing constraints of the form $x_j = 0$, one for every variable, to the system $\boldsymbol{Ax}=\boldsymbol{b}$. The solution is obtained via Algorithm 1 with the exception that candidate constraints can only be taken from among the variable zeroing constraints. The *maxFS* solution thus preserves the largest number of variable zeroing constraints. The drawback of this formulation is the blow-up in model size. Where the original model has m constraints and n variables, the revised model has $m + n$ constraints.

Jokar and Pfetsch (2008) proposed a different formulation for solution by a variant of Algorithm 1, specifically for sparse solutions of underdetermined systems of linear equations. The model uses the Basis Pursuit change of variables $x_j = u_j - v_j$ and objective function (see Section 1.2). The nonnegative u_j and v_j variables operate as elastic variables in a virtual variable zeroing constraint, and the Basis Pursuit objective function operates as the elastic objective function. The virtual variable zeroing constraint on variable x_j is removed by setting the objective function values of u_j and v_j to zero.

The set of candidates consists of those variables x_j having nonzero values at the LP solution. The winning candidate is the one whose virtual removal most reduces Z; the objective coefficients of the associated u_j and v_j variables are then permanently zeroed. A shorter list of candidates is obtained by sorting the variables in decreasing order by $u_j + v_j$ and taking only the top k from this list. This implementation provides excellent results in the experiments conducted by Jokar and Pfetsch. They also show that using a short list of size $k = 1$ provides a very large speed-up with little loss in solution quality.

The Jokar and Pfetsch implementation is efficient: where the original model has m constraints in n variables, their formulation has m constraints in $2n$ variables, which has little impact on solution time since the speed of the simplex method for LPs is mainly proportional to the number of constraints. The algorithm sometimes adds unneeded variables to the support set, so Jokar and Pfetsch employ a post-processing step. During post-processing, each support is temporarily forced to zero in turn; if the reduced set of supports still yields a feasible solution, then the support is permanently zeroed.

These two examples hint at the range of possible formulations for solution via variants of Algorithm 1. The formulation elements that can be adjusted are described below.

Constraints can be elasticized by the addition of nonnegative elastic variables, or left in their original state.

Variable zeroing can be accomplished in multiple ways:

1. Implicitly using the Basis Pursuit substitution of $u_j - v_j$ in place of x_j, where u_j and v_j are nonnegative, and the associated Basis Pursuit objective function. Variables can be encouraged to small (if not zero) values by assigning large objective coefficients to their associated u_j, v_j pair, or can be encouraged to take nonzero values by assigning smaller or zero values to the objective function coefficients of the associated u_j, v_j pair.
2. Implicitly by adding elastic zeroing constraints of the form $x_j + e_j^+ - e_j^- = 0$ where e_j^+ and e_j^- are nonnegative, and using the standard elastic Z minimization. As in the previous point, the relative size of the elastic variable objective coefficient can encourage the variable value towards or away from zero.
3. Explicitly by adding nonelastic variable zeroing constraints of the form $x_j = 0$.
4. Explicitly by setting the variable lower and upper bounds to zero.

In the first two options above, appropriate adjustments can be made if there are upper or lower bounds on the variables.

The objective function can be set up in multiple ways:

1. Where the $x_j = u_j - v_j$ variable substitution has been used, the objective is to minimize the sum of the nonnegative u_j and v_j variables.
2. If the constraints are elasticized, or if there are elastic variable zeroing constraints, or both, then the objective is minimize the sum of the elastic variables.
3. If the $x_j = u_j - v_j$ variable substitution has been used, and the constraints have been elasticized, then the objective is to minimize the sum of the nonnegative u_j and v_j variables and the nonnegative elastic variables.
4. The objective function coefficients can be dynamically adjusted as the solution proceeds.

The solution style can be set as additive or subtractive. Algorithm 1 is additive: constraints are added to the cover set as the algorithm proceeds. However it is also possible to start with all constraints in the cover set, and to remove constraints from it as the algorithm proceeds. For example, in the subtractive style the algorithm can remove a constraint from the cover set if doing so does not make the complementary set of constraints infeasible. The post-processing algorithm described later is subtractive.

Using the different possibilities for each formulation element listed above, a wide variety of LP-based *maxFS* algorithm variants can be generated. We explore a few of the options in this paper.

3 New maxFS algorithms for underdetermined systems of linear equations

Four new *maxFS*-based algorithms were created by combining the formulation elements in different ways. The initial model has m constraints and n variables.

Method A. The constraints are elasticized and variables are zeroed by setting their upper and lower bounds to zero. All variables are initially zeroed. The candidate list consists of variables having a bound with nonzero reduced cost, sorted in decreasing order by absolute value. Supports are added by removing the upper and lower bounds on the variables. This gives a model having m constraints in $n + 2m$ variables. Note that this method is the only one that evaluates intermediate solutions based on the constraint violations (i.e. the sum of the constraint elastic variables).

Method B. This is a variant of the Jokar and Pfetsch implementation. Constraints are not elasticized, and the variable substitution $x_j = u_j - v_j$ is used. This gives a model in m constraints and $2n$ variables, when all n original variables are unbounded in both directions, or fewer variables when some of the original variables are nonnegative or nonpositive. Candidates are variables that have nonzero values and are not already in the list of supports. A reduced set of candidates is chosen by taking the top k largest magnitude candidates. The extensions to the Jokar and Pfetsch implementation are as follows:

1. When a candidate is moved into the support set, the objective function coefficients of its corresponding u_j and v_j variables are reduced to 0.1 instead of 0. This has the advantage of encouraging supports that were incorrectly added to the support set to take the value zero. The values of the variables at the final solution are checked and only those that are actually nonzero constitute the output set of supports.
2. Because the constraints are not elasticized, they are satisfied at all times. Thus every intermediate LP solution generated by candidate testing is checked for the actual number of supports. The solution having the fewest supports is remembered, and if it has fewer members than the support set generated by the main algorithm, then it is returned instead.

Method C. Constraints are not elasticized. Variables are zeroed by explicit elastic constraints, so the objective function seeks to minimize the sum of the elastic variables. This gives a model of size $m + n$ constraints in $3n$ variables when all n original variables are unbounded. There are fewer constraints and elastic variables if some original variables can never be zero because their bounds prevent this (e.g. $x \geq 1$), and fewer elastic variables if some of the original variables are nonnegative or nonpositive. There are two sets of candidates: (i) original variables that have nonzero values and are not already in the list of supports, and (ii) original variables having the value zero and a nonzero dual price in the associated variable zeroing constraint. A reduced set of candidates can be chosen by taking the top k largest magnitude candidates from each list (thus $2k$ candidates in total). Candidates are moved into the support set by releasing the elastic variable zeroing constraints by setting the objective function coefficients of their elastic variables to zero. Intermediate solutions are checked.

Method M1. The Basis Pursuit (BP) algorithm has some attractive properties. If the input vector has k-sparsity much smaller than m, then it returns the correct solution after solving a single LP. If it returns an incorrect solution, it normally has a small relative squared error (defined in Section 4.1) for compressed sensing. However method B is able to provide correct solutions up to a much larger value of k/m. Thus we combine the two algorithms in method M1. When the equations must be satisfied exactly, BP is run first, and if the solution has fewer than $m - 3$ supports, then it is returned immediately, otherwise method B is also run and the smaller of the two solutions is returned. If some error is permitted in the equations, then both methods are always run and the smaller of the two solutions is returned. If the sparsities of the two solutions match, then the BP solution is returned since it tends to have a smaller relative squared error compared to the input vector.

The LP-based solutions also make it easy to adjust the solution to allow for errors, or to limit the number of supports, as described next.

Absolute or relative error tolerances. Controlled error in the sets of linear equations is permitted by adjusting the $\boldsymbol{b}$ matrix. LP solvers typically treat $\boldsymbol{b}$ as two vectors: lower bounds $\boldsymbol{b}^l$ and upper bounds $\boldsymbol{b}^u$. For an equality constraint i, $b_i^l = b_i^u$. To permit a small absolute error Δ^{abs} in equation i the bounds are adjusted to $b_i^l = b_i^{orig} - \Delta^{abs}$ and $b_i^u = b_i^{orig} + \Delta^{abs}$. To permit a small relative error Δ^{rel} in equation i the bounds are adjusted to $b_i^l = b_i^{orig} - \left|b_i^{orig}\right| \Delta^{rel}$ and $b_i^u = b_i^{orig} + \left|b_i^{orig}\right| \Delta^{rel}$. It is also possible to cap the relative error to be less than a given Δ^{abs} when b_i^l and b_i^u are initialized. Error tolerances are used in the compressed sensing application to remove noise and allow greater signal compression (with some signal quality loss) (Chen et al. 2001). The same idea can be applied for general linear systems if fewer supports are desired.

