# An Algorithm for Linear Programming

Math à la Carte Report 2005-1

November, 2005

Fred T. Krogh

**Abstract**

This report documents some of the new algorithmic ideas in the code tentatively identified as **dopt**. This code is intended in the future to incorporate features beyond those described here.

## Contents



## 1 Introduction

Our problem of interest is:

$$\text{Subject to bounds on } C\mathbf{x} - \mathbf{b} \text{ and } \mathbf{x}, \qquad \text{minimize } \mathbf{c}^T\mathbf{x}, \tag{1}$$

where $\mathbf{x}$ is a real $n$-vector $C$ is a real matrix with $m_c$ rows and $n$ columns, and $b$ is a real $m_c$-vector. The problem is *linear* in the sense that with exact arithmetic it can be solved exactly in a finite number of steps. When upper and lower bounds are equal the condition is treated as an equality constraint. It is convenient to treat $\mathbf{c}^T$ as a row of $C$ with an unbounded value, and this is what the algorithm does.

Unlike the simplex algorithm, our algorithm makes no effort to stay on a vertex and does not introduce slack variables to deal with linear inequalities.

The algorithm internally works in a transformed coordinate system. The variables in the current coordinate system are identified by $\hat{\mathbf{x}}$. A given $\hat{x}_k$ may correspond to an original $x_j$ or to a transformed variable generated when making a constraint active. In the internal coordinates $\hat{\mathbf{a}}_i^T\hat{\mathbf{x}} - \hat{b}_i$, may refer to some other row in the original matrix or to one of the original variables. We (at least attempt to) attach a $\hat{}$ to variables in the internal coordinate system. $\hat{C}$ overwrites $C$ in memory. Initially the first $n$ cells of $\hat{\mathbf{x}}$ are identical to $\mathbf{x}$, and the following $m_c$ cells are used for the initial value of $C\mathbf{x} - \mathbf{b}$.

The first $n$ cells of $\hat{\mathbf{x}}$ one can think of as variables of which the last $\alpha$ are *active* and currently have fixed values. The first $n - \alpha$ of these variables are free to change in value, although at a given time only a subset of them do so. A method of this type is said to employ an *active set* strategy.

Variable transformations are accomplished using elementary transformations from the right. This is like the standard LU factorization. (We assume the reader has the kind of background provided in books on computational linear algebra such as[**?**] when referring to things such as Gaussian elimination, Householder transformations, Givens rotations, etc.)

If we assume the constraint row is $\hat{\mathbf{a}}_i^T$, and that the pivot element is $a_{i,p}$ The transformations involve multiplying $\hat{C}$ from the right with a matrix of the form $I + \mathbf{e}_p\mathbf{t}^T$, where $\mathbf{e}_p$ is 0 except for a 1 in row $p$, $t_k = 0$ for $i \geq p$ and for $i < p$, $t_j = -\hat{a}_{i,j}/\hat{a}_{i,p}$. At the same time the vector $\hat{\mathbf{x}}$ must be updated by multiplying it by the inverse of this matrix, $I - \mathbf{e}_p\mathbf{t}^T$. A given row of physical storage may thus contain several logical rows. The right most part of the row will always corresponds to original rows in the constraint matrix, but when such a row is active other parts would corresponds to variables which have been converted to general constraints. All but the leftmost part of the row corresponds to active constraints. The value associated with the right hand side is now corresponds to the value for the inactive variable now stored in the leftmost part of the row.

The algorithm insures that variables are always logically ordered from lowest indexed to highest (or left to right) as: inactive simple variables, active variables and active inequalities, factored equalities, equalities on bounds, and variables to be ignored. Finally in the code, variables with an index $> n$ correspond to inactive variables which have become general constraints and then other inactive inequalities.

When thinking about the various transformations we sometimes find it useful to imagine that we start with additional rows in our coefficient matrix of the form $I\hat{\mathbf{x}} = \mathbf{x}$. As we apply transformations this changes to $T\hat{\mathbf{x}} = \mathbf{x}$, and for every row here that is not a row of the identity matrix, there is a row someplace else that has been factored. We recommend thinking in these terms if you find something confusing about the transformations.

Let us mention a few of the terms used later.

| | |
|---|---|
| Active Set | The set of variables which are on a bound and not free to change in value. |
| Inactive Set | The set of variables which are free to change in value. A subset of these change on each iteration. |
| General Constraint | A constraint involving a linear combination of what are currently the variables. |

# 2 The Karush Kuhn Tucker (KKT) Conditions

The current gradient, $\hat{\mathbf{g}}$, gives the direction in which the objective is increasing most rapidly. If $\hat{g}_i$ is positive we would like to decrease $\hat{x}_i$ in order to reduce the objective. If $x_i$ is setting on a lower bound such a decrease is not possible and $\hat{g}_i$ is said to satisfy the KKT conditions with respect to that condition.

The *Lagrange multipliers* or *dual variables* at a solution satisfy $[IC^T]\boldsymbol{\lambda} = \mathbf{g}$, (where the $I$

comes from the conditions on the first $n$ $x_i$) and since we can multiply this equation on the left by the transpose of the transformations we are making, we can also write $[I\hat{C}^T]\boldsymbol{\lambda} = \hat{\mathbf{g}}$. The $\lambda_i$'s associated with the inactive constraints are all 0. Thus the conditions on the Lagrange multipliers and on $\hat{g}$ can be written

| Condition | Active State | Satisfies KKT | Violates KKT |
|---|---|---|---|
| $=$ | Satisfied | always | never |
| $\geq$ | On a lower bound | $\hat{g}_i \geq 0$ | $\hat{g}_i < 0$ |
| $\leq$ | On an upper bound | $\hat{g}_i \leq 0$ | $\hat{g}_i > 0$ |

# 3 Getting Started

If the user doesn't specify an initial point, $\mathbf{x} = \mathbf{0}$ is used. The first step in the starting procedure is to ensure that all the initial variables satisfy their bounds. Any that do not are immediately set to the nearest bound, and the $\hat{\mathbf{x}}_i$, $i > n$ are updated to account for the change.

