Mathematical Programming 31 (1985) 153-191
North-Holland

# CONOPT: A GRG CODE FOR LARGE SPARSE DYNAMIC NONLINEAR OPTIMIZATION PROBLEMS

Arne DRUD

*Development Research Department, The World Bank, 1818 H Street, N.W., Washington, DC 20433, USA*

Received 26 August 1983
Revised manuscript received 5 September 1984

The paper presents CONOPT, an optimization system for static and dynamic large-scale nonlinearly constrained optimization problems. The system is based on the GRG algorithm. All computations involving the Jacobian of the constraints use sparse-matrix algorithms from linear programming, modified to deal with the nonlinearity and to take maximum advantage of the periodic structure in dynamic models. The paper presents the main features of the system, especially the inversion routines and their data structures, the dynamic setting of tolerances in Newton's algorithm, and the user features in the overall packaging. The difficulties with implementing a practical GRG algorithm are described in detail. Computational experience with some medium to large models is presented, indicating the viability of CONOPT for certain real-life problems, particularly those involving almost as many constraints as variables.

*Key words*: Large-scale Systems, Dynamic Models, Optimization, Nonlinear Programming, Generalized Reduced Gradient, Nonlinear Constraints, Sparse Matrix Techniques, Dynamic Tolerances.

## 1. Introduction

Two classes of algorithms are currently available for solving large nonlinear optimization problems, i.e. problems with 100 to 1000 constraints and a similar number of variables. These classes consist of algorithms based on solving a sequence of linearly constrained problems (SLC algorithms), and algorithms based on the generalized reduced gradient approach (GRG algorithms) [3]. GRG algorithms attracted much attention after Colville's comparative study [8], but a recent comparative study by Schittkowski [29] has revived interest in SLC algorithms. The excellent large-scale code MINOS/AUGMENTED [25], based on SLC techniques, has reinforced this interest. Unfortunately, no comparative studies have looked at large-scale problems, mainly because the number of codes is very small; furthermore, formatting a large set of large-scale test problems for different codes is an enormous task. We are therefore currently not in a position to recommend one algorithm class over another for large problems, based on computational evidence.

The views and interpretations in this document are those of the author and should not be attributed to the World Bank, to its affiliated organizations or to any individual acting in their behalf.

The present paper presents a GRG-based optimization system called CONOPT, which is designed for large time-dependent nonlinear optimization problems. CONOPT can, of course, also be used to solve large static problems by defining only one time period.

The concepts in the GRG algorithm are quite simple but as in most conceptual algorithms, there are many undefined implementation details. The difference between a good and a bad GRG code therefore lies in the choice of data structures, in the detailed implementation of the different components, and in the way the components work together through data structures. Since the concepts of the GRG algorithm are assumed to be well known, this paper will contain little theory; instead, it will describe how some of the crucial GRG components have been implemented and why these implementation choices have been made. The description will concentrate on the general case of large-scale static models, and only elaborate on how the dynamic structure is used in a few subsections.

GRG algorithms can have several disadvantages. In particular, much effort can be needed to maintain feasibility far from the optimum, and recomputations of the Jacobian and subsequent reinversion of the basis after each line search can be expensive. The paper tries to explain how we have reduced these disadvantages by proper choice of dynamic feasibility tolerances and of data structures for the basis inverse. After an initial problem definition and a short description of the GRG algorithm in Sections 2 and 3, the paper describes the way CONOPT presents itself to the user, i.e. the way a user enters a model into CONOPT (Section 4). It also describes some important systems features that make CONOPT convenient for the user; these include completely dynamic core allocation, error reporting and recovery, and time-limit handling. Subsequent sections describe subcomponents of the GRG algorithm itself: computing the Jacobian (Section 5), choosing and inverting a basis (Section 6), computing the reduced gradient (Section 7), and computing the search direction (Section 8). Section 9 is concerned with details of the one-dimensional search, including how CONOPT restores feasibility, what tolerances it uses, and how it handles bounds on the variables. Section 10 describes an algorithm for finding a first feasible solution; the algorithm is very fast for almost feasible problems and is therefore suitable for restarting perturbed problems. The last two sections contain computational results with some medium to large models and outline areas for future research and development.

## 2. Notation and problem statement

### 2.1. *The static model*

In most of this paper we will be concerned with the time-independent part of CONOPT, and the model we will consider *internally* in CONOPT has the following format:

$$\min \quad x_j \tag{1-S}$$

$$\text{s.t.} \quad f(x) = b \quad \text{and} \tag{2-S}$$

$$l \leqslant x \leqslant u \tag{3-S}$$

where

$\boldsymbol{x}$ is an $m$-vector of optimization variables,

$f$ is a mapping from $\mathbb{R}^m$ into $\mathbb{R}^n$,

$\boldsymbol{b}$ is an $n$-vector of right-hand sides,

$\boldsymbol{l}$ and $\boldsymbol{u}$ are $m$-vectors of lower and upper bounds, some of which may be minus or plus infinity, and

$x_j$ is the $j$-th component of $\boldsymbol{x}$, usually a slack variable.

If a model has inequality constraints they will be converted into equalities by the input routines through the addition of properly bounded slack variables. Section 4 contains more details on the actual input.

## 2.2. *The dynamic model*

CONOPT can take advantage of the time structure of dynamic models provided the model has the following format:

$$\min \quad \sum_{t=1}^{T} x_{jt} \tag{1-D}$$

$$\text{s.t.} \quad f_t(x_t, x_{t-1}, \ldots, x_{t-p}) = b_t, \quad t = 1, \ldots, T, \quad \text{and} \tag{2-D}$$

$$l_t \leqslant x_t \leqslant u_t, \quad t = 1, \ldots, T, \tag{3-D}$$

where

$\boldsymbol{x}_t$ is an $m$-vector of optimization variables in period $t$,

$f_t$ is an $n$-dimensional function of constraint values in period $t$,

$\boldsymbol{b}_t$ is an $n$-vector of right-hand sides in period $t$,

$\boldsymbol{l}_t$ and $\boldsymbol{u}_t$ are $m$-vectors of lower and upper bounds in period $t$ (some of the bounds may be minus or plus infinity),

$x_{jt}$ is the $j$-th component of $\boldsymbol{x}_t$,

$p$ is the largest number of lags in the model,

$T$ is the time horizon, and

the values of the lagged variables $x_0, \ldots, x_{1-p}$ are known and fixed.

## 2.3. *Design assumptions*

There are many design decisions in any large-scale system, and choices must be based on assumptions about the models that the system will be used to solve. The static part of CONOPT is built around the following assumptions:

- all functions are twice differentiable,
- all functions are defined for all values of the optimization variables satisfying the bounds,
- $m$ and $n$ are large, i.e. greater than 50,
- the functions are sparse, i.e. the number of nonzero Jacobian elements in each equation is small,
- most functions are linear and the remaining functions have substantial linear parts, and
- the models are well scaled.

Although we assume many functions to be linear we still assume that the models are 'rather nonlinear'. By this vague term we mean that the optimal step length in most one-dimensional searches will be determined by nonlinearities and not by bounds.

For dynamic problems we make the additional assumptions:
- all variables appear unlagged at least once,
- the unlagged Jacobian, $\boldsymbol{J}_{tt} = \partial f_t / \partial \boldsymbol{x}_t$ has full row-rank at all feasible points,
- the structure of the functions or the sparsity pattern of $\boldsymbol{J}_{tt-l} = \partial f_t / \partial \boldsymbol{x}_{t-l}$, $l = 0, \ldots, p$, is time-independent,
- most data such as right-hand sides, bounds, and coefficients in linear terms are time-independent, and
- the number of time periods is small, usually less than 25.

## 3. A generic GRG algorithm

For ease of reference in the rest of the paper, the main steps in a GRG algorithm (see also [3]) will be reproduced here:

0. Read the Model Input.
1. Find a feasible solution, $\boldsymbol{x}^0$. Set the iteration counter $k$ to 0.
2. Compute the Jacobian $\boldsymbol{J}^k = \partial f / \partial \boldsymbol{x}^k$.
3. Separate the variables into $n$ basic variables and $m-n$ nonbasic variables, subscripted by $b$ and $n$ respectively, such that the current basis $\boldsymbol{J}_b^k = \partial f / \partial \boldsymbol{x}_b^k$ is nonsingular. It is preferable that the basis is well-conditioned and the basic variables are away from their bounds.
4. Compute the multipliers, $\boldsymbol{u}^{\mathrm{T}} = \boldsymbol{e}_{jb}^{\mathrm{T}} (\boldsymbol{J}_b^k)^{-1}$, and the reduced gradient, $\boldsymbol{g}_n^{\mathrm{T}} = \boldsymbol{e}_{jn}^{\mathrm{T}} - \boldsymbol{u}^{\mathrm{T}} \boldsymbol{J}_n^k$. (Superscript T denotes transpose, $\boldsymbol{e}_j$ is an $m$-dimensional unit vector with $+1$ in position $j$, and $\boldsymbol{e}_{jb}$ and $\boldsymbol{e}_{jn}$ are the basic and nonbasic components of $\boldsymbol{e}_j$, respectively.)
5. Stop if the current point satisfies the Kuhn–Tucker conditions.
6. Separate the nonbasic variables into superbasics, subscripted by $s$, and fixed nonbasic variables.
7. Compute a search direction for the superbasics, $\boldsymbol{d}_s$, based on $\boldsymbol{g}_s$, the superbasic part of the reduced gradient, and an estimate of the Hessian of the reduced objective, $\boldsymbol{H}_s = \partial^2 x_j / \partial \boldsymbol{x}_s^2$.

8. Perform a one-dimensional search along $d_s$. For each step length, $\Theta$, solve $f(x_b, x_s^k + \Theta d_s, x_n^k) = b$ for $x_b$ using $(J_b^k)^{-1}$ in a Newton-type procedure, and extract the value of the objective. The step length must be so small that all variables remain between their bounds.

9. Save the best solution found in step 8 in $x^{k+1}$, set $k = k+1$, and go to 2.

## 4. Model input and systems features

CONOPT's system features and overall packaging are of course irrelevant from an algorithmic point of view. Nevertheless, they are extremely important for improving the productivity of a practical user, especially an unsophisticated user; we have therefore chosen to devote some space to them. The description that follows is only an overview: it would be impossible to cover all details, since the non-optimizing parts of CONOPT account for more than half the code.

CONOPT is a FORTRAN program of around 25 000 lines, including comments. From the user's point of view, CONOPT is organized as a stand alone optimization system, which is called through a procedure at the operating system level. A model is defined for the CONOPT procedure through three or more files. The way in which the procedure is called and the way the files are defined is machine and operating system dependent, but the format of the files is machine-independent.