Limited number of supports. It is possible to restrict the allowable number of supports to be less than some value *sl*. Since the new methods described above return the supports one by one, it is possible to simply stop adding supports when *sl* supports have been returned. It is also possible to let any method find a set of supports and then select a subset of size at most *sl*. Where there are more than *sl* supports, one way to make the selection is to choose the *sl* largest absolute value supports.

Post-processing. Some algorithms generate a list of supports that can be shortened by a post-processing step, as originally described by Jokar and Pfetsch (2008). In post-processing, each support is temporarily forced to zero in turn; if the reduced set of supports still yields a feasible solution, then the support is permanently zeroed. Note that the order of the variables has an impact on the outcome, but there is no way to know the best order in advance: we simply use the order of the variables as listed in the support set returned by the initial algorithm. Post-processing is quite successful for some algorithms, but requires the solution of k LPs, where the initial solution is k-sparse.

Obtaining the output vector. The new algorithms are intended to return a list of the supports, but not their values. However the actual output vector is needed for comparison to the input vector. This is obtained by solving a final LP constructed as follows: remove all non-support variables from the $\boldsymbol{A}$ matrix, make the change of variables $x_j = u_j - v_j$, elasticize all of the constraints (by adding $e_i^+ - e_i^-$ to each constraint, where these variables are nonnegative) and add the elastic objective function $min \sum_j (u_j + v_j) + 100 \sum_i \left(e_i^+ + e_i^-\right)$. This finds a solution that has the smallest possible values of x_j; the large penalty on the constraint elastic variables prevents minor error tolerance issues.

4 Experiments: Undetermined random linear systems

In a common compressed sensing scenario, the problem is finding a sparse solution for the underdetermined systems of linear equations $\boldsymbol{Ax}=\boldsymbol{b}$, where $\boldsymbol{A}$ and $\boldsymbol{b}$ are given, $\boldsymbol{A}$ is random, and the variables are not bounded. We experiment with methods for finding exact solutions, solutions with a limited amount of equation error, and solutions having a limited number of supports.

4.1 Experimental setup

Experiments were conducted under these conditions:

Hardware. Intel 64-bit Xeon 6-core E5-1650 v3 CPU, 3.50 GHz, with 32 Gbytes of RAM. Windows 10 operating system.

Software. Algorithms are programmed in Matlab R2017b. The LP solver is Mosek (2018), called via the Mosek Fusion API for Matlab, release 8.1.0.34. The feasibility tolerance for the LP solutions is the MOSEK default 1×10^{-6}. The same tolerance is applied to identify supports.

Test data. Random encoding matrices $\boldsymbol{A}$ of size 128×256 are generated using a uniform distribution between 0 and 1 (Matlab rand function). Each column of the matrix was then normalized to a length of 1.

Known input vectors of k-sparsity are generated from $\boldsymbol{A}$ by randomly selecting k columns and assigning each selected column a weight chosen randomly from a normal distribution with mean 0 and standard deviation 1. Ten trials are conducted at each selected value of input sparsity.

Methods compared. Basis Pursuit (BP), and the maxFS-based formulations JP (the Jokar and Pfetsch formulation), A, B, C and M1.

Evaluation metrics:

- *Number of successes.* A count of the number of instances in which the recovery algorithm succeeds in returning the known input vector, over 10 trials. This is used when the error delta is zero, i.e. an exact solution is required (within accepted tolerances).
- *Sparsity of solution.* When a nonzero error is allowed, every method returns a solution having fewer nonzeros than the input vector. Thus it is more meaningful to examine the sparsity of the solution vector in comparison to the sparsity of the input vector.
- *Solution time.* The total time needed for the sparse solution algorithm to complete. For some algorithms this includes post-processing time (not all algorithms require this). Times are averaged over 10 trials, on a lightly loaded machine.
- *Relative squared error.* This evaluates the differences between the input vector $\boldsymbol{x}^{in}$ and $\boldsymbol{x}^{sol}$, the vector output by the sparse recovery algorithm: $\frac{\sum_j \left(x_j^{sol} - x_j^{in}\right)^2}{\sum_j \left(x_j^{in}\right)^2}$.

4.2 Exact solutions

These experiments examine the case where the allowable error delta on the equations is 0.0 (aside from the default LP solver feasibility tolerance of 1×10^{-6}). There are 10 tests at every second input sparsity from 62 to 98. Input sparsities begin at 62 since previous work indicates that existing algorithms such as basis pursuit fail well before $k/m = 0.5$ (in this case at $k = 64$). Sparsities beyond 86 are not shown because there are zero successes. Ten different random $\boldsymbol{A}$ matrices are used at each input sparsity at each list length; the same random $\boldsymbol{A}$ matrix is used for each method, but the $\boldsymbol{A}$ matrices differ between list lengths. Three list lengths are tested: 1, 7, and unlimited. The number of successful recoveries of the input vector are summarized in Table 1.

Table 1: Number of successes of 10 trials for exact solutions.

	List Length 1						**List Length 7**						**List Length Unlimited**					
k	**BP**	**JP+**	**A+**	**B+**	**C+**	**M1+**	**BP**	**JP+**	**A+**	**B+**	**C+**	**M1+**	**BP**	**JP+**	**A+**	**B+**	**C+**	**M1+**
62	1	10	0	10	10	10	0	9	0	9	9	9	0	10	1	10	10	10
64	0	10	0	10	10	10	0	10	0	10	10	10	0	9	0	9	9	9
66	0	9	0	9	9	9	0	10	0	10	10	10	0	$\underline{10}$	0	$\underline{10}$	$\underline{10}$	$\underline{10}$
68	0	$\underline{9}$	0	9	$\underline{9}$	9	0	$\underline{9}$	0	$\underline{10}$	$\underline{9}$	$\underline{10}$	0	6	0	8	6	8
70	0	8	0	$\underline{10}$	8	$\underline{10}$	0	7	0	8	7	8	0	5	0	8	5	8
72	0	6	0	6	6	6	0	6	0	8	6	8	0	4	0	7	4	7
74	0	3	0	3	3	3	0	4	0	5	4	5	0	4	0	5	4	5
76	0	6	0	8	6	8	0	3	0	5	3	5	0	3	0	5	3	5
78	0	4	0	4	4	4	0	3	0	4	3	4	0	3	0	4	3	4
80	0	2	0	1	2	1	0	1	0	1	1	1	0	1	0	3	1	3
82	0	0	0	0	0	0	0	1	0	1	1	1	0	0	0	0	0	0
84	0	2	0	2	2	2	0	0	0	1	0	1	0	0	0	1	0	1
86	0	0	0	0	0	0	0	2	0	2	2	2	0	0	0	0	0	0

Post-processing is indicated by a "+" added to the method name. Its effectiveness varies by method. Over the 570 trials for each method, the post-processing success rates are BP 0%, JP 18.8%, A 1.2%, B 57.7%, C 36.8%, M1 2.8%. Post-processing was applied for all methods except BP.

For the 128×256 $\boldsymbol{A}$ matrices in this experiment, the tested sparsities range between $0.48m$ ($k = 62$) and $0.77m$ ($k = 98$). As shown by Jokar and Pfetsch (2008), BP cannot reliably return the correct solution when the sparsity exceeds $0.31m$. Method A+ is completely ineffective at the range of sparsities tested, whereas the other new methods are much more effective. Method A+ is the only new method that uses an equation-oriented evaluation of the intermediate solutions: the sum of the constraint elastic variables. All of the other new methods use a variable-oriented evaluation of the intermediate solutions based on the size of the supports, which is more directly related to the sparsity goal.