Any variables that do not effect the value of any $\hat{x}_i$ for $i > n$ are permuted to a high index where they can be neglected from this point on. Such variables are set to the value of the bound that gives the minimal value of the objective. If the bounds are equal for a variable that variable is immediately converted to an equality on that variable and the variable is permuted to the right (a higher index) where it is ignored from this point on.

Empty rows are likewise permuted to a location where they no longer need to be examined.

By default rows and columns are scaled to balance the sizes of the rows and columns appearing in $\hat{C}$. This simplifies the later selection of good pivots, and the tests that are made for removing a constraint from the active set.

The next step is to process the equality constraints. After the factorization, the $\hat{\mathbf{x}}_i$ for $i > n$ are updated to account for the change necessary to satisfy the equality constraint. Note that in this process some of the original variables have turned into general constraints, and that these constraints may be violated as a result of satisfying the equality conditions. The variables and columns that have become active as a result are ignored except when done, where a backsolve is necessary in order to compute the associated Lagrange multipliers.

Thus when done with the equalities there are fewer variables, and no equality constraints. Unfortunately at this time we may have a number of constraints that are not satisfied. However constraints on variables $\hat{x}_i$ for $i \leq n$ are always satisfied except for some drift which can occur when having problems with rounding errors or in rare cases when having having trouble getting feasible.

# 4 Getting Feasible

When constraints are violated, the algorithm substitutes is own objective in place of that from the user. The objective in this case is to minimize the $L_1$ norm of the violated constraints. The gradient in this case is made up of additions and subtractions of the rows of $\hat{C}$ corresponding to the violated constraints. The row will be added in if the upper bound is

violated, and subtracted if the lower bound if violated. The algorithm is very similar to that for the linear programming case, except that checks must be made for previously violated constraint becoming feasible and when they do to adjust the gradient for the change. There are no infeasible constraints during this phase as what were infeasible constraints initially have been converted into being part of the objective during this phase. The algorithm used is quite similar to that described for the linear programming case below.

# 5 Linear Programming

This algorithm has 3 main components: Selecting which components to include in the next update of $\hat{\mathbf{x}}$, deciding how far to move in that direction, and when a constraint restricts a move to decide on a pivot column.

Most of the work occurs in making transformations of variables and in checking the KKT conditions, so some extra work elsewhere is likely to be helpful if it reduces on average the number of these actions. Given a direction, $\mathbf{d}$ in which to move, each $\delta\hat{x}_i$ that is part of that direction contributes to an inner product that defines how much to change the $\hat{x}_i$, $i > n$ for a given move $\mu\mathbf{d}$. Each nonzero component of $\mathbf{d}$ adds to this cost so it makes sense not to include directions for which there is a fairly high probability that the desired direction will change as a result of future transformations. Finally there is a cost of deciding just how far it is possible to move in a given direction.

Each nonzero $d_i$ is equal to $-\hat{g}_i$. Other strategies we have tried have not been as successful. The choice of components to include in $\mathbf{d}$ is simply based on the size of the $|\hat{g}_i|$. We include the largest one, and any that are "close". The initial choice of $\mu$ is simply the smallest value which just reaches a bound for an associated $x_i$. If a value of 0 is computed, then that variable is made active and at least for the time being is not considered for inclusion in $\mathbf{d}$. This situation changes if this $\hat{g}_i$ at some later point is a sufficiently large violation of the KKT conditions.

Using the inner products, the individual $\hat{x}_i$, $i > n$ are checked to find the first, if any, that is reached. If so, a pivot is selected and transformations performed to convert the $\hat{x}_i$ from being a general constraint into being a variable, and the whole process starts over.. Otherwise the variable that is reached is made active, and if there is more than 1 variable left in $\mathbf{d}$ the inner products are updated to account for the variable just removed, a new $\mu$ is computed and the search repeated.

Of course if the $\mu$ is unbounded and no constraint is encountered, then the minimum is unbounded. And if the KKT conditions are all satisfied and all constraints are active we have found the minimum.

For the pivot column we select the column with the coefficient of largest magnitude, after adding a penalty for pivot choices which would have the new constraint immediately fail the KKT test.

When the code is not making much progress, factors are attached to constraints which get smaller the more times a constraint goes active or inactive. Variables with small factors are less likely to be made active when more than one constraint limits the move, and are less likely to be made inactive as a result of checking the KKT conditions. Because rounding errors can make progress impossible on problems which should be solved, constraint violations are

ignored when inner products are quite small.

It may happen that a equality, has no pivot choice which is larger than might be expected from computational noise. Such an equality is dependent on others. If it also has a large residual then the equalities are inconsistent and the computations is terminated with an indication that this is the case. Otherwise the equality is redundant and is moved into an area where it is ignored.

This same dependency problem can occur with the gradient. That is, all directions which are available have $|\hat{g}_i|$ values which are so small that their sign is in question. In this case an option in the code lets one choose between accepting the current point as the solution, or continuing the computation by attempting to minimize $x_i$ for the smallest value of $i$ which allows such freedom. This process is then continued until no freedom is left thus, insuring the the solution point returned lies on a vertex, *i.e.* the solution will not be an interior point of some hyperplane.