### *4.1. The MPS file*

CONOPT tries to stay as close as possible to the industry standard for Linear Programming (LP). A large part of the model is therefore defined through a modified MPS or CONVERT format file, as defined in the CDC/APEX III manual [30] or the IBM/MPSX manual [31]. The types of constraints (equal, less than or equal, greater than or equal, or nonbinding), the values of the right-hand sides, and the non-default upper and lower bounds on structural and logical variables are all defined in standard ROWS, RHS, BOUNDS, and RANGES sections. The COLUMNS section defines the sparsity pattern of the Jacobian. A nonlinear element in the Jacobian is identified by a special coefficient with a default value of 999 999. A linear element in the Jacobian can be identified by its numerical value or by another special coefficient with a default value of 9999. Bound sets with a name starting with INIT, e.g., INITIAL, are used to provide initial values for some or all of the variables.

The extensions to the MPS format used in CONOPT have been inspired by similar extensions used in other codes, e.g. [21, 24].

### *4.2. The FCOMP subroutine*

The values of the nonlinear and unspecified linear components of each constraint, i.e., the components identified by 999 999. or 9999. in the MPS file, must be supplied to CONOPT through a second file containing a FORTRAN subroutine called FCOMP. FCOMP can also contain constant additive terms that otherwise would have been defined in the RHS section. Derivatives are not defined by the user and are therefore not part of FCOMP. They are computed numerically by CONOPT, as will be explained in Section 5.

FCOMP is supplied with a vector of optimization variables that always satisfy the lower and upper bounds, and it must return all constraint components in another vector. It was originally considered that FCOMP might compute the value of one constraint, identified by an input parameter, in each call. This would facilitate the computation of derivatives and make block decompositions possible. The expected savings were not thought large enough, however, to compensate for the extra subroutine linkage and conditional branching overhead - and, above all, for the added complications for the user.

Since constraints and variables are identified by names in the MPS file and by indices in FCOMP it is necessary to define some mapping between the constraint names and the function indices, and between the variable names and the variable indices. The default mappings are defined by the positions of the constraint and variable names in the MPS file. These implicit mappings make it hard to modify a model without recoding large parts of FCOMP. CONOPT therefore includes an option for defining one or both mappings explicitly. A special right-hand side in the MPS file, named FUNCTION, maps the constraint names into function indices and a special bound set, named VARIABLE, maps the variable names into variable indices.

### 4.3. *The CONTROL program*

The third input file must contain a control program. We have implemented a procedural control program similar to the one in CDC's LP-system APEX III [30]. It has verbs for calling different CONOPT procedures like MODEL (real input), CHECK, OPTIMIZE, OUTPUT, WRITE (create coded restart file), SAVE (create binary restart file), and RESTART (restart from SAVE-file). Other verbs like SET and STEP can define so-called CR-cells containing tolerances, options, iteration and time-limits, and names of the selected right-hand side, bound set, range set, and initial value bound set. CRPRINT can display the CR-cells, TITLE can change the page headers, and the USER verb activates a user supplied FORTRAN subroutine that can initialize common blocks for FCOMP or print special reports. The remaining verbs, TEST (set condition code), BRANCH (multiway conditional branch), PERFORM (multiway conditional branch with saved return address), and NEXT (return to line after last PERFORM), can be used to define conditional execution sequences and simple subroutine constructions. The major executing verbs are all followed by a default BRANCH unless the user supplies his own BRANCH or PERFORM verb. For example the input verbs are followed by default by a branch that depends on whether a major or minor error has been encountered.

### 4.4. *Input checking*

One of the design objectives of CONOPT has been to ensure that as many as possible of the inconsistencies in the model input should be caught and reported to the user, so that expensive optimizations are not attempted on models containing errors. The MPS file is tested for undeclared row names, split columns, multiple definitions of the same information (row name, matrix element, bound value, etc.), consistent bounds, and all the other standard LP-tests. If a FUNCTION right-hand side or a VARIABLE bound set defines a mapping, CONOPT checks that the relationship is one to one and that indices are defined for all nonlinear constraints and variables.

All the tests described so far are relatively straightforward because all necessary information is readily available. The problem area, as in most other nonlinear optimization systems, is in the FCOMP subroutine. CONOPT does not know what is inside FCOMP; it can only call FCOMP with different $x$-vectors as arguments and observe function values returned by FCOMP. The tests currently implemented try to make sure that a constraint function in FCOMP does not depend on a variable unless it was declared in the MPS file, and that linear functions in FCOMP really are linear within tight tolerances. The presence of undeclared variables in a constraint is tested by evaluating the constraint at a random point satisfying the bounds, assigning new random values to all undeclared variables, and evaluating the constraint again. If the constraint value is different, it must depend on at least one of the variables that was changed, i.e. on one of the undeclared variables. The inconsistency is found by resetting the undeclared variables to their initial values one by one and observing when the constraint value changes. The test is very cheap if no errors are found, requiring only one FCOMP call per nonlinear constraint. It may not find an error if the undeclared constraint derivative is identically zero over a large area. This happens very infrequently, however, and the test has proved itself very useful in practice.

### 4.5. *Error recovery*

If all input tests are passed and the CONTROL program contains an OPTIMIZE verb, CONOPT will start optimizing. At this point some very undesirable things could happen. A division by zero or another arithmetic exception could cause the job to abort without giving the user any idea of what went wrong. Alternatively, the job could reach its time-limit and abort, leaving the user with nothing but the bill.

To avoid the first of these problems, CONOPT uses machine-dependent error recovery routines to regain control after an arithmetic exception. A logical variable will indicate whether the error happened in CONOPT or in FCOMP; if it happened in the latter, CONOPT will tell the user that he has made a mistake and will print all the information passed on to FCOMP as well as the content of the constraint vector at abort time. Depending on the FORTRAN compiler, there may also be an estimate of the line in which the error occurred. If the error happened in CONOPT, the system will write a message of apology and urge the user to submit the problem to the author so that CONOPT can be corrected.

The time-limit problem has also been eliminated. CONOPT checks through a machine-dependent routine how much time is left when it starts executing, and it stops when 80% of the time available has been used. This leaves sufficient time to save and/or print the solution. The default BRANCH that follows OPTIMIZE in the CONTROL program automatically calls SAVE and OUTPUT.

### 4.6. *Debugging facilities*

A system of the size of CONOPT is bound to have some bugs. Facilities for debugging have therefore been incorporated as an integral part of CONOPT. All major routines contain WRITE statements that describe the flow of control and the values of all important variables. Definition of CR-cells in the control program permits the test output to be turned on and off independently in more than 25 functionally distinct parts of the code; in most cases, the detail of the output can be varied. It is therefore possible to get a good picture of what happens in one part of CONOPT without being swamped by output from irrelevant parts.

When an error occurs in iteration 100 we are not interested in test output from the first 95 iterations. In this case we would, through the CONTROL program, set the iteration limit to 95, optimize until this limit is reached, reset the limit to 105, turn the selected test output on, and continue the optimization. Great care has been taken to retain all relevant information when an iteration sequence is interrupted, so that a continuation of the optimization will produce exactly the same iteration sequence as an optimization run without the interruption. It is even possible to SAVE a binary copy of the solution status and, after a RESTART in a later job, to produce exactly the same iteration sequence as would have been produced by an uninterrupted job. If an error occurs at the end of a very expensive job it is possible to rerun the job and save the status a few iterations before the error occurs, and then locate the error through a sequence of small inexpensive jobs.

### 4.7. *Memory management*

A final systems detail is that of memory management. The user never has to worry about dimension statements or allocation of working storage. On virtual storage machines like IBM, a large piece of working storage is reserved once and for all, and CONOPT manages its allocation for different purposes. On machines like CDC where memory is at a premium, CONOPT starts with a small amount of working storage and requests more memory when it is needed. Whenever a vector exceeds its initial allocation (during input for example), CONOPT reallocates the working storage and moves the information accordingly.

### 4.8. *Dynamic models*

The assumptions in Section 2.3 make it possible to define a dynamic model essentially by defining the model for one period and adding a few series of time-dependent quantities. The ROWS, COLUMNS, RHS, BOUNDS, and RANGES sections are therefore only defined for one period. The only difference from static models is that coefficients, right-hand sides, and bound values can be defined as time-dependent through the use of a special coefficient. The values of the time-dependent quantities are defined in a separate SERIES file.

The structure of each of the lagged Jacobians, $J_{tt-l}$, $l = 1, \ldots, p$, is defined in a LAGS section in the MPS file. The LAGS sections have the same format as the COLUMNS section.

The FCOMP subroutine is also written for one period only. It receives $x_t, \ldots, x_{t-p}$ plus the value of $t$ and must return the nonlinear part of $f_t(x_t, \ldots, x_{t-p})$.

## 5. The Jacobian

There are many advantages to using analytic derivatives. If properly coded, they are both more accurate and faster to compute than numerical derivatives. Analytic derivatives also have important disadvantages, however. It is labor intensive to code all derivatives by hand, especially for large models where the number of different expressions can be in the hundreds. In addition, it can be difficult for an unsophisticated user to understand the data structures into which he has to store the derivative values. Finally, the chances of error are rather large, although the system could use finite differences to run checks and to inform the user about likely errors.

Because of the above problems, it was decided to compute derivatives numerically. The linear Jacobian elements that are defined through FCOMP (hereafter called linear FCOMP elements) are computed in a setup phase. The columns with linear FCOMP elements are perturbed one by one, and the derivatives are computed via first differences. A fairly large perturbation is used to minimize the effect of round-off errors.

The nonlinear elements in the Jacobian are computed repeatedly, and one FCOMP call per nonlinear column per evaluation can become very expensive. A setup routine based on [9] is therefore used to organize the columns of the nonlinear Jacobian into groups such that each group has at most one nonlinear element in each row. The nonlinear elements are evaluated in groups: all columns in a group are perturbed, FCOMP is called, and each nonlinear element in the group is computed numerically as the ratio of the row and column perturbations. The direct use of the row perturbation assumes that the perturbation does not contain terms from linear FCOMP elements. Subtracting these terms from linear FCOMP elements would require either a search for the relevant elements or a special data structure to reference them. We have avoided this complication by slightly changing the group-building heuristic described in [9]: a column is simply not added to a group during the building process if this would mean that the group would get a row with both a linear and a nonlinear derivative.

The number of groups, and therefore the number of FCOMP calls needed to compute all derivatives, is generally quite small. The group-selection heuristics in

[7] should give fewer groups and therefore fewer FCOMP calls than the current techniques, but the savings in overall computer time are not expected to be more than a few percent and the change, although desirable, is of low priority.

The discussion of analytic derivatives at the beginning of this section assumed that a human user had to code the derivatives by hand. Eventually, however, we expect that more than half the problems submitted to CONOPT will be generated by computerized modeling systems like GAMS [5], in which case correct analytic derivatives can be generated automatically and inserted into any type of data structure. We are therefore considering adding an option so that the user can supply analytic derivatives. The option will be designed for computer-generated models and will probably not be friendly for a human user.