The underlines in Table 1 indicate the largest input sparsity row above which all trials have 9 or more successes out of 10. This is the largest input k-sparsity for which the method and list length can be considered reliably successful. The highest sparsity that can be reliably recovered by any method is 70 ($0.55m$), for methods B+ and M1+, at list length 1. It is surprising that this happens at list length of 1; previous work in analyzing infeasible sets of linear constraints indicated that longer list lengths tend to be better. This result may be an artifact of this particular experiment: recall that any m variables will yield a feasible solution; thus it may be that choosing the single most promising variable at each iteration, as in list length 1, is the most reliable way to the shortest sequence of selected variables.

When a method fails to succeed it almost always return a solution of size m. In some cases the newer methods are able to find solutions of size $m-1$ or $m-2$. Note also that methods JP+ and C+ have identical success rates at all sparsities and over all 3 list lengths. This indicates that candidate magnitude is a better metric for variable selection than variable bound sensitivity.

Average solution times are summarized in Table 2. As expected, solution times for the new methods grow with the list length. Times also increase with the k-sparsity of the input vector, typically with a peak in the middle of the range of input sparsities tested. BP consists of a single LP solution, so solution times are stable across the range of input sparsities. The solution time for method C+ is roughly double that of method B+ or method C+ because it uses a candidate list that is twice as long.

Table 2: Average solution times (seconds) for exact solutions.

	List length 1						List Length 7						List Length Unlimited					
k	**BP**	**JP+**	**A+**	**B+**	**C+**	**M1+**	**BP**	**JP+**	**A+**	**B+**	**C+**	**M1+**	**BP**	**JP+**	**A+**	**B+**	**C+**	**M1+**
62	0.1	3.8	2.2	3.9	6.4	3.6	0.1	21.9	14.3	23.2	42.1	23.2	0.1	202.9	340.7	206.5	613.7	206.3
64	0.1	4.1	2.1	4.2	6.9	4.3	0.1	20.3	14.0	21.2	38.9	21.3	0.1	223.5	351.8	228.4	671.6	228.1
66	0.1	4.6	2.1	4.8	7.7	4.8	0.1	21.4	14.2	22.5	41.5	22.4	0.1	228.1	358.7	232.7	670.7	232.5
68	0.1	4.7	2.2	4.9	7.8	5.0	0.1	24.0	13.7	23.3	46.4	23.4	0.1	303.0	353.2	281.2	833.6	281.1
70	0.1	5.2	2.1	4.6	8.7	4.7	0.1	28.1	14.1	28.1	53.5	28.1	0.1	309.8	354.1	283.7	859.6	283.8
72	0.1	5.6	2.1	5.8	9.6	5.9	0.1	29.8	13.6	28.6	57.7	28.6	0.1	333.0	356.7	304.2	908.8	306.2
74	0.1	7.0	2.1	7.5	11.5	7.6	0.1	33.5	14.0	34.6	64.5	34.8	0.1	341.7	359.2	341.0	922.0	340.6
76	0.1	6.1	2.1	5.8	10.1	5.9	0.1	35.0	14.0	35.0	66.8	35.3	0.1	350.6	352.9	347.9	944.5	347.9
78	0.1	6.7	2.1	7.2	11.1	7.4	0.1	35.5	14.1	37.0	68.1	37.2	0.1	350.4	354.2	357.2	961.1	357.2
80	0.1	7.6	2.2	8.7	12.6	8.8	0.1	38.0	13.9	42.2	72.4	42.3	0.1	377.4	354.1	382.7	1011.9	383.2
82	0.1	9.2	2.1	8.8	15.1	8.8	0.1	38.2	14.2	42.8	73.0	42.8	0.1	381.0	357.8	404.4	1032.3	405.2
84	0.1	11.3	3.0	11.4	12.9	10.4	0.1	39.3	14.2	42.5	75.0	42.5	0.1	382.0	354.3	398.3	1045.3	398.6
86	0.1	8.1	2.2	9.0	13.4	9.0	0.1	37.2	14.1	41.1	71.5	41.2	0.1	386.7	349.7	409.0	1052.7	409.3
88	0.1	8.1	2.2	8.9	13.6	9.1	0.1	39.5	14.2	44.0	75.4	44.2	0.1	381.0	352.3	403.4	1030.7	404.9
90	0.1	7.8	2.1	8.6	12.9	8.7	0.1	39.5	14.3	44.4	75.4	44.4	0.1	379.9	356.7	402.0	1030.4	401.5
92	0.1	7.8	2.1	8.5	12.9	8.7	0.1	39.2	13.9	43.7	75.5	43.8	0.1	383.4	359.3	406.2	1041.5	407.7
94	0.1	7.8	2.1	8.6	13.1	8.8	0.1	39.5	14.3	44.3	75.3	44.5	0.1	386.1	356.8	410.0	1045.7	412.3
96	0.1	7.8	2.1	8.6	13.0	8.7	0.1	39.4	14.0	43.9	75.4	44.0	0.1	381.6	349.9	404.6	1030.6	405.1
98	0.1	7.9	2.1	8.6	13.0	8.8	0.1	39.2	14.0	43.9	75.0	44.0	0.1	385.4	359.0	411.3	1050.0	411.4

List length 1 is not only faster, but has generally higher solution accuracies, and hence is preferred in this application. Even at this list length, solution times are much longer than for BP, making these methods useful for archival purposes rather than in real-time applications.

The average relative squared errors are shown in Table 3. The best values at each input sparsity at each list length are shown in boldface. Table 3 shows that method M1+ generally gives the lowest relative squared error at any given list length and input sparsity. The results between list lengths are not directly comparable because different $\boldsymbol{A}$ matrices are used for each list length. M1+ provides lower errors than BP at the lower input sparsities tested because it finds the correct solution more often. At higher input sparsities, both methods fail, producing solutions of size m and in this case M1+ outputs the BP solution. Thus for the higher input sparsities methods BP and M1+ have identical errors.

Table 3: Average relative squared errors for exact solutions.

	List Length 1						List Length 7						List Length Unlimited					
k	**BP**	**JP+**	**A+**	**B+**	**C+**	**M1+**	**BP**	**JP+**	**A+**	**B+**	**C+**	**M1+**	**BP**	**JP+**	**A+**	**B+**	**C+**	**M1+**
62	0.085	**0**	0.673	**0**	**0**	**0**	0.096	0.017	0.463	0.014	0.017	0.010	0.095	**0**	0.503	**0**	**0**	**0**
64	0.105	**0**	0.673	**0**	**0**	**0**	0.088	**0**	0.676	**0**	**0**	**0**	0.118	0.018	0.577	**0.016**	0.019	0.021
66	0.079	**0.008**	0.610	**0.008**	**0.008**	0.016	0.075	**0**	0.429	**0**	**0**	**0**	0.130	**0**	0.608	**0**	**0**	**0**
68	0.106	0.051	0.564	0.057	0.051	**0.014**	0.095	0.056	0.421	**0**	0.056	**0**	0.179	0.186	0.734	0.147	0.186	**0.056**
70	0.122	0.109	0.777	**0**	0.109	**0**	0.204	0.168	0.658	0.134	0.167	**0.056**	0.142	0.187	0.721	0.115	0.188	**0.055**
72	0.184	0.152	0.969	0.330	0.152	**0.099**	0.134	0.119	0.651	0.067	0.119	**0.037**	0.159	0.210	0.820	0.212	0.210	**0.051**
74	0.194	0.337	0.857	0.424	0.337	**0.157**	0.187	0.175	0.720	0.180	0.175	**0.109**	0.190	0.253	0.849	0.312	0.253	**0.123**
76	0.173	0.080	0.779	0.158	0.080	**0.042**	0.200	0.357	0.833	0.324	0.357	**0.121**	0.164	0.265	0.780	0.186	0.265	**0.092**
78	0.215	0.233	0.871	0.257	0.233	**0.156**	0.192	0.305	0.648	0.330	0.305	**0.110**	0.212	0.289	0.743	0.382	0.289	**0.158**
80	0.177	0.274	0.950	0.586	0.274	**0.165**	0.232	0.397	0.757	0.339	0.397	**0.222**	0.226	0.307	0.807	0.336	0.307	**0.176**
82	**0.247**	0.315	0.884	0.332	0.315	**0.247**	0.240	0.388	1.002	0.765	0.388	**0.226**	**0.256**	0.337	1.012	0.400	0.337	**0.256**
84	0.274	0.345	1.120	0.438	0.345	**0.243**	0.315	0.557	0.850	0.946	0.557	**0.301**	0.209	0.357	1.008	0.477	0.357	**0.197**
86	**0.290**	0.483	1.091	0.603	0.483	**0.290**	0.260	0.444	0.786	0.372	0.444	**0.232**	**0.274**	0.548	1.143	0.487	0.548	**0.274**
88	**0.300**	0.531	1.066	0.522	0.531	**0.300**	**0.267**	0.523	1.188	0.696	0.523	**0.267**	**0.305**	0.664	0.890	0.545	0.664	**0.305**
90	**0.261**	0.630	1.316	0.765	0.630	**0.261**	**0.311**	0.746	1.018	1.030	0.746	**0.311**	**0.338**	0.773	1.163	0.594	0.773	**0.338**
92	**0.285**	0.635	1.268	0.735	0.635	**0.285**	**0.265**	0.498	1.032	0.510	0.498	**0.265**	**0.267**	0.561	0.911	0.706	0.561	**0.267**
94	**0.294**	0.609	1.193	0.911	0.609	**0.294**	**0.363**	0.823	1.218	1.222	0.823	**0.363**	**0.271**	0.655	0.829	0.839	0.655	**0.271**
96	**0.350**	0.825	1.058	0.804	0.825	**0.350**	**0.334**	0.794	1.057	0.757	0.794	**0.334**	**0.411**	0.782	1.337	1.553	0.782	**0.411**
98	**0.343**	0.744	1.230	0.749	0.744	**0.343**	**0.378**	0.788	1.300	0.891	0.788	**0.378**	**0.342**	0.715	1.107	0.993	0.716	**0.342**
avg	*0.215*	*0.335*	*0.945*	*0.404*	*0.335*	***0.172***	*0.223*	*0.377*	*0.827*	*0.451*	*0.377*	***0.176***	*0.226*	*0.374*	*0.871*	*0.437*	*0.374*	***0.179***