The matrix operations in a GRG algorithm are similar to those of an LP algorithm. The Jacobian is therefore stored in the same way, i.e. the elements are sorted by columns, their values and row numbers are stored in two parallel vectors, and a shorter vector points to the start of each column. CONOPT also stores a column number to facilitate other operations used mainly in dynamic models.

Dynamic models do not give rise to many complications. The structure of the Jacobian, shown in Fig. 1, is represented by the structure for one period, and all constant linear elements are stored only once. The values of nonlinear and time-dependent elements are stored with one copy per period. The two types of element, constant and time-dependent, are distinguished by being in separate data structures. The lagged Jacobians are usually very sparse with many empty columns. Only nonzero elements with row and column numbers are stored, since we never need to access the lagged Jacobians by columns.

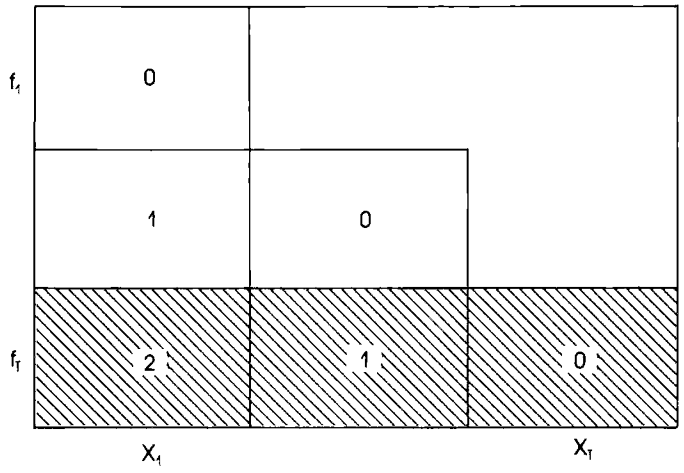

Fig. 1. The structure of the Jacobian of a dynamic model. The blocks with the same number have the same sparsity pattern, so the pattern for the hatched area is sufficient to represent the whole sparsity pattern. The stored row and column numbers are relative to the period, and the lag number is imbedded in the column number.

## 6. Basis selection and basis inversion

### 6.1. Static models

The word matrix-inversion is used throughout this paper to denote the creation of a factorization that makes it easy to solve sets of equations involving the matrix or its transpose.

The set of basic variables can in principle change from iteration to iteration as mentioned in step 3 in the GRG algorithm. We prefer to keep the same set as long as possible, however, either until a basic variable hits a bound in step 8, or until $J_b^k$ becomes badly conditioned. The reasons are that a stable basis set facilitates accumulation of second-order information, that the selection of a basis set involves some overhead, and that inversions of a sequence of basis matrices with the same sparsity pattern and similar numerical values can be done very efficiently.

Given this overall basis strategy, a basis selection and inversion routine will have to work in two modes, search mode and reinversion mode. In search mode, we select and invert a basis from a set of $n$ or more candidate variables, preferably choosing variables that are away from their bounds. This mode is used in the following cases:

1. For the initial basis, or after 2 or 3 below fails. The candidate set consists of all variables.
2. After a basic variable hits a bound in the one-dimensional search. The candidate set consists of all old basic and superbasic variables except the basic variable that hit a bound.
3. After a reinversion fails as described below. The candidate set consists of all old basic and superbasic variables.

In reinversion mode, we invert the same basis matrix as before, but with new numerical values.

CONOPT assumes that the nonlinearities of both constraints and objective are so strong that most one-dimensional searches end before a variable hits a bound. We often observe that more than 80% of all one-dimensional searches end in an interior point, and we assume that more than 80% of all inversions are reinversions. It is therefore important that reinversions are fast, even if some of the speed is gained at the expense of the search-mode inversions.

Our search-mode inversion routine is a modification of Hellerman's and Rarick's $P^3$ routine [17], used on the rectangular matrix of all basis candidates. The principles of $P^3$ are assumed to be known. The key features of our modification are as follows:

1. Columns are stacked and unstacked on a spike stack or selected as triangular columns as in $P^3$ on the basis of a tally function. The algorithm stops when $n$ pivot columns have been chosen, and the columns that are left on the spike stack become nonbasic or superbasic. In general, we expect that more basis candidates will result in fewer spikes in the basis, because most of the potential spikes will never be unstacked.

2. Hellerman and Rarick's tally function is augmented with a term that reflects the distance from bound. Variables close to their bounds are favored during spike selection, i.e. selection of potential superbasics, and variables far from their bound are favored during selection of triangular columns, i.e. selection of variables that definitely will become basic.

3. During the inversion, a row-wise representation of the sparsity pattern is added to the permanent column-wise representation, so as to facilitate searches through the rows of the matrix. The space needed for this representation is shared with vectors that are only used as intermediates in the one-dimensional search, so no extra core is needed.

4. The factorization of the basis is stored using alpha-vectors, following [19]. The triangular columns are embedded in the Jacobian itself and the updated spike columns are stored as additional columns of the Jacobian, using the standard Jacobian data structure. To avoid problems with zeros that becomes nonzeros at a different point, we create the logical structure of the updated column and use its sparsity pattern to decide which elements to store.

5. The pivot element of a column is accepted if it is numerically larger than max(RTPIVA, RTPIVR cnorm), where cnorm is the norm of the updated column and RTPIVA and RTPIVR are absolute and relative tolerances with initial values 0.05. If the pivot is too small, the column is stacked as a spike and the first column from the top of the spike stack satisfying the test is selected. If no such column exists, the inversion is considered a failure and we increase the set of basis candidates as described above. If all columns are already candidates, we accept the best pivot element provided that it satisfies some minimal tolerances. If not, we stop the job with a message that the Jacobian does not have full row-rank.

With the inversion scheme adopted here, reinversions are very fast. After the Jacobian has been recomputed, it is sufficient to update the spike columns and save their new nonzero values. Since we used the logical sparsity pattern during the search-mode inversion, no new nonzero positions will appear and no tests for zero are therefore needed. Nevertheless, it is still necessary to test that the pivot elements, both in the spike columns and in the triangular columns, are large enough. We use slightly smaller tolerances than in 5 above, so a reinversion will always be possible in the neighborhood of a point at which a successful search-mode inversion was performed.

An earlier version of the inversion routine, [11], was based on the $P^4$ routine, [18], with some additions to take care of the degrees of freedom in the basis selection. However, the vector orientation of FCOMP prevented the block structure from $P^4$ from being used in other parts of CONOPT and it was abandoned in favor of the simpler $P^3$ routine.

## 6.2. Dynamic models

One of the important differences between static and dynamic GRG codes comes from the inversion routine. In early work on GRG codes for dynamic models, it was assumed that a block triangular basis with square diagonal blocks corresponding to the time structure could always be found (see [2]). This would be very convenient, because it implies that the overall inverse basis can be represented by the inverse of the smaller diagonal blocks. Some classes of models do have this helpful basis structure, for example models that in each period have $n$ unbounded 'state variables', $x_t^s$, for which $\partial f_t / \partial x_t^s$ is always nonsingular. Some optimization models based on economic models belong to this class, and [22] describes a GRG code for this application.

Unfortunately, models with many active bounds in late periods may need non-square blocks in their bases as shown in Fig. 2. An early discussion of the problem in relation to GRG codes is given in [1]. Apart from the basis selection, the inversion in a dynamic GRG code is similar to the inversion in staircase LP, and some of the techniques from this field could be used. For further details and references, see [13]. There are, however, some important differences between general LP staircase models and time dynamic nonlinear models in CONOPT. The time orientation of FCOMP, and of the data structure for the Jacobian, makes it important to preserve as much of the time structure as possible in the factorization of the inverse, so that it can be used efficiently with single-period vectors of function values or single-period columns of the Jacobian. Fortunately, the extra freedom in the basis selection can help us select a basis that has square or almost square in-period blocks.

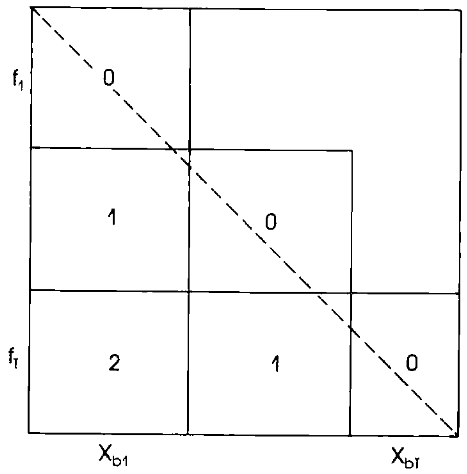

Fig. 2. A typical basis matrix for a dynamic model with nonsquare in-period or 0-lag blocks. The lower right corner of all 0-lag blocks must be on or above the diagonal for the basis to be nonsingular.

To take as much advantage of the structure as possible, CONOPT contains a special dynamic basis selection and inversion routine. It is based on the conceptual reorganization of the basis shown in Fig. 3. The basis contains in-period blocks, i.e. blocks of columns that have their pivot rows in their own period, and inter-period columns, i.e. columns that have their pivot rows in later periods. The inversion or

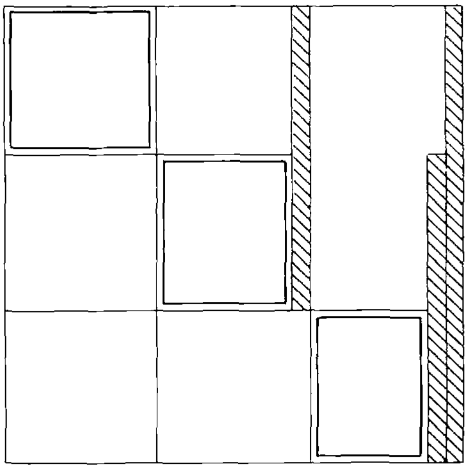

Fig. 3. The reorganized basis matrix with in-period blocks and inter-period columns.

factorization of this basis structure into easily invertible matrices is derived from the factorization in Fig. 4. For $t = 1$, $\boldsymbol{E}_{1t}$ is empty, and for $t > 1$ it is factorized by recursive uses of Fig. 4, i.e. $\boldsymbol{E}_{1t} = \boldsymbol{E}_{1t-1}\boldsymbol{E}_{2t-1}\boldsymbol{E}_{3t-1}\boldsymbol{E}_{4t-1}$. $\boldsymbol{E}_{2t}$ is analytically invertible and can be represented by $\boldsymbol{A}_t$ itself. $\boldsymbol{E}_{3t}$ is invertible through an inversion of $(\boldsymbol{C}_t, \boldsymbol{I})$ within period $t$ only. And the final factor $\boldsymbol{E}_{4t}$ can be split into a product of inter-period alpha-vectors using a product or elimination form of the inverse. The only factor that destroys the time structure is $\boldsymbol{E}_{4t}$. This is not serious for the computation of dual variables, reduced gradients, or tangent directions, because these operations are oriented towards the whole time horizon. However, $\boldsymbol{E}_{4t}$ must be integrated with the single-period FCOMP calls in the feasibility-restoring Newton steps. The implementation of this is described in Section 9.2.