Method A+, with its constraint-oriented evaluations, gives the largest relative squared errors. Methods JP+ and C+ give identical relative squared errors over all input sparsities and list lengths. Since they also give identical solution rates, method JP+ dominates method C+ because it is much faster. Method M1+ at list length 1 is preferred for this compressed sensing scenario due to the combination of accuracy, speed, and low relative squared error.

A significant result is that methods B+ and M1+ at list length 1 are reliable up to $k/m = 0.55$ ($k = 70$ where $m = 128$). This is a much higher k/m ratio than BP or similar existing methods, meaning that the transmitted or stored $\boldsymbol{b}$ vector can be much smaller. Jokar and Pfetsch (2008) found BP to be reliable up to about $k/m = 0.31$, thus for accurate recovery of the sparse signal $\boldsymbol{b}$ must have length $k/0.31 = 3.2k$. For methods B+ and M1+ at list length 1, $\boldsymbol{b}$ must have a length of just $k/0.55 = 1.83k$.

4.3 Approximate solutions

The LP formulation makes it easy to allow small constraint equation errors by adjusting the equation tolerances. This can be beneficial for two reasons: (i) denoising (Chen et al. 2001), and (ii) finding solutions that are sparser than the input signal. Experiments with an allowable constraint error deviation of ± 0.1 in each b_i (a value suggested by Chen et al. 2001)) were conducted under the conditions described in Section 4.1. Since the input vector is not recovered, the important metrics are the sparsity and relative squared error of the solution.

The solution sparsity results are summarized in Table 4. Post-processing is very effective for all methods in this experiment, including Basis Pursuit. The smallest average output sparsities for each list length in each row are shown in boldface.

The constraint error tolerance allows all methods to return solutions having fewer supports than the input vector. BP+ and row-oriented Method A+ return solutions having many more supports than the other methods. There is little difference between the average solution sparsities of the better methods (JP+, B+, C+, M1+), though JP+ and C+ are marginally better on average. These methods return solutions having 40–50% fewer supports than the input vector at this level of constraint error.

The average relative squared errors are shown in Table 5. The lowest errors on average are given by methods B+ and M1+ at all list lengths. However there is a noticeable pattern in the results: B+ and M1+ dominate up to about $0.64m$ ($k = 80$), and BP+ dominates thereafter. The slightly fewer supports returned by JP+ and C+ come at the cost of higher relative squared errors. As expected the relative squared errors for constraint error delta 0.1 are significantly larger than for the exact solutions. The errors are expected to be smaller for smaller constraint error deltas.

Table 4: Average supports in solution for constraint error delta 0.1.

	List Length 1						**List Length 7**						**List Length Unlimited**					
k	**BP+**	**JP+**	**A+**	**B+**	**C+**	**M1+**	**BP+**	**JP+**	**A+**	**B+**	**C+**	**M1+**	**BP+**	**JP+**	**A+**	**B+**	**C+**	**M1+**
62	57.6	37	42.3	37	**36.9**	37	57.3	**35.4**	41	36.3	**35.4**	36.3	55.2	34.8	38.5	34.8	**34.6**	34.8
64	59.6	39.1	44.8	**38.4**	39	**38.4**	61.1	37.3	42.2	37.9	**37.2**	37.9	62.8	39	43.7	40	**38.9**	40
66	59.5	37.7	45.3	**37**	37.9	**37**	59.2	**36.4**	40.7	**36.4**	**36.4**	**36.4**	59.1	36.9	40	**36.4**	36.6	**36.4**
68	60	39.1	45.6	39.3	**38.8**	39.3	62.3	38.5	45.1	39.2	**38.4**	39.2	61.1	**37.6**	42.1	38.3	37.7	38.3
70	66.6	40.2	49.4	**40**	**40**	**40**	65.3	**39.4**	46	40	**39.4**	40	64.1	41.2	45.4	42	**41.1**	42
72	65.5	**43**	51.1	43.5	43.1	43.5	65.3	**40.8**	45.6	41.1	**40.8**	41.1	64.5	41.1	47	**41**	**41**	**41**
74	67.1	**42.2**	48.6	43.2	**42.2**	43.2	67.1	42.4	50.1	43.2	**42.1**	43.2	64.4	41.5	45.8	42.6	**41.4**	42.6
76	63.7	**42.3**	48.3	42.4	**42.3**	42.4	65.5	41.6	46.1	**41.2**	41.3	**41.2**	66.1	**42.9**	47.1	44.2	**42.9**	44.2
78	68.2	43.8	50.1	**42.6**	43.5	**42.6**	64.4	43	47.5	**42.6**	43.1	**42.6**	67.7	**42.2**	48.3	43.4	42.5	43.4
80	70.2	**46.2**	52.5	46.8	**46.2**	46.8	68.9	43.6	49.1	43.5	**43.5**	43.5	68.6	**43.6**	49.7	44.2	**43.6**	44.2
82	66.2	**44.3**	51	44.4	**44.3**	44.4	67.3	**44.5**	51.3	44.7	**44.5**	44.7	69.6	45.2	51.9	45.1	**45**	45.1
84	70.4	44.7	52.2	**44.1**	44.5	**44.1**	70.1	44.4	52.1	44.9	**44.3**	44.9	72.6	**46.6**	51.6	46.8	**46.6**	46.8
86	72.5	**47.4**	55.5	48	**47.4**	48	72	**46.5**	52.6	47.8	**46.5**	47.8	71.8	46.1	50.9	47	**45.9**	47
88	73.1	46.2	52.9	46.9	**46.1**	46.9	69.6	**45.2**	51.9	45.6	**45.2**	45.6	72.6	47.8	53.1	47.8	**47.7**	47.8
90	72.9	**47.2**	54.2	47.6	**47.2**	47.6	73.5	**46.4**	53	46.7	**46.4**	46.7	74.2	47.3	52.7	48.2	**47.3**	48.2
92	76.5	**49.8**	57.9	51.1	**49.8**	51.1	74.8	**48.4**	55.1	49	48.5	49	74.4	47.4	55.2	48	**47.2**	48
94	76.1	**50.4**	57.1	50.4	50.5	**50.4**	75.9	48.6	54.2	48.7	**48.4**	48.7	74.8	48.4	53.3	48.9	**48.3**	48.9
96	75.7	**49.3**	57.1	50.5	**49.3**	50.5	75.1	49.1	56.1	**49**	49	**49**	75	48	54.6	**47.6**	47.8	**47.6**
98	79	50	58.5	**49.5**	49.9	**49.5**	78	**48.1**	58.1	49.4	48.2	49.4	76.2	**47.4**	53.2	47.8	47.6	47.8
avg	*68.4*	***44.2***	*51.3*	*44.4*	***44.2***	*44.4*	*68*	***43.1***	*49.4*	*43.5*	***43.1***	*43.5*	*68.1*	***43.4***	*48.6*	*43.9*	***43.4***	*43.9*

Table 5: Average relative squared error for constraint error delta 0.1.

	List Length 1						**List Length 7**						**List Length Unlimited**					
k	**BP+**	**JP+**	**A+**	**B+**	**C+**	**M1+**	**BP+**	**JP+**	**A+**	**B+**	**C+**	**M1+**	**BP+**	**JP+**	**A+**	**B+**	**C+**	**M1+**
62	0.287	0.260	0.367	**0.254**	0.259	**0.254**	0.279	0.278	0.453	**0.237**	0.281	**0.237**	0.294	0.201	0.325	**0.182**	0.191	**0.182**
64	0.310	0.305	0.513	**0.278**	0.305	**0.278**	0.298	0.260	0.347	**0.251**	0.256	**0.251**	0.421	0.354	0.508	**0.352**	0.354	**0.352**
66	0.288	0.268	0.372	**0.179**	0.270	**0.179**	0.275	0.184	0.320	**0.175**	0.185	**0.175**	0.281	0.193	0.297	**0.175**	0.179	**0.175**
68	0.329	0.240	0.374	**0.222**	0.232	**0.222**	0.352	**0.235**	0.412	0.258	0.235	0.258	0.346	0.338	0.440	**0.271**	0.336	**0.271**
70	0.351	0.237	0.580	**0.221**	0.235	**0.221**	0.389	0.397	0.597	**0.344**	0.395	**0.344**	0.381	0.329	0.598	**0.310**	0.330	**0.310**
72	0.389	0.384	0.591	**0.365**	0.388	**0.365**	0.392	0.334	0.527	0.370	**0.332**	0.370	0.332	0.253	0.428	**0.213**	0.250	**0.213**
74	0.445	0.470	0.594	**0.439**	0.469	**0.439**	**0.399**	0.448	0.584	0.436	0.449	0.436	0.423	0.383	0.496	**0.351**	0.382	**0.351**
76	0.377	0.329	0.540	**0.283**	0.329	**0.283**	0.369	0.307	0.426	**0.250**	0.315	**0.250**	0.464	0.436	0.468	**0.389**	0.436	**0.389**
78	**0.420**	0.522	0.624	0.447	0.517	0.447	0.435	0.454	0.626	**0.377**	0.447	**0.377**	0.416	0.357	0.495	**0.339**	0.358	**0.339**
80	**0.455**	0.594	0.625	0.494	0.595	0.494	0.466	0.487	0.662	**0.392**	0.486	**0.392**	0.431	0.428	0.557	**0.412**	0.433	**0.412**
82	**0.408**	0.509	0.597	0.451	0.509	0.451	0.456	0.420	0.624	**0.364**	0.409	**0.364**	0.459	0.503	0.740	**0.453**	0.504	**0.453**
84	0.477	0.585	0.781	**0.466**	0.577	**0.466**	**0.536**	0.657	0.818	0.577	0.664	0.577	**0.390**	0.530	0.653	0.415	0.524	0.415
86	0.471	0.489	0.735	0.436	0.491	**0.436**	0.522	0.583	0.627	**0.455**	0.579	**0.455**	0.514	0.581	0.707	**0.481**	0.584	**0.481**
88	**0.522**	0.653	0.694	0.534	0.656	0.534	**0.503**	0.591	0.746	0.523	0.591	0.523	**0.492**	0.592	0.742	0.538	0.590	0.538
90	**0.482**	0.574	0.702	0.529	0.571	0.529	**0.619**	0.665	0.726	0.620	0.665	0.620	**0.632**	0.790	0.897	0.688	0.793	0.688
92	**0.468**	0.654	0.768	0.551	0.652	0.551	0.546	0.623	0.739	**0.542**	0.620	**0.542**	**0.550**	0.758	0.843	0.648	0.760	0.648
94	**0.544**	0.672	0.728	0.609	0.667	0.609	**0.558**	0.675	0.822	0.596	0.679	0.596	**0.529**	0.725	0.683	0.665	0.725	0.665
96	**0.541**	0.820	0.879	0.714	0.823	0.714	**0.519**	0.773	0.906	0.679	0.775	0.679	**0.495**	0.597	0.900	0.558	0.604	0.558
98	0.588	0.813	0.951	**0.571**	0.809	**0.571**	**0.546**	0.815	1.055	0.682	0.806	0.682	**0.556**	0.717	0.790	0.636	0.719	0.636
avg	*0.429*	*0.494*	*0.632*	***0.423***	*0.492*	***0.423***	*0.445*	*0.483*	*0.632*	***0.428***	*0.483*	***0.428***	*0.442*	*0.477*	*0.609*	***0.425***	*0.477*	***0.425***

Solution times are summarized in Table 6, and are much smaller than for the exact solutions. At one extreme, method B+ takes about a quarter of the time at unlimited list length vs. the exact solution. On the other hand, since post-processing is used for method B, method B+ takes much longer. Times are likely to be shorter for larger allowable constraint error tolerances.

Considering the average output sparsities, relative squared errors, and solution times, method B+ at list length 1 is preferred in this scenario.

Table 6: Solution times (seconds) for constraint error delta 0.1.

	List Length 1						List Length 7						List Length Unlimited					
k	**BP+**	**JP+**	**A+**	**B+**	**C+**	**M1+**	**BP+**	**JP+**	**A+**	**B+**	**C+**	**M1+**	**BP+**	**JP+**	**A+**	**B+**	**C+**	**M1+**
62	3.3	2.2	1.8	2.2	4.0	5.4	3.2	9.6	3.8	9.1	21.7	12.3	3.2	57.2	75.4	57.0	352.3	60.2
64	3.5	2.5	2.0	2.3	4.4	5.8	3.5	10.4	4.0	10.0	23.3	13.5	3.6	72.8	91.6	73.6	417.3	77.3
66	3.4	2.5	2.0	2.2	4.3	5.5	3.4	9.9	3.8	9.4	22.7	12.6	3.3	61.5	82.2	61.5	377.0	64.9
68	3.8	2.5	2.0	2.4	4.5	6.1	3.7	11.0	4.7	10.5	24.8	14.3	3.5	67.6	89.6	67.2	397.0	70.7
70	4.2	2.6	2.4	2.5	4.7	6.6	3.8	11.4	4.9	10.7	25.3	14.5	3.8	80.8	96.9	81.7	442.0	85.6
72	4.0	2.7	2.4	2.6	5.0	6.7	3.9	11.9	4.8	11.2	26.7	15.0	3.9	80.9	99.6	80.6	442.9	84.3
74	4.0	2.8	2.3	2.8	5.0	6.8	4.1	12.5	5.5	12.1	27.8	16.0	3.9	83.1	98.0	83.9	444.5	87.7
76	3.8	2.8	2.2	2.7	5.0	6.5	4.0	12.1	4.8	11.1	26.9	15.1	4.1	88.6	106.1	89.6	464.6	93.7
78	4.1	3.0	2.3	2.8	5.2	6.9	4.0	12.8	5.0	11.8	28.5	15.9	4.1	86.9	112.3	88.0	460.4	91.8
80	4.3	3.2	2.6	3.2	5.9	7.5	4.3	13.4	5.3	12.5	29.7	16.9	4.1	89.6	110.8	89.6	475.8	93.8
82	4.0	3.0	2.5	3.0	5.3	7.0	4.1	13.6	5.7	12.8	29.9	16.8	4.4	97.5	116.0	96.0	491.2	100.5
84	4.3	3.1	2.5	2.9	5.5	7.3	4.6	13.6	6.1	13.2	30.2	17.8	4.4	102.6	123.7	102.4	512.4	106.4
86	4.6	3.4	2.8	3.3	5.9	7.9	4.5	14.5	6.0	13.9	32.0	18.5	4.5	102.7	118.0	104.6	510.1	108.9
88	4.4	3.3	2.6	3.2	5.8	7.6	4.3	13.9	5.7	13.0	30.7	17.3	4.5	109.1	122.0	109.0	536.4	113.6
90	4.4	3.2	2.7	3.2	5.8	7.7	4.9	14.7	6.3	13.8	32.0	18.8	4.7	108.2	117.1	110.8	528.7	115.6
92	4.8	3.7	3.0	3.7	6.4	8.5	4.8	15.8	6.2	14.9	34.5	19.7	5.1	106.8	133.4	108.9	516.2	113.5
94	4.7	3.8	2.9	3.6	6.5	8.4	4.8	15.5	6.2	14.6	33.8	19.4	4.6	111.2	118.1	113.6	530.2	118.2
96	4.9	3.5	2.9	3.7	6.1	8.5	4.8	15.8	6.6	15.0	34.6	19.8	4.7	111.8	133.3	110.6	540.5	115.9
98	4.9	3.8	3.1	3.7	6.4	8.5	4.9	15.4	6.9	15.0	33.5	19.9	4.8	108.8	124.1	110.3	528.8	114.9
avg	*4.2*	*3.0*	*2.5*	*3.0*	*5.4*	*7.1*	*4.2*	*13.0*	*5.4*	*12.4*	*28.9*	*16.5*	*4.2*	*90.9*	*108.9*	*91.5*	*472.0*	*95.7*