The dynamic inversion and basis selection routine in CONOPT can now be summarized. It assumes as input a set of basis candidate columns. For $t = 1$ to $T$, execute steps 1 and 2:

1. Select and invert a submatrix of $\boldsymbol{J}_{tt}$ using the static basis selection and inversion routine described above. As before, the inverse basis is represented by triangular columns embedded in the in-period Jacobian and updated in-period spike columns saved separately. The procedure is modified so that it rejects rows without a good pivot element. These rows are logically assigned unit columns for pivot until final pivots from earlier periods can be assigned in step 2. The result of this step is the selection and inversion of $(\boldsymbol{C}_t, \boldsymbol{I})$ in $\boldsymbol{E}_{3t}$.

2. For each row $j$ without a final pivot element, replace the preliminary unit column with an inter-period pivot column using steps 2.1 to 2.3:

2.1. Find the vector of possible pivot elements by updating the pivot row using the usual formula: $\boldsymbol{c}^{\mathrm{T}} = (\boldsymbol{e}_j^{\mathrm{T}}\boldsymbol{B}_{1\to t}^{-1})\boldsymbol{J}_{1\to t} = \boldsymbol{\pi}_j^{\mathrm{T}}\boldsymbol{J}_{1\to t}$. $\boldsymbol{e}_j$ is the unit vector to be replaced, $\boldsymbol{B}_{1\to t}^{-1}$ is the current inverse basis, and $\boldsymbol{J}_{1\to t}$ is the Jacobian for period 1 to $t$. The BTRAN routine described in the next section is used.

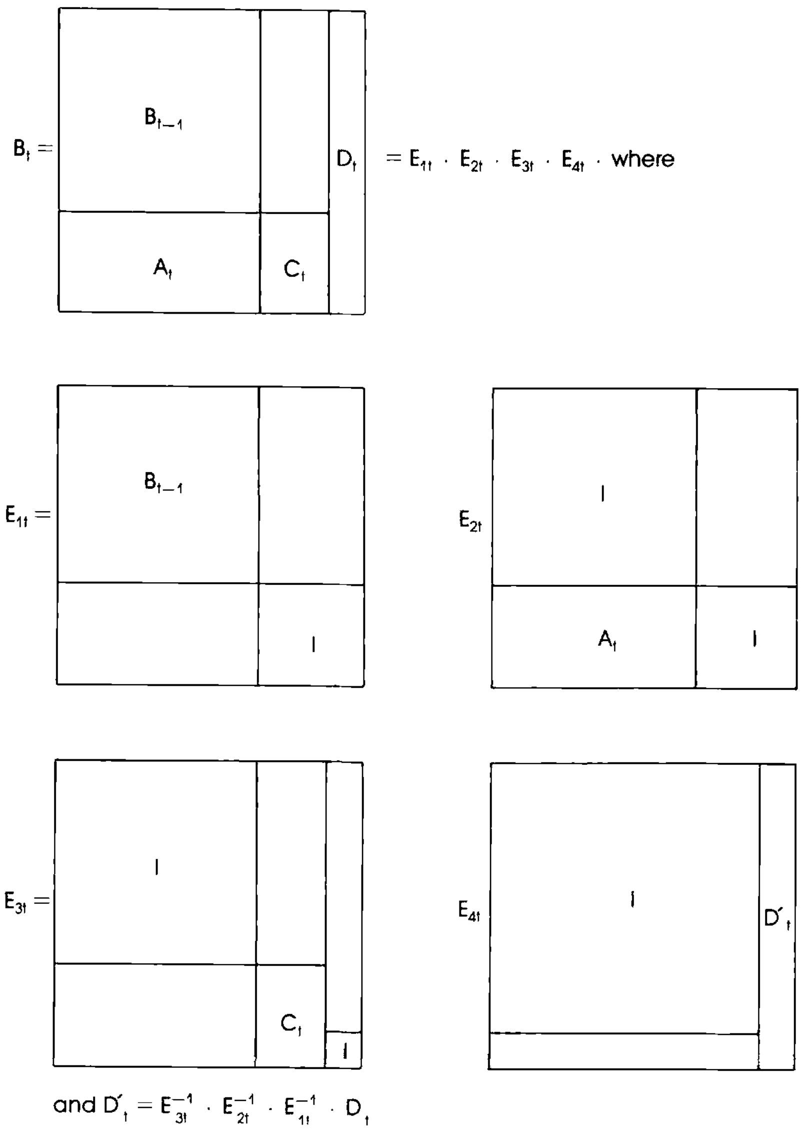
$B_t =$
$B_{t-1}$
$D_t$
$= E_{1t} \cdot E_{2t} \cdot E_{3t} \cdot E_{4t} \cdot$ where
$A_t$
$C_t$
$E_{1t} =$
$B_{t-1}$
I
$E_{2t}$
I
$A_t$
I
$E_{3t} =$
I
$C_t$
I
$E_{4t}$
I
$D'_t$
and $D'_t = E_{3t}^{-1} \cdot E_{2t}^{-1} \cdot E_{1t}^{-1} \cdot D_t$

Fig. 4. The factorization of a basis matrix into 4 simpler matrices. $\boldsymbol{B}_{t-1}$ is a square matrix that covers all periods before $t$, $\boldsymbol{A}_t$ and $\boldsymbol{C}_t$ correspond to the rows in period $t$, and $\boldsymbol{D}_t$ represent the columns that originate before period $t$ and have their pivot in period $t$. The empty rectangles contain only zeros.

2.2. Find a good pivot element in $c$. The pivot element is chosen to be greater than a fraction of $\max(c_i)$. First priority is given to columns from periods that are already linked to period $t$ with inter-period pivots. If no such column exists we choose a column from as late a period as possible, so as to make the inter-period alpha-vector as short as possible. The distance from bounds is used as a secondary criterion.

2.3. Update the selected pivot column $\boldsymbol{a}_i$ using $\boldsymbol{\alpha}_i = \boldsymbol{B}_{1\to t}^{-1}\boldsymbol{a}_i$ and store it as an inter-period alpha-vector with pivot in row $j$. The alpha-vector is considered part of $\boldsymbol{B}_{1\to t}^{-1}$ during the next pass of 2.1 to 2.3. The result of step 2 is the selection and inversion of $\boldsymbol{E}_{4t}$.

The set of basis candidates is managed as in the static routine. The degree of freedom in the selection of basic variables is used in step 1 to give the in-period blocks as many columns as possible, thereby decreasing the number of inter-period pivots. The freedom is also used in step 2.2 to make the inter-period alpha-vectors short; this saves storage and weakens the harmful effects of $\boldsymbol{E}_{4t}$ in the Newton step.

The dynamic reinversion routine is just as simple as the static one. In-period blocks are reinverted using the static reinversion routine, and the numerical values of the inter-period alpha-vectors are simply recomputed using step 2.3 above. No structural work is needed and the only tests are those that test the size of the pivot elements.

A special mixture of the reinversion and the basis selection and inversion routine is used after a basic variable hits a bound and leaves the basis. Let the time period of the variable that leaves the basis be connected with all time periods from $t_1$ to $t_2$ through inter-period pivots. We first perform a reinversion for period 1 to $t_1 - 1$. Between period $t_1$ and $t_2$ we perform a basis selection and inversion. The basis selection can choose variables before $t_1$ as pivots, thereby increasing the size of the inter-period block, and it can break the block into smaller blocks. It cannot affect periods after $t_2$, however, so a reinversion is sufficient for period $t_2 + 1$ to $T$.

## 7. The reduced gradient

The computation of dual variables or Lagrange multiplier estimates follows a standard LP BTRAN procedure. The dual variables are subsequently used in a pricing procedure where CONOPT prices all columns, nonbasic as well as basic. The nonbasic reduced costs define the reduced gradient. The basic reduced costs should be zero and can therefore be used to test for the numerical stability of the basis. If they are large compared to intermediate results and nonbasic reduced costs, the pivot tolerances RTPIVA and RTPIVR are tightened for future inversions, and an extra BTRAN is applied to the basic reduced costs to improve the dual variables and reduced costs in an iterative refinement fashion. The pivot tolerances are relaxed if the basic reduced costs are repeatedly small.

In the dynamic setting we must, of course, use the time staged inverse basis, i.e. the in-period $\boldsymbol{E}_{3t}$ factors, the inter-period $\boldsymbol{E}_{4t}$ factors, and the lagged $\boldsymbol{E}_{2t}$ factors. In the implementation we have integrated the ordinary BTRAN operation with the pricing operation or computation of the updated objective row. The effect is that all lagged Jacobian elements are accessed in one pass instead of requiring one pass of the basic elements for the $\boldsymbol{E}_{2t}$ multiplication and one pass of the nonbasic elements for the pricing. It is therefore not necessary in this routine to be able to access the lagged Jacobian by column; as mentioned in Section 5, this fact is reflected in our simple data structure.

The first stop test in CONOPT is based on the norm of the reduced gradient. The solution is considered optimal if all components are less than RTREDG or less than RTRGER times the largest intermediate result in BTRAN. The default of both is $\varepsilon_M^{0.4}$ where $\varepsilon_M$ is the relative machine precision. The test is only applied if the solution is accurate, however, i.e. if the tolerance on the objective as defined in Section 9.1 is minimal. If the reduced gradient is small but the solution is inaccurate, we perform a line search, and the accuracy will automatically be tightened as described in Section 9.4. The reduced gradient will be tested again at the more accurate point in the next iteration.

## 8. The superbasis and the search direction

The management of the superbasis, or the subspace optimization strategy, is fairly standard. At the first feasible point, all nonbasic variables away from their bounds are chosen as superbasic, and a variable will only leave the superbasis if it hits a bound or if it becomes basic, usually because it replaces a basic variable that hits a bound. During the iterations, nonbasic variables are allowed to enter the superbasis before the beginning of each line search, provided that their reduced gradient is sufficiently large. The mathematics are set out below.

For a given set of superbasic variables, the search direction for the superbasics can be computed using a conjugate-gradient, a quasi-Newton (variable metric), or a Newton method. Newton methods can be excluded immediately because the Hessian of the reduced objective function is too expensive to compute. Quasi-Newton methods generally give fewer iterations than conjugate-gradient methods, but their core requirements and updating time grows quadratically with the size of the superbasis compared to a linear growth for conjugate gradients. Some large-scale systems have therefore included both methods, and they switch from quasi-Newton to conjugate gradient when the number of superbasics exceeds a user-defined limit (see [23]).

In CONOPT we have decided that the extra overhead in a quasi-Newton method is always worthwhile. We have assumed that a problem with many superbasics will also have many constraints, so the costs of computing a reduced gradient and performing a line search will always be large compared to the cost of a quasi-Newton update. We have also assumed that CONOPT is not used on problems with more than 150 superbasics, so the extra core for the Hessian estimate does not become excessive compared to the core used for the code and for other problem-dependent arrays.

The quasi-Newton update used in CONOPT is the BFGS or the complementary DFP update, and the implementation is similar to that of MINOS [23]. Actually, the routines R1ADD, R1SUB, R1PROD, and DELCOL from MINOS are used with only minor changes, through the courtesy of M. Saunders. Only the changes to the MINOS implementation will be described. The reader is referred to [23] for a comprehensive description.

The basis selection algorithm has been designed to find as sparse a basis as possible. We have to pay a price for this extra sparsity: the set of basic variables can change by more than one variable even if only one variable hits a bound. If this happens, it becomes too difficult to update the Hessian estimate and we reinitialize it as the unit matrix.

If exactly one variable is replaced in the basis, we test whether the Hessian estimate contains any second order information, i.e. whether it has been changed from the initial unit matrix. If this is not the case, we simply keep the unit matrix. If second order information is available, we update the Hessian estimate in the old basic/superbasic space, project it into the new space, and delete the variable that hit a bound and caused the basis change. The projection and variable deletion are done as in MINOS. The update is slightly more complicated because of the non-linearities. The new reduced gradient needed in the update of the Hessian is only computed in the new basic/superbasic space using the new Jacobian. A projection back to the old space but with the new value of the Jacobian is therefore necessary. Fortunately, the projection of the reduced gradient and the projection of the Hessian estimate can be represented by the same updated row of the Jacobian, so there are essentially no extra costs.

When a nonbasic variable is made superbasic following the test below, the Hessian is augmented with one row and one column. The off-diagonal elements are set to zero, and the new diagonal element is set to RMEAN, the geometric mean of the other diagonal elements. This value is chosen instead of the usual 1 to get an augmentation that is asymptotically invariant to scalings of the objective function.

At the beginning of each line search we compute the reduced gradient, $\boldsymbol{g}_s$, the search direction for the old superbasics, $\boldsymbol{d}_s$, and the slope along this direction, $\mathrm{SLOPE} = \boldsymbol{d}_s^{\mathrm{T}}\boldsymbol{g}_s$. The nonbasic variable with the largest reduced gradient component pointing away from its current bound is then identified. If the reduced gradient, $g_{\max}$, passes the test

$$g_{\max}^2 \mathrm{LKSAME}^2\, \mathrm{RTINSB}/\mathrm{RMEAN} \geq \mathrm{abs}(\mathrm{SLOPE})$$

it is added to the superbasis, $\boldsymbol{d}_s$ and SLOPE are updated, and the search for a nonbasic variable to enter the superbasis is repeated. RTINSB is a tolerance with default value 4 and LKSAME is the number of consecutive times this nonbasic variable has been a candidate for the superbasis.

The subspace optimization strategy used in CONOPT lets variables enter and leave the superbasis in each iteration, and it can in theory cause cycling between subspaces, unlike more accurate subspace optimizations that can prevent cycling. However, CONOPT is designed for problems that may pass through hundreds of intermediate subspaces, so we cannot afford to do a strict optimization on each. The weak subspace strategy is therefore still used, despite its disadvantages. Fortunately, a recent paper by Dembo and Sahi [10], seems to indicate that weak subspace minimization can be protected against cycling. These results will be considered for inclusion in CONOPT in the near future.

The second stop test in CONOPT is based on SLOPE. SLOPE is twice the predicted change in objective based on the quasi-Newton formula. If this number is smaller than the minimal tolerance on the objective, we consider the solution optimal. As with the reduced gradient test, this test is only applied if the current solution is accurate.

## 9. The one-dimensional search

Once the search direction for the superbasic variables has been computed, we perform a one-dimensional search along this direction. In one of the early GRG papers, [2], Abadie suggested that the search be performed along the tangent to the constraint surface, and that feasibility be restored only from the final step. However, most later implementations attempt to return to the constraint surface at each step (see e.g. [20]). This approach makes it easier to handle models with linear objective functions and models that are so nonlinear that it is difficult to restore feasibility for large step lengths. CONOPT follows this approach and restores feasibility at each step.

During the one-dimensional search, the bounds on superbasic variables are in danger of being violated; during the iterations that restore feasibility, the bounds on basic variables can be violated. Several implementations, e.g. [2] and [20], allow the bounds to be temporarily violated, and bounds are only guaranteed to be satisfied in the final point of the line search. This means that the constraint functions may be called outside their domain of definition, however, and it becomes the user's responsibility to handle exceptions. This is contrary to the philosophy in CONOPT, that FCOMP never should be called with values that are outside the lower or upper bounds. Once a user has specified a set of bounds, he/she should never have to worry about them again.

As a consequence of the decision always to remain within bounds, the one-dimensional search in CONOPT is slightly different from one-dimensional searches in other GRG codes. In the following subsections we will describe the three main components of our one-dimensional search:

- the Newton algorithm that restores feasibility for given values of the superbasic variables and given estimates for the basic variables,
- the algorithm that computes initial estimates for the basic variables and manages the bounds on superbasic and basic variables, and
- the algorithm components that suggest step lengths and manage stop criteria.

### *9.1. The Newton algorithm – Static version*

For each step in the one-dimensional search we solve the subproblem

$$f(x_b, x_n) = b \tag{5}$$

with respect to $x_b$ for fixed $x_n$. Since (5) must be solved many times, the solution of (5) is the most expensive component of many GRG codes, and great care must be taken with it. It is important not to solve (5) more accurately than necessary and not to spend too much time trying to solve it if it is in fact not solvable.

To determine an appropriate degree of accuracy, it is helpful to see how the output is used. The output consists of the value of the objective function and the values of the basic variables. At all intermediate points we only use the objective function value to determine a new step length, and we only use the values of the basic variables to determine new initial values for the next Newton call. We cannot expect that doing one extra iteration now and making the basics more accurate will save us more than one iteration in the next Newton call, so our main stop criterion should be based on the error, df, in the objective function. df is the change that would occur in the objective function if the residuals were all reduced to zero, and it can be approximated by the change in objective function that one extra Newton iteration would give, i.e.

$$\mathrm{df} = \Delta x_j = \boldsymbol{e}_j^{\mathrm{T}} \Delta \boldsymbol{x}_b = \boldsymbol{e}_j^{\mathrm{T}} \boldsymbol{J}_b^{-1} \boldsymbol{r} = \boldsymbol{u}^{\mathrm{T}} \boldsymbol{r}$$

where $\boldsymbol{r}$ is the current vector of residuals and $\boldsymbol{u}$ is the vector of multipliers computed in step 4 of the GRG algorithm. Note that the df computation is very cheap.

The threshold for df should be large when the objective changes are large, and small when we approach the optimum and the objective changes are small. We therefore stop when abs(df) is less than RTOBJ = max(RTDFMN, edf $10^{-2}$), where RTDFMN = $(1+\mathrm{abs}(f_{\mathrm{obj}}))\varepsilon_M^{0.8}$ is an absolute lower bound, and edf is an estimate of the change in objective during the current line search, based on the slope of the objective at step length zero and a moving average of earlier objective changes. Apart from the RTDFMN term, the test is invariant to scalings of both the objective function and the constraints, and it performs a gradual tightening of the tolerances that allow inaccurate and cheap Newton solutions far from the final point, and guarantees accurate Newton solutions close to the optimum. We usually see 0, 1 or 2 Newton iterations before convergence, both in the initial phase where we make large steps and large residuals are compared with loose tolerances, and in the final phase where we make small steps and where small residuals are compared with tight tolerances.

The tests that stop the iterations if they are not likely to converge are based on the following principles. There is an overall iteration limit (default 10). Hitting this limit is expensive, however, so after each iteration we compute both the rate of decrease of the residuals and the number of iterations needed before convergence at the computed rate. If the expected number of remaining iterations is less than half the total number of remaining iterations, we continue; otherwise we stop without a solution. The purpose of the one-half value is to make the test gradually weaker, so that we do not stop without a solution after having invested several iterations, unless the rate of convergence deteriorates substantially. Experience shows that if the algorithm is not aborted by this test after the first iteration, it will almost always converge.

We can now describe the Newton algorithm used in CONOPT:

0. Set niter = 0.

1. Set niter = niter + 1. If niter > LFNWIT, return (failure). (Default LFNWIT = 10.)

2. Computer $\boldsymbol{r} = \boldsymbol{b} - \boldsymbol{f}(\boldsymbol{x}_b, \boldsymbol{x}_n)$. If niter > 1 only compute residuals of the nonlinear equations and set the linear residuals to zero.

3. Set ar = $\|\boldsymbol{r}\|_1$, df = $\boldsymbol{r}^{\mathrm{T}} \cdot \boldsymbol{u}$, adf = abs(df), and compute $p$ = (LKNWIT − niters + 1)/2.

4. If ar ≤ RTNEW, go to 6. (Default RTNEW = 0.001$*n$.)

5. If niter > 1 and ar(ar/arold)$^p$ > RTNEW then return (failure), else go to 8.

6. If adf ≤ RTOBJ, go to 12.

7. If niter > 1 and adf(adf/adfold)$^p$ > RTOBJ, return (failure).

8. Set arold = max(ar, RTNEW) and adfold = max(adf, RTOBJ).

9. Solve $\boldsymbol{J}_b \Delta \boldsymbol{x}_b = \boldsymbol{r}$ with respect to $\Delta \boldsymbol{x}_b$ using the same inverse basis as in the reduced gradient computation.

10. Set $\boldsymbol{x}_b := \boldsymbol{x}_b + \Delta \boldsymbol{x}_b$.

11. If a bound with value bnd is violated by more than (1 + abs(bnd))RTBND1, return (failure). If a bound is violated by less, set the basic variable to the bound value. Go to 1. (Default RTBND1 = $10^{-5}$.)

12. Set objective := $x_j$ + df and return (success).

Steps 2, 9 and 10 are standard GRG-Newton procedures, where the changes in basic variables are computed as the constraint residuals multiplied by the inverse basis from step length zero. Note that the residuals in all linear equations will be zero after the first iteration. Only nonlinear residuals are therefore computed in later iterations. Most of the other steps are included to handle bounds and to improve efficiency.

If the 1-norm of the residuals is large (default limit 0.001 $n$), we test that it converges fast (steps 4 and 5). If the residuals are small, we assume that the approximation of df is sufficiently accurate and we test that df converges fast (steps 6 and 7).