4.4 Limited support solutions

Rather than permitting small errors in the constraint equations, an alternative is to directly limit the number of supports to be less than some upper limit *sl*. This can be done for any solution method by taking the output set of supports, choosing the *sl* largest magnitude supports, and then recalculating the output solution vector using only the chosen supports. The steps in the recalculation are:

1. For variables not chosen as supports, set the upper and lower bounds to zero. For the chosen variables, use the change of variables $x_j = u_j - v_j$.
2. Set up the weighted elastic program as described in "Obtaining the Output Vector" in Section 3.
3. Solve the resulting LP to find the values of the chosen supports.

This recalculation finds the solution that comes closest to satisfying the constraints using only the chosen supports. The assumption is that the variables having the largest magnitudes in the original solution are the most important and will give a solution that has the smallest relative squared error.

The *maxFS*-based algorithms provide a different way of choosing a limited number of supports. These methods choose supports in a sequence, each time choosing the candidate variable that most reduces some elastic evaluation measure. Thus we can simply choose the first *sl* supports returned by one of these methods, and then carry out the recalculation of values as described above. This should be faster since the solution process is truncated.

An experiment was conducted to test the selection of supports using the order of their return by a *maxFS*-based algorithm. Given a k-sparse input vector, an upper limit of $sl = k/2$ supports was imposed, for k ranging from 62 to 98, with a constraint error delta of zero and 10 trials for each method at each value of k. The other conditions are as described in Section 4.1. Methods compared include (i) a random selection of *sl* supports, (ii) the *sl* largest magnitude variables returned by the BP solution, and (iii) *maxFS*-based methods JP, B and C, with supports chosen in sequence with list lengths 1 and 7. Method M1 is omitted because the *maxFS*-based sequences will be identical with those for method B. Method A is omitted due to its previous poor results. The random selection method is included to give an idea of the relative scale of improvement provided by the other methods. No method is post-processed; there is no point with the *maxFS*-based methods since the number of supports is limited to less than the actual number in the input vector, and the selection of the largest elements makes this unnecessary in BP.

Note that in all experiments, the number of constraints not satisfied by a solution (by any method) is equal to $128 - sl$. This is a property of the random $\boldsymbol{A}$ matrices (of size 128×256).

The average relative squared errors are summarized in Table 7. The best results at each input sparsity at each list length are shown in boldface. Comparing to Table 5 we see that the relative squared errors are smaller in almost every case for the support-limited approach than for the constraint error delta approach, and with fewer supports (compare to Table 4). This suggests that limiting the number of supports may provide better denoising than allowing a nonzero equation delta.

Which method performs best depends on the input k-sparsity. At lower values (up to say input $k = 78$ or $0.61m$) the *maxFS*-based methods (JP, B, C) provide lower errors than BP, but the situation reverses at higher input sparsities. This breakpoint corresponds with the input sparsity beyond which the *maxFS*-based methods do not reliably recover the exact input solution. As expected, random selection of sl supports provides terrible results.

Solution times are summarized in Table 8. Times are generally less than those for equation error delta 0.1. This is mainly due to the omission of post-processing, and partly due to the earlier termination of the solution process.

In conclusion it appears that limiting the supports generally gives sparser solutions, with lower relative squared errors, in less computation time as compared to using equation error deltas. The *maxFS*-based methods are preferred up to about $k/m = 0.61$ (method B has the lowest average in this range), with selection of the sl largest magnitude BP+ supports thereafter.

5 Experiments: General linear systems

Sparse solutions for general linear systems that include inequalities and variable bounds are useful, e.g. for sparse portfolio selection where the risk is linearized (Kremer et al 2017), and classifier feature selection (Fung and Mangasarian 2004). The algorithms described previously can be modified to handle this case as well. One modification for simplicity is to limit the list of candidate zero-valued variables by eliminating from consideration variables that must be supports due to their bounds, e.g. $x \geq 1$.

Table 7: Average relative squared error for limited supports.

		List Length 1					**List Length 7**				
k	**sl**	**random**	**BP**	**JP**	**B**	**C**	**random**	**BP**	**JP**	**B**	**C**
62	31	1.440	0.125	**0.108**	**0.108**	**0.108**	1.281	0.199	0.143	**0.137**	0.143
64	32	1.357	0.156	**0.120**	**0.120**	**0.120**	1.328	0.181	0.136	**0.128**	0.136
66	33	1.293	0.165	**0.122**	0.126	**0.122**	1.365	0.132	0.113	**0.111**	0.113
68	34	1.381	0.132	**0.117**	0.122	**0.117**	1.309	0.149	0.143	**0.140**	0.143
70	35	1.281	**0.275**	0.289	0.307	0.289	1.378	**0.199**	0.216	0.212	0.216
72	36	1.439	0.181	**0.156**	0.163	**0.156**	1.401	0.169	0.140	**0.117**	0.140
74	37	1.485	**0.252**	0.296	0.259	0.296	1.446	0.274	0.303	**0.257**	0.303
76	38	1.300	0.295	0.311	**0.249**	0.311	1.316	**0.277**	0.341	0.306	0.341
78	39	1.348	0.233	0.266	**0.221**	0.266	1.468	**0.209**	0.269	0.218	0.269
80	40	1.384	0.279	0.338	**0.270**	0.338	1.388	**0.231**	0.333	0.303	0.333
82	41	1.549	**0.326**	0.453	0.370	0.453	1.383	**0.351**	0.418	0.359	0.418
84	42	1.476	**0.320**	0.466	0.395	0.466	1.388	**0.360**	0.405	0.400	0.405
86	43	1.545	**0.295**	0.376	0.403	0.376	1.550	**0.353**	0.542	0.507	0.542
88	44	1.610	**0.323**	0.481	0.423	0.481	1.449	**0.412**	0.625	0.514	0.625
90	45	1.529	**0.358**	0.629	0.582	0.629	1.657	**0.425**	0.545	0.513	0.545
92	46	1.576	**0.379**	0.572	0.471	0.572	1.494	**0.389**	0.662	0.605	0.662
94	47	1.540	**0.383**	0.581	0.476	0.581	1.624	**0.520**	0.865	0.785	0.865
96	48	1.470	**0.395**	0.609	0.575	0.609	1.507	**0.448**	0.755	0.670	0.755
98	49	1.553	**0.509**	0.846	0.798	0.846	1.515	**0.438**	0.793	0.683	0.793
avg		*1.450*	***0.283***	*0.376*	*0.339*	*0.376*	*1.434*	***0.301***	*0.408*	*0.367*	*0.408*

Table 8: Average solution times (seconds) for limited supports.

		List Length 1					**List Length 7**				
k	**sl**	**random**	**BP**	**JP**	**B**	**C**	**random**	**BP**	**JP**	**B**	**C**
62	31	0.0	0.1	1.8	1.8	3.5	0.0	0.1	11.7	12.0	23.0
64	32	0.0	0.1	1.8	1.8	3.5	0.0	0.1	12.3	12.5	24.1
66	33	0.0	0.1	1.9	1.9	3.7	0.0	0.1	12.8	13.0	25.1
68	34	0.0	0.1	1.9	2.0	3.7	0.0	0.1	12.8	13.2	25.0
70	35	0.0	0.1	1.9	2.0	3.7	0.0	0.1	13.1	13.3	25.2
72	36	0.0	0.1	2.0	2.1	3.9	0.0	0.1	13.6	13.9	26.3
74	37	0.0	0.1	2.1	2.1	4.0	0.0	0.1	14.0	14.3	27.7
76	38	0.0	0.1	2.1	2.1	4.0	0.0	0.1	14.1	14.4	27.4
78	39	0.0	0.1	2.1	2.2	4.1	0.0	0.1	14.4	14.8	28.2
80	40	0.0	0.1	2.2	2.2	4.3	0.0	0.1	14.8	15.1	28.5
82	41	0.0	0.1	2.2	2.3	4.3	0.0	0.1	14.8	15.2	28.6
84	42	0.0	0.1	2.3	2.4	4.5	0.0	0.1	15.2	15.4	29.3
86	43	0.0	0.1	2.3	2.4	4.4	0.0	0.1	15.6	16.3	30.8
88	44	0.0	0.1	2.4	2.5	4.6	0.0	0.1	16.9	17.1	32.4
90	45	0.0	0.1	2.5	2.6	4.7	0.0	0.1	16.3	16.7	31.7
92	46	0.0	0.1	2.5	2.6	4.8	0.0	0.1	16.6	17.2	32.2
94	47	0.0	0.1	2.5	2.6	4.9	0.0	0.1	16.9	17.3	32.8
96	48	0.0	0.1	2.5	2.6	4.9	0.0	0.1	17.2	17.7	33.2
98	49	0.0	0.1	2.6	2.7	5.0	0.0	0.1	17.5	18.1	34.1
avg		*0.0*	*0.1*	*2.2*	*2.3*	*4.2*	*0.0*	*0.1*	*14.8*	*15.1*	*28.7*

We tested the following algorithms:

MILP. This is a standard mixed-integer linear program that seeks to minimize the number of supports. As noted in previous work (e.g. Jokar and Pfetsch (2018)), the "big-M" formulation causes too many solution failures to be useful. Results are omitted for this reason.

genLP. Solve the LP with no objective function, which in MOSEK stops at the first feasible solution found. Count the number of supports.

genBP. Like Basis Pursuit, minimize the sum of the absolute values of the potential zero variables using the standard change of variables. Count the number of supports.

GenJP, GenA, GenB, GenC. The general linear system version of the corresponding $maxFS$-based algorithms described earlier. Changes to the earlier algorithms include:

1. Handling bounded variables. (i) Some variables must be supports due to bounds that restrict them to nonzero values; genA omits zeroing constraints for these, and the other methods omit them from the objective function. (ii) For elastic minimization, nonnegative variables have an objective weight of 1.0, nonpositive variables have a weight of -1.0, and variables that span zero are split into two nonnegative variables via the change of variables $x_j = u_j - v_j$ where both u_j and v_j have objective weights of 1.0.
2. All methods output a fall-back result if the main method times-out, or fails, or produces a result that is worse than the fall-back. The fall-back for GenA is the result from the single initial non-elastic LP solution (i.e. genLP). The fall-back for GenJP, GenB, and GenC is the solution having the smallest number of supports at any intermediate elastic solution.

Post-processing. This has two stages. The first stage solves the LP with all non-supports forced to zero and with no objective function (this simply checks feasibility); if any support variables are equal to zero in the solution they are removed from the support set. In the second stage, variables are forced to zero one by one and if the LP has a feasible solution with the reduced set of supports, the variable is removed from the support set permanently. The output of stage 1 is the input to stage 2.

5.1 Experiments

The test set consists of the LP models in the Netlib (2018) repository and some larger models from the *kennington* set at the ZIB (2018) repository, for a total of 114 models (the Netlib *forplan* model is omitted due to difficulties in reading the file). Table 9 summarizes some overall statistics for the models (since these are linear programs, the statistics include the objective row and nonzeros). Total time for any solution is limited to 3600 seconds for the base algorithm plus post-processing.

Table 9: General linear system statistics.

	average	**median**	**maximum**
Variables	12,535.8	1,595.5	232,966.0
Rows	3,696.2	621.5	105,127.0
Nonzeros	61,047.1	8,214.0	1,630,758.0

There are 12 models which have zero supports; this solution is returned by all of the methods. There are an additional 15 models for which all methods return the same non-zero number of supports. GenLP fails on one very large model. No *maxFS*-based run fails completely due to the availability of the fall-back solutions.

Table 10 summarizes the results. Given the failure of the exact MILP algorithm, we can only evaluate relative performance. We compare the result found for each method to the fewest supports returned by any method for a given input model. Of the individual methods, genJP with unlimited list length finds the fewest supports the most often (65 of 114 models), and with list length 1 finds the largest number of unique minima. GenJP with list length 1 has the lowest average difference from the fewest supports (6.8). The best result in each column is shown in boldface.

Multiple methods can be run concurrently on a multi-core machine. One promising combination of methods is to run genJP at all 3 list lengths and take the best result ("best of genJP"): this returns the fewest supports for 89 of 114 models (78%) and has a significantly lower average difference from fewest supports (1.2). It also finds 15 unique minima if it replaces the 3 list lengths.

Table 10: Overall results for general LPs.

		Times fewest of 114	**Unique minima**	**Avg diff from fewest supports**	**Largest diff from fewest supports**	**Total solution time (hrs)**
	genLP	37	0	210.4	12819	3.2
	genBP	42	1	96.2	3149	**2.4**
	LL 1	60	*5*	*6.8*	*315*	8.3
genJP	**LL 7**	61	4	22.3	770	15.4
	unlim	*65*	3	73.2	2201	41.5
best of	**genJP**	**89**	**15**	**1.2**	**29**	41.5
	LL 1	38	1	218.4	12819	9.1
genA	**LL 7**	38	0	226.0	12819	13.4
	unlim	53	0	267.7	12819	43.9
	LL 1	57	0	25.0	1342	8.5
genB	**LL 7**	58	1	37.3	1128	16.1
	unlim	60	1	107.9	3816	42.2
	LL 1	57	0	29.2	1448	8.9
genC	**LL 7**	56	0	57.6	3361	17.3
	unlim	58	1	142.2	3807	60.3

Table 10 also shows the solution time totalled over the 114 solutions. GenLP and genBP are very fast since they require the solution of a single LP. For the other methods, as the list length increases, so does the solution time. The total solution time for "best of genJP" is the sum of the maximum solution times for each model over the 3 list lengths. Table 11 gives information on the number of time-outs, which impacts the total solution time. The number of time-outs increases with the list length, as expected.

GenLP times out on 1 model (and has no fall-back solution), while genBP succeeds for all models. Table 11 gives the exit status for the *maxFS*-based methods. The *maxFS* column gives the number of models for which the *maxFS*-based main algorithm returns the solution, while the *fall-back* column gives the number of models for which the fall-back method returns the solution. The two columns in italics show the number of time-outs and subproblem failures: these events lead to fall-back solutions as also happens when the fall-back solution cardinality is smaller than the cardinality of the solution found by the *maxFS*-based main algorithm. The main *maxFS*-based algorithm is rarely successful for genA: the fall-back genLP solution is usually better. The two genA failures are due to the inability to build the models in Matlab (it builds a dense matrix model before sparsifying it).

Table 11: Exit status for general LPs.

		MaxFS	**Fall-back**	***Time-out***	***Subprob. failure***
	LL 1	61	53	*5*	*1*
genJP	**LL 7**	55	59	*12*	*1*
	unlim	47	67	*34*	*1*
	LL 1	19	95	*4*	*2*
genA	**LL 7**	20	94	*9*	*2*
	unlim	31	83	*37*	*2*
	LL 1	54	60	*5*	*1*
genB	**LL 7**	54	60	*12*	*1*
	unlim	42	72	*35*	*1*
	LL 1	54	60	*6*	*1*
genC	**LL 7**	53	61	*15*	*1*
	unlim	27	87	*59*	*0*

Table 12 shows the results of post-processing the general LPs. The "models" column shows the number of models which did not time out when running the main heuristic, thus allowing the post-processing routines to run. The remaining columns give the fraction of the outcomes in each category over the number of models indicated. "Stage 1 (2)" success indicates that the first (second) post-processing stage provides the improved output solution. There are a small number of failures in which the post-processing routine initial LP determines that the input set of supports does not allow a feasible solution, though this same set was verified before output from the main algorithm: this is due to minor tolerance differences or accumulated numerical errors. Post-processing is generally quite fast unless the input support set is very large. Stage 2 of post-processing is much more successful than stage 1.