Step 11 is included to guarantee that the variables passed on to FCOMP always satisfy the bounds. If the point suggested by the Newton step is far outside one of the bounds, we expect that no solution exists within the bounds and we return to the step-length procedure in which a new and smaller step is chosen.

The Newton algorithm returns a status code, which defines whether Newton was successful or not. The code also distinguishes between the case where the initial point was already feasible and the case where it was not, and between failure due to slow convergence and failure due to violated bounds. Note, however, that too large a value of RTBND1 can transform a bound violation into a slow convergence and that a large bound violation may occur even when a solution exists. The status code must therefore be interpreted with caution by the line search routine. Experiments are currently underway to change RTBND1 dynamically, but a final choice has not yet been made.

## 9.2. *The Newton algorithm - Dynamic version*

When there are no inter-period pivots, we have a block-triangular system of equations and can therefore use the static Newton routine $T$ times, once with each of the $T$ in-period inverse bases. This fits well with the time orientation of the user's FCOMP subroutine.

When inter-period pivots link more time periods together, the equations from all time periods linked in one block must be solved as one simultaneous set of equations. To utilize the time structure of the inverse and of FCOMP, CONOPT breaks the iterations into an outer loop that runs over the periods in the block, from $t$begin to $t$end, and an inner loop that iterates within one time period. The main components of the procedure are as follows:

1. For $t =$ $t$begin to $t$end, execute the following in-period substeps:

1.1. Compute $\boldsymbol{r}$, the vector of residuals in period $t$.

1.2. If the residuals are small, go to next $t$. If the residuals are not converging, return (failure). The actual tests are the ones described in Section 9.1 above.

1.3. Compute $\Delta x_b$ assuming that all periods before $t$ are feasible:

$$\Delta \boldsymbol{x}_b = \boldsymbol{B}_{1\to t}^{-1}\begin{Bmatrix}\boldsymbol{0}\\ \boldsymbol{r}\end{Bmatrix} = \boldsymbol{E}_{4t}^{-1}\boldsymbol{E}_{3t}^{-1}\boldsymbol{E}_{2t}^{-1}\boldsymbol{E}_{1t}^{-1}\begin{Bmatrix}\boldsymbol{0}\\ \boldsymbol{r}\end{Bmatrix} = \boldsymbol{E}_{4t}^{-1}\boldsymbol{E}_{3t}^{-1}\begin{Bmatrix}\boldsymbol{0}\\ \boldsymbol{r}\end{Bmatrix} = \boldsymbol{E}_{4t}^{-1}\begin{Bmatrix}\boldsymbol{0}\\ \boldsymbol{B}_{t\to t}^{-1}\boldsymbol{r}\end{Bmatrix}$$

i.e. first apply the in-period inverse to the residuals and then apply the inter-period alpha-vectors with pivot in period $t$. Note that $\Delta x_b$ can contain nonzeros between period $t$begin and $t$.

1.4. Set $x_b := x_b + \Delta x_b$.

1.5. If a bound is more violated than RTBND1, return (failure). If a bound is violated by less than RTBND1, move the variable back to the bound. Go to 1.1.

2. If no changes were made to lagged values in step 1.4 in any period i.e. if no inter-period alpha-vectors were used, then return (success).

3. If the lagged changes are diverging, return (failure), otherwise go to step 1.

The routine makes one period feasible before it considers the next. During iterations in later periods it assumes (but does not check) that the earlier periods remain feasible. This is not an unreasonable assumption since the only changes in variables in earlier periods are those derived from the inter-period alpha-vectors, so new infeasibilities will be second order terms. The advantage of the assumption is that the routine only requires residuals for one period at a time, which is the way FCOMP produces them, and that the lagged Jacobian in the factor $\boldsymbol{E}_{2t}$, is not needed directly. The linear part of the lagged Jacobian is, of course, used indirectly to compute the residual. After one pass of all time periods we must check that the residuals in earlier periods have remained small and, if necessary, make another round of adjustments. It is our experience that one round of adjustments is usually enough. The second round is only needed to test the final residuals.

## 9.3. *Bounds on basic and superbasic variables and initial values for Newton's algorithm*

The superbasic variables move along the search direction $\boldsymbol{d}_s$ in a straight line. This makes it easy to determine once and for all the step length at which the first superbasic hits a bound. This step length, $\theta^s_{\max}$, is an absolute upper bound on the step length.

The basic variables move along a curve that is implicitly defined through the Newton algorithm. The upper bound on the step length from bounds on basic variables, $\theta^b_{\max}$, is therefore only defined implicitly through failures of the Newton algorithm due to bound violations for steps greater than $\theta^b_{\max}$. To avoid frequent Newton failures due to bound violations, $\theta^b_{\max}$ is estimated on the basis of extrapolations of the values of the basic variables. The extrapolations (and similar interpolations) are, of course, also used as initial values for the Newton algorithm to speed it up.

For the first step, CONOPT uses a linear extrapolation for the basics based on the base point and the tangent, $\boldsymbol{d}_b$, in the base point computed from

$$\boldsymbol{d}_b = -\boldsymbol{J}_b^{-1}(\boldsymbol{J}_s\boldsymbol{d}_s).$$

In the static case, the computation relies on a standard FTRAN routine as in LP. In the dynamic case, we have, as in BTRAN, integrated the computations of the lagged parts of $\boldsymbol{J}_s \cdot \boldsymbol{d}_s$ with the multiplication using the $\boldsymbol{E}_{2t}$ factor; all lagged Jacobian elements are thus again accessed in one pass, independent of the split of variables in basic and nonbasic.

After a feasible point is found with some positive step length, we fit a quadratic model, first through the base point, the new point, and the tangent, and later through three points. CONOPT keeps three $m$-vectors for this purpose. The space for two of the intermediate vectors is re-used as scratch storage by the inversion routine as mentioned in Section 6.

During the interpolation or extrapolation processes, one of the old vectors is always overwritten by the initial values for the new point. If Newton fails, we are therefore left with only two feasible points or with the base point and the base tangent, and the next interpolation or extrapolation must use an inferior linear model. Because of this loss of a point each time Newton fails, we approach a bound in a rather conservative way: before testing whether a basic variable will hit a bound, we multiply the distance to the bound, abs(bnd $- x$), by a 'safety factor'. Currently CONOPT uses max(0.8, $1-$abs(bnd $- x$)/($1+$abs(bnd))). When the basic variable is far from its bound, the extrapolation will be uncertain and the safety factor is small. As the basic variable moves closer to its bound, the extrapolation will be shorter and the safety factor will be closer to 1. If the previous Newton call accepted the extrapolated point as immediately feasible, we set the safety factor to 1, and the next extrapolation will therefore produce a point with the critical basic variable exactly at its bound.

The safety procedure gives few Newton failures, but it may require 4 or 5 step length increases before the basic variable ends at its bound if it started far away.

Fortunately, the last small increases in step length are cheap, since the extrapolated values for the basic variables are good; the number of Newton iterations is therefore small.

During the extrapolations, extra precautions must be taken with a degenerate basis. If a basic variable remains very close to a bound throughout the line search, it can create false upper bounds on the step length. We have eliminated this problem by disregarding basic variables whose extrapolation intersects the bound at a very flat angle from the bound tests. The tests are similar in spirit to the CHUZR-tests in the DEVEX LP-code [16].

## *9.4. Step-length determination, stop criteria, and exception handling*

A one-dimensional search procedure for a GRG code is more complicated than a one-dimensional search procedure for an unconstrained optimization code, because the former must handle some problems that do not occur in an unconstrained model. It is likely that function values are not available for large steps because Newton's algorithm fails, and we can find that the error in the function values caused by inaccurate Newton solutions is comparable to the changes in function value for small steps. After a short treatment of the general stop criteria, we describe in some detail how CONOPT handles these two problem areas.

The initial step length is computed as the expected change in objective function divided by the slope at step zero. If this step length is close to 1 - the optimal Newton step, it is changed to 1. After at least one function value has been found, a new step length is computed based on a quadratic model, subject to the constraint that the step length cannot increase by more than a factor ALPHA. The default value of ALPHA is initially 4, but it may decrease as mentioned below. Bounds on variables are incorporated as described in the previous subsection. The search is stopped if the expected improvement in objective function from the quadratic model is less than RTONED (default 0.2) times the improvement so far. In cases where the step is defined by a bound on a variable, however, we do not stop unless the variable is within (1 + abs(bnd)) RTBND1 of the bound, independent of the expected improvement in objective function.

Newton failures can occur at several points in a one-dimensional search. If Newton fails before a first improved objective function value has been found, we make the step smaller. If the failure was caused by a bound, we multiply the step by a factor 0.9; and if the convergence was slow, we multiply it by a factor 1/ALPHA. Newton can also fail in an interpolation step after an improved point has been found, although this is unlikely. In this case we stop the search, reinvert, and compute a new search direction.

The last case of a Newton failure is after an extrapolation step where an improved point has already been found. If the failure was due to a bound, we assume that the step exceeded but was close to $\theta^b_{\max}$; we cut the step increase by a factor 0.9, and call Newton again without any improvement tests. If the failure was due to slow convergence, we replace the step by the geometric mean of the previous step and the largest feasible step, replace the step-length multiplier ALPHA by its square root (for the duration of this line search only), and again apply the improvement criterion mentioned above before the next Newton call. If ALPHA has already been decreased twice, we define the one-dimensional search as badly behaved and stop. Many GRG codes stop the one-dimensional search after the first Newton failure in an extrapolation (see e.g. [20]), but we have found that the smaller value of the step length multiplier ALPHA, combined with the extrapolation of basic variables, will often let us increase the step length considerably beyond the point where Newton first failed. And although the step-length multiplier is smaller, it is applied to a larger step, so the objective improvement can be considerable.

Inaccurate objective function values can cause problems for small step lengths. New step lengths are computed from differences in objective function values and the computed step lengths are therefore useless when the differences in objective functions are comparable in magnitude to the errors in these objective functions. We therefore keep track of df, the error estimate computed in the Newton algorithm, for each feasible point. Before a one-dimensional search is started, we test whether the error estimate for the base point is less than the error tolerance for the coming one-dimensional search. If it is not, we call Newton at the base point before starting the one-dimensional search. This precaution should minimize the problem of badly behaved one-dimensional searches. During the search, we impose a lower bound on the step corresponding to the step where the expected decrease in objective is equal to the error tolerance. A smaller step is only allowed if a variable previously hits a bound. Whenever an interpolation suggests a smaller step, the one-dimensional search is stopped and the following recovery sequence is initiated:

A: The expected change in objective (edf) and the Newton tolerances on residuals (RTNEW) and objective (RTOBJ) are divided by 10, and Newton is called to make the base point more accurate before the one-dimensional search is called again with the old direction vectors and old inverse basis.

B: The improved base point is usually enough to get the optimization back on track, but if the next one-dimensional search also fails, we recompute the Jacobian, reinvert in the new base point, compute new direction vectors, and call the one-dimensional search again.

C: If this still is not enough, we switch to the steepest descent direction.

D: Increase the superbasis if there are any nonbasic candidates, and

E: Invert with search for a new basis with larger pivot tolerances.

F: A to E are repeated until a line search gives an improvement or until all tolerance are minimal, in which case CONOPT declares that it cannot solve the model. We should insert a routine here that tests whether the noise level in FCOMP is very high, or first and/or second derivatives are unreasonable. The best format for such tests is not yet clear, however.

While tightening the tolerances in A above, we may find that Newton's algorithm fails because a degenerate basic variable exceeds a bound. We cannot simply rely on the one-dimensional search algorithm to cut the step length as the ordinary Newton algorithm does, so CONOPT contains a more elaborate Newton routine with Newton steps less than one and basis changes for this purpose. It is based on the phase-1 algorithm in the next section.

## 10. A phase-1 algorithm

Most papers on GRG codes describe their phase-1 algorithm very superficially if at all, but our general impression is that most codes minimize a sum of absolute or squared residuals, using the standard GRG procedure and starting from an all-logical basis. This approach is easy to implement and it is as reliable as the underlying GRG code. Unfortunately, however, this phase-1 will need at least $k$ line searches, where $k$ is the number of structural variables in the first feasible basis, independent of the initial values of the structural variables.

By comparison, Newton's algorithm can find a feasible point very quickly if good initial values are provided and a good basis is chosen. Based on these observations, we have implemented the following algorithm:

0. Choose an initial point $\boldsymbol{x}$.

1. Compute the Jacobian, $\boldsymbol{J}$, and select and invert a basis. The usual criteria of basic variables away from bounds and good conditioning applies. Define the basis as new.

2. Compute the Newton direction for the basic variables, $\Delta x_b = -\boldsymbol{J}_b^{-1}(\boldsymbol{f}(\boldsymbol{x}) - \boldsymbol{b})$.

3. Find the step length $\alpha$ at which the Newton direction hits a bound.

4. If $\alpha < 1$ go to 5. Otherwise take a full Newton step, $\boldsymbol{x}_b^+ = \boldsymbol{x}_b + \boldsymbol{\Delta x}_b$, $\boldsymbol{x}_n^+ = \boldsymbol{x}_n$. If the point is feasible, $|\boldsymbol{f}(\boldsymbol{x}^+) - \boldsymbol{b}|_1 \leq$ RTNEW, return (success). If the residuals did not decrease fast, go to 8. Otherwise set $\boldsymbol{x} := \boldsymbol{x}^+$, define the basis as old, and go to 2 where the old inverse basis is used again.

5. Take a step length of $\alpha$ in the Newton direction, $\boldsymbol{x}_b^+ = \boldsymbol{x}_b + \alpha \Delta \boldsymbol{x}_b$, $\boldsymbol{x}_n^+ = \boldsymbol{x}_n$. If the residuals did not decrease fast, go to 8, otherwise set $\boldsymbol{x} := \boldsymbol{x}^+$. A critical basic variable, $j$, is now at a bound.

6. Compute the Jacobian, reinvert the old basis in the new point, and compute a new Newton step. If the previous critical variable is no longer critical, go to 3, otherwise perform a basis change where $j$ leaves the basis. The incoming basic variable is selected as follows: Compute the updated row in which variable $j$ has its pivot, i.e. the row of potential pivot elements $\boldsymbol{c}^{\mathrm{T}} = (\boldsymbol{e}^{\mathrm{T}}\boldsymbol{J}_b^{-1})\boldsymbol{J} = \boldsymbol{\pi}^{\mathrm{T}}\boldsymbol{J}$ where $\boldsymbol{e}$ is a unit vector that picks out the proper row. Choose a column $p$ to enter the basis. It must satisfy:

a. abs$(c_p) >$ RTPIVA i.e. an absolute pivot tolerance,

b. $\Delta x_p = \Delta x_j / c_p \geq 0$ if $x_p$ is at its lower bound and $\Delta x_p \leq 0$ if $x_p$ is at its upper bound. If there is more than one $p$-candidate, choose one that maximizes the step $\alpha_p$ for the incoming variable, namely $\alpha_p = (\mathrm{bnd}_p - x_p)/\Delta x_p$ if max $\alpha_p < 1$. If max $\alpha_p \geq 1$, maximize the pivot element abs$(c_p)$ among columns with $\alpha_p \geq 1$. If a $p$ is found, replace column $j$ by $p$ in the basis, define the basis as new and go to 2.

7. If no $p$ is found in 6, we declare the solution infeasible and return (failure). The weighted sum of infeasibilities $\text{abs}(\boldsymbol{\pi}^{\mathrm{T}}(\boldsymbol{f}(\boldsymbol{x})-\boldsymbol{b}))$ has the value $\text{abs}(\Delta x_j)>0$, and it is at a local minimum or at a stationary point.

8. Convergence was slow. If the basis was old, go to 1. If the basis was new, the slow convergence can only be due to second order terms. Decrease $\alpha$ until a sufficiently fast decrease in residuals is found or until $\alpha<\alpha_{\min}$ (default $10^{-7}$). Set $\boldsymbol{x}_b := \boldsymbol{x}_b + \alpha \Delta \boldsymbol{x}_b$. If $\alpha$ is small (default limit is 0.01) for two consecutive iterations, we make a heuristic basis change where the basic variable with maximum $\text{abs}(\Delta x_b)$ leaves the basis; alternatively, we reinvert the same basis. In either case, we then define the basis as new, and go to 2.

The algorithm tries to make Newton steps, and it can only be stopped by two things: a bound on a basic variable or slow convergence due to large second order terms. A bound on a basic variable leads to a basis change in step 6. The incoming variable is chosen so that it will not hit a bound immediately and, if possible, so that it has a good step bound and a good pivot element. This should improve the next iterations. An anti-degeneracy measure could be added to prevent cycling if $\alpha=0$ repeatedly, but it has not yet been necessary. An incoming basic variable can always be found unless a certain weighted sum of infeasibilities (see step 7) is at a local minimum or at a stationary point. In the linear or convex case this would prove infeasibility, even if the sum of infeasibilities is not minimum, so we use it as an indication that no feasible solution exists.

Slow convergence due to large second order terms, handled in step 8, is a more serious problem. CONOPT currently contains a heuristic that tries to get around the problem by choosing a new basis. The basic variable that changes most, i.e. the variable that is likely to cause the largest second order terms, is removed from the basis and the criterion in step 6 is used to choose the entering variable. A better approach would be to incorporate a steepest-descent step, as in Powell's hybrid algorithm [28], or to switch to a standard phase-1 algorithm. These additions have high priority and will be implemented soon.

The phase-1 algorithm described above will usually find a feasible solution after very few iterations when good initial values are available. A few basis changes may be needed. If only some variables have been initialized with good values and the remainder have been initialized by default at a bound, more iterations are needed. The first bases will have many basic variables at the bound and it is likely that $\alpha$ will be zero. After some iterations with small or zero steps, however, the Newton direction will usually be directed into the feasible space and the iterates will move very quickly to a solution.

Some models do fail with this phase-1 algorithm - notably ones containing terms that are very nonlinear close to the bounds, such as $\log(x)$, $x \geqslant 10^{-7}$. We have often solved these models in practice by changing the bounds to exclude the worst nonlinearities, but an automatic solution must also of course be implemented.

In dynamic problems, CONOPT searches for a feasible solution one period at a time. In the first period it uses the static procedure described above. In later periods CONOPT tests two initial points. The first is one supplied by the user or defined by default, and the second is an extrapolation from earlier periods. The point with the smallest sum of residuals is used to start the static phase-1 procedure. During basis changes, we try to choose the incoming basic variable from the current period, so that other periods will remain unchanged and we can continue to work on the current period only. Sometimes, however, there is no incoming variable with a good pivot element in the current period. In this case, CONOPT tries to introduce a variable from an earlier period in the basis, and in the next iterations it continues to solve the larger block of periods with the static phase 1 procedure. If a feasible solution cannot be found in one period, there is no point in continuing to later periods, so CONOPT declares the problem infeasible and stops.

We have found that this approach is very fast for problems with constant or gradually changing bounds. The extrapolation from earlier periods usually generates a reasonably good initial point with many variables between bounds, and the static phase-1 procedure finds a feasible solution after only a few iterations, often without any basis changes. When new bounds become active in a period, however, e.g., when terminal conditions are imposed in the last period, extra work is needed to obtain feasibility.

The algorithm can be thought of as based on the homotopy

$$\boldsymbol{h}(\boldsymbol{x}, \theta) = \boldsymbol{f}(\boldsymbol{x}) - (\boldsymbol{b}\theta + \boldsymbol{f}(\boldsymbol{x}_0)(1-\theta)) = 0$$

where $\boldsymbol{x}_0$ is the initial (or current) point, and $\theta$ is the homotopy parameter. The algorithm approximates a path for $\boldsymbol{x}$ as a function of $\theta$ from $\boldsymbol{x}(0) = \boldsymbol{x}_0$ to a solution $\boldsymbol{x}(1)$. Since we are not interested in the path itself, we restart the path each time a better point has been found. [14] contains a general discussion of this type of path-following algorithm.

## 11. Computational experiments

CONOPT has been used extensively for several years in the World Bank's research work. During the first 6 months of 1983 it was called more than 2000 times, mainly in connection with research and development work on economic models. Models change frequently because of the research environment; many of them have data errors or inconsistencies, others are infeasible, and still others are badly scaled. A general-purpose system should be judged by its ability to work efficiently in all these different cases. CONOPT has not yet reached this point; model builders typically learn to adjust to CONOPT and models developed at the late stages of a project consequently benefit from this learning process and solve much more successfully than early ones.

This section will describe our computational experiments with a few models. The models presented here are mainly late-stage, well established models whose authors have been through the learning process. Some of the models are only solved for a single time horizon or a few different ones - the situation usually found in practice.

On the other hand some models for which data are available are solved for several different time horizons to show how solution times and core requirements depend on size. The models have been chosen to demonstrate how different characteristics of a model influence performance.

All experiments were performed on a CDC Cyber 176 using the FTN compiler with OPT = 2. This computer has an address space of 131 000 words of 60 bits. The code, including the user's FCOMP subroutine, uses approximately 51 000 words, leaving 80 000 words available for working storage. In the version used for most of the experiments reported here CONOPT packs 4 integers or 32 logicals into one word. The latest version does not use this packing; it is a little faster and the code is 3000 words smaller, but it uses more working storage. The times reported are total execution time including reading input, hashing variable and constraint names, setting up core allocation, and printing the MPS-type output file, but excluding time to compile FCOMP and load the system. The total compile and load time is from 1.0 to 1.2 CP-second, relatively independent of the size of FCOMP.