Table 12: Post-processing results for general LPs.

		Models	**Stage 2**	**Stage 1**	**No improvement**	**Failure**	**Time out**
	genLP	113	0.593	0.044	0.336	0.018	0.018
	genBP	114	0.553	0.105	0.325	0.009	0.009
	LL 1	109	0.321	0.055	0.578	0.046	0.000
genJP	**LL 7**	102	0.265	0.069	0.627	0.039	0.000
	LL unlim	80	0.150	0.063	0.763	0.025	0.000
	LL 1	110	0.555	0.082	0.336	0.018	0.009
genA	**LL 7**	105	0.562	0.086	0.324	0.019	0.010
	LL unlim	77	0.273	0.052	0.649	0.026	0.000
	LL 1	109	0.422	0.037	0.514	0.028	0.000
genB	**LL 7**	102	0.412	0.039	0.520	0.029	0.000
	LL unlim	79	0.304	0.063	0.608	0.025	0.000
	LL 1	108	0.417	0.037	0.519	0.028	0.000
genC	**LL 7**	99	0.424	0.040	0.505	0.030	0.000
	LL unlim	55	0.291	0.055	0.636	0.018	0.000

Method genJP is a little better overall than methods genB and genC for finding sparse solutions to general LPs. The "best of genJP" is markedly better. GenA returns much worse results than all other methods.

6 Discussion

In the compressed sensing scenario (underdetermined systems of linear equations in unbounded variables, with dense random $\boldsymbol{A}$ matrices) the best methods for exact recovery of sparse input vectors are B+ and M1+, which provide exact recovery with high probability up to $0.55m$ at list length 1, at the lowest relative squared errors. M1+ continues to provide the smallest relative squared errors beyond that point as well. Both methods require significantly more solution time that Basis Pursuit.

For approximate solutions in the compressed sensing scenario by allowing a constraint error delta, methods B+ and M1+ are again the best in terms of compression and relative error delta, though all of the *maxFS*-based methods (with the exception of A+) do well. The list length makes relatively little difference. When limiting the number of supports, the best results are given by relying on the sequence of supports returned by the *maxFS*-based methods JP, B, and C at lower input sparsities (up to say input sparsity $0.61m$), but BP+ is better at higher values of k-sparsity. Limiting the supports generally gives sparser solutions, with lower relative squared errors, in less computation time as compared to using constraint error deltas.

For general linear systems that include inequalities and variable bounds, and whose $\boldsymbol{A}$ matrices are usually sparse, the *maxFS*-based genJP formulation returns the smallest cardinality solutions on average at list length 1 and records the most minimum-cardinality solutions when the list length is unlimited. The best results are given by running all three list lengths of genJP concurrently.

There are several interesting observations from the experiments:

- Formulations for a *maxFS*-based approach to the sparse solutions problem can use constraint-based evaluations (minimizing elastic variables on the constraints) or variable-based evaluations (elastic variables on explicit variable zeroing constraints, or minimizing variable magnitudes). As the experimental results show, variable-based evaluations are far superior. Methods A and genA use constraint-oriented evaluations and provide far worse results than all of the other maxFS-based methods.
- In the compressed sensing scenario:
  - All of the *maxFS*-based methods (except A+) give better sparse solution recovery than Basis Pursuit, and are able to do so at higher k/m ratios. The drawback is longer solution times.
  - Limiting the number of supports based on the order that they are selected in a *maxFS*-based algorithm gives greater compression at lower relative squared error in less computation time than allowing a constraint error delta. This may indicate that limiting the supports in this way is a better option for denoising.

7 Conclusions

This paper examines the use of *maxFS*-based algorithms for finding sparse solutions to linear systems under several scenarios: (i) underdetermined linear systems of equations in unbounded variables with random matrices as seen in compressed sensing applications, including approximate solutions obtained by allowing constraint equation errors and limiting the number of supports, and (ii) general linear systems including inequalities and bounded variables. Experimental results show that the *maxFS* approach provides very good results.

The main contributions are:

- Several new and effective *maxFS*-based methods for finding sparse solutions of linear systems.
- Two of the new methods (B+ and M1+ at list length 1) allow greater compression of the transmitted signal in the compressed sensing scenario.
- A new method with potential for improved denoising in compressed sensing: limiting the number of supports by using the order in which they are found by a *maxFS*-based method.
- The first methods specifically developed for general linear systems.

- The observation and experimental confirmation that variable-based subproblem evaluations are much more effective than constraint-based evaluations (e.g. the sum of the constraint elastic variables) in the *maxFS*-based methods for finding sparse solutions.

There is wide scope for future research, including:

- Exploring other *maxFS*-based formulations and methods. The wide variety of formulation options holds out hope that even better methods await discovery.
- Finding ways to determine the best parameter values for approximate solutions in the compressed sensing scenario. What is the best equation error delta to use? Use absolute error or relative error? Use a capped relative error? What is the best upper limit on the number of supports?
- Multiple methods can be run simultaneously on multi-core machines. What is the most effective combination of methods to run in each of the scenarios studied?

References

KP Bennett, EJ Bredensteiner (1997). A Parametric Optimization Method for Machine Learning, INFORMS J. Comput 9:(311–318).

AB Bordetskii, LS Kazarinov (1981). Determining the Committee of a System of Weighted Inequalities. Cybernetics, 17(6):766–772.

K Bryan, T Leise (2013). Making Do With Less: an Introduction to Compressed Sensing, Siam Review 55(3):547–566.

EJ Candes (2008). The Restricted Isometry Property and its Implications for Compressed Sensing, Comptes rendus mathematique, 346(9–10):589–592.

SS Chen, DL Donoho, MA Saunders (2001), Atomic Decomposition by Basis Pursuit, SIAM review 43(1):129–59.

JW Chinneck, (1996), An Effective Polynomial-Time Heuristic for the Minimum-Cardinality IIS Set-Covering Problem, Annals of Mathematics and Artificial Intelligence 17:127–144.

JW Chinneck (2001), Fast Heuristics for the Maximum Feasible Subsystem Problem, INFORMS Journal on Computing 13(3):210–223.

DL Donoho (2006), Compressed Sensing, IEEE Transactions on Information Theory 52(4):1289–1306.

GM Fung, OL Mangasarian (2004). A Feature Selection Newton Method for Support Vector Machine Classification, Computational optimization and applications, 28(2):185–202.

S Jokar and ME Pfetsch (2008), Exact and Approximate Sparse Solutions of Underdetermined Linear Equations, SIAM Journal on Scientific Computing 31(1):23–44.

PJ Kremer, S Lee, M Bogdan, S Paterlini (2017). Sparse Portfolio Selection via the Sorted l_1-Norm, arXiv preprint arXiv:1710.02435.

SG Mallat, Z Zhang (1993). Matching Pursuits With Time-Frequency Dictionaries, IEEE Transactions on Signal Processing 41(12):3397–3415.

OL Mangasarian (1999). Minimum-Support Solutions of Polyhedral Concave Programs, Optimization 45:(149–162).

MOSEK (2018). The MOSEK Optimization Software, online at http://www.mosek.com.

Netlib (2018). The Netlib LP models, http://www.netlib.org/lp/data/index.html.

YC Pati, R Rezaiifar, PS Krishnaprasad (1993). Orthogonal Matching Pursuit: Recursive Function Approximation with Applications to Wavelet Decomposition. In Signals, Systems and Computers, Conference Record of the Twenty-Seventh Asilomar Conference, ed. A Singh, pp. 40–44. IEEE, Piscataway, NJ.

ME Pfetsch (2008). Branch-And-Cut for the Maximum Feasible Subsystem Problem, SIAM J. Optim. 19:(21–38).

ZIB (2018). Kennington LP models, http://www.zib.de/koch/perplex/data/kennington/mps/.