### *11.1. The OPEC model*

This is a model that describes optimal pricing and extraction of a limited resource for the OPEC cartel. It is described in [27]. The model has 5 equality constraints and 6 structural variables per period, plus a nonbinding constraint and an associated slack variable that together represent the objective function component for the period. More detailed characteristics of this and the following models are shown in Table 1. The solution characteristics for 8 different time horizons are shown in Table 2.

### *11.2. The Manne model*

This is the model described in the MINOS/AUGMENTED paper [25]. It has an inequality that only applies to the terminal period. This is implemented in CONOPT as an inequality with a time dependent right-hand side; the right-hand side is very large in all but the last time period.

The model has been tested in 8 versions (see Table 3) with varying amounts of bounds and with the constraints as equalities or inequalities. The results are shown in Table 4. The first conclusion is that the equality constrained versions, 1 to 4, are much faster than the corresponding inequalitity constrained versions, 5 to 8. Phase 1 is slower with the equalities, but the number of line searches is smaller once a feasible solution is found. The reason is probably the reduced dimension of the search space. However, the first feasible point generated by phase 1 seems to be a more important but rather uncontrollable factor. The fast versions, 2 and 4, both have 98 superbasics and an objective value of 8.99 in the first feasible point. In each of these versions, one superbasic variable hits a bound and is removed from the superbasis in each of the first 86 line searches. The last 19 line searches required for version 2 and the last 17 required for version 4 increase the superbasis again

Table 1

Problem characteristics for the test models

| *Statistics per period* | | OPEC | MANNE/1-4 | MANNE/5-8 | Coffee | Chemical process | Indonesia |
|---|---|---|---|---|---|---|---|
| Constraints incl. obj. | - linear | 4 | 2 | 2 | 11 | 11 | 40 |
| | - nonlinear | 2 | 2 | 2 | 23 | 13 | 74 |
| | - total | 6 | 4 | 4 | 34 | 24 | 114 |
| Variables incl. slacks | - linear | 3 | 3 | 5 | 16 | 4 | 25 |
| | - nonlinear | 4 | 2 | 2 | 19 | 23 | 107 |
| | - total | 7 | 5 | 7 | 35 | 27 | 132 |
| Unlagged Jacobi elements | - constant | 10 | 7 | 9 | 56 | 33 | 248 |
| | - *time series* | 0 | 0 | 0 | 0 | 3 | 20 |
| | - variable | 5 | 2 | 2 | 21 | 61 | 293 |
| | - total | 15 | 9 | 11 | 77 | 97 | 561 |
| Lagged Jacobi elements | - constant | 4 | 2 | 2 | 16 | 7 | 7 |
| | - time series | 0 | 0 | 0 | 0 | 0 | 0 |
| | - variable | 0 | 0 | 0 | 20 | 4 | 92 |
| | - total | 4 | 2 | 2 | 36 | 11 | 99 |
| Max lag | | 1 | 1 | 1 | 7 | 1 | 1 |

Table 2

Solution characteristics for the OPEC model

| Time periods | Variables | Constraints | Phase I iterations | Line searches | | Basis changes | CP-Sec[a] | Working storage | Superbasis | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | Constrained | Unconstrained | | | | Min | Max | Final |
| 5 | 35 | 30 | 5 | 0 | 8 | 0 | 0.31 | 1 024 | 5 | 5 | 5 |
| 10 | 70 | 60 | 10 | 0 | 11 | 0 | 0.50 | 1 024 | 10 | 10 | 10 |
| 20 | 140 | 120 | 20 | 0 | 14 | 0 | 0.99 | 1 536 | 20 | 20 | 20 |
| 30 | 210 | 180 | 30 | 0 | 19 | 0 | 1.60 | 2 560 | 30 | 30 | 30 |
| 50 | 350 | 300 | 52 | 0 | 65 | 0 | 6.66 | 4 096 | 42 | 50 | 50 |
| 75 | 525 | 450 | 77 | 8 | 79 | 0 | 11.71 | 7 168 | 42 | 75 | 75 |
| 100 | 700 | 600 | 102 | 19 | 99 | 0 | 19.12 | 10 752 | 42 | 100 | 100 |
| 200 | 1400 | 1200 | 202 | 2 | 243 | 0 | 76.55 | 30 720 | 42 | 200 | 200 |

[a]CDC Cyber 176.

Table 3

Characteristics of the 8 versions of the Manne model

| Version | Equality/ Inequality | Upper bound on $I$ | Lower bound on $C$ |
|---|---|---|---|
| 1 | eq | no | 0.95 |
| 2 | eq | yes | 0.95 |
| 3 | eq | no | 0.01 |
| 4 | eq | yes | 0.01 |
| 5 | ineq | no | 0.95 |
| 6 | ineq | yes | 0.95 |
| 7 | ineq | no | 0.01 |
| 8 | ineq | yes | 0.01 |

and establish optimality. Both slow versions, 1 and 3, start from a vertex with a much smaller objective value of $-0.97$. The search is more irregular; variables, especially basic variables, keep hitting bounds that are not binding in the optimal solution. CONOPT must consequently spend extra time performing basis changes and search mode inversions, and in building up second order information that is lost during changes in the superbasis.

As mentioned earlier, CONOPT has been designed for problems where most of the line searches have their optima at interior points, i.e. where the step lengths are determined from nonlinearities and not from bound values. This assumption is clearly not satisfied for this model, which is why the computing times reported here (adjusted for differences in machine speed) are 5 to 10 times those reported for MINOS/AUGMENTED. Nevertheless, the model seems to be useful for testing different subspace minimization strategies since the bounds are so important.

### *11.3. The world coffee model*

The coffee model is an econometric model of world coffee demand and production. It has been used by T. Cauchois to evaluate the viability of different cartels in the coffee market [6].

Some results obtained with the model are given in Table 5. The model is rather badly scaled, producing problems with the optimality tolerances for models with more than 2000 constraints.

### *11.4. Chemical process model*

This model, contributed by J. Bisschop, describes a chemical reaction. The objective is to minimize the total process time, subject to bounds on the final concentrations and the final volume. The control variables are material inflows and heat exchanges. The process is modeled with 19 time steps, so as to meet CONOPT's requirement of a fixed number of time periods; the length of each time step is made a variable in the model. The length of the time step in each period (except the first) is made equal to the lagged time step, so that they all become equal.

Table 4

Solution characteristics for the 8 versions of the Manne model

| Version | Variables | Constraints | Phase I iterations | Line searches | | Basis changes | CP-Sec[a] | Working storage | Superbasis | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | Constrained | Unconstrained | | | | Min | Max | Final |
| 1 | 500 | 400 | 100 | 160 | 281 | 128 | 58.6 | 8 704 | 40 | 98 | 98 |
| 2 | 500 | 400 | 43 | 86 | 19 | 0 | 9.2 | 8 704 | 14 | 98 | 24 |
| 3 | 500 | 400 | 100 | 96 | 574 | 88 | 103.3 | 8 704 | 40 | 99 | 99 |
| 4 | 500 | 400 | 43 | 86 | 17 | 0 | 9.1 | 8 704 | 14 | 98 | 24 |
| 5 | 700 | 400 | 2 | 302 | 392 | 117 | 94.1 | 10 240 | 40 | 98 | 98 |
| 6 | 700 | 400 | 2 | 296 | 25 | 137 | 41.1 | 6 144 | 40 | 24 | 24 |
| 7 | 700 | 400 | 2 | 261 | 617 | 212 | 129.6 | 10 240 | 40 | 99 | 99 |
| 8 | 700 | 400 | 2 | 301 | 46 | 237 | 54.4 | 7 680 | 40 | 67 | 24 |

[a] CDC Cyber 176.

Table 5
Solution characteristics for the coffee model

| Time periods | Variables | Constraints | Phase I iterations | Line searches |  | Basis-changes | CP-Sec[a] | Working storage | Superbasis |  |  |
|---|---|---|---|---|---|---|---|---|---|---|---|
|  |  |  |  | Constrained | Unconstrained |  |  |  | Min | Max | Final |
| 10 | 350 | 340 | 23 | 8 | 33 | 6 | 9.0 | 4 096 | 3 | 10 | 3 |
| 20 | 700 | 680 | 33 | 10 | 61 | 6 | 27.7 | 7 168 | 11 | 20 | 11 |
| 30 | 1050 | 1020 | 43 | 10 | 50 | 8 | 37.6 | 10 240 | 20 | 30 | 20 |
| 40 | 1400 | 1360 | 53 | 9 | 46 | 9 | 53.4 | 12 800 | 26 | 33 | 27 |
| 50 | 1750 | 1700 | 63 | 11 | 103 | 9 | 125.3 | 15 872 | 28 | 34 | 28 |
| 60 | 2100 | 2040 | 73 | 12 | 116 | 12 | 183.2 | 18 944 | 29 | 34 | 30 |
| 80 | 2800 | 2720 | 93 | 13 | 152 | 13 | 334.7 | 25 088 | 32 | 37 | 37 |

[a] CDC Cyber 176.

The overall model has 456 constraints and 513 variables. It begins from an almost feasible point and requires 21 phase-1 iterations to become feasible. After 10 line searches, problems arise with a degenerate point, the tolerances are tightened, and a new phase 1 with 22 iterations is performed. It takes 90 more line searches to become optimal. 25 of the 100 line searches are stopped by a bound on a variable, but there are basis changes in 31 iterations. The last 6 basis changes are caused by small pivot elements, which are in turn caused by poor scaling. The superbasis changes from 36 variables in the first feasible point to 16 in the optimal solution, and the overall optimization requires 85.8 CP-sec and 8704 words of working storage.

### *11.5. The Indonesia model*

This model is currently under development by A. Gelb, see [15]. It describes in detail production, investment, capital accumulation, income generation, savings and consumption, and imports and exports in a 6 sector model. The model has 18 nonbasics per period but is run in many different versions, usually with many fixed variables leaving only 1 or 2 superbasics per period. The number of periods is usually 10, so the size of the overall model is 1140 constraints, 1320 variables, and 10 to 20 superbasics.

Each time a new version of the model is developed, all nonbasics are fixed and a new base trajectory is computed. An accurate solution is usually found in 15 to 30 phase-1 iterations and 3 to 5 CP seconds. This trajectory is then used for several optimization runs. A sample of 11 optimizations required between 10 and 60 line searches, with an average of 28. A variable hit a bound in 30% of all line searches. The solution times were between 9.5 and 63 CP seconds, with an average of 32; the final number of superbasic variables were between 1 and 11, with an average of 4.6.

### *11.6. A static-dynamic comparison*

The previous models have been large because they had many time periods. The model with the largest number of constraints per period, the Indonesia model, had only 114 constraints per period. To test the capabilities of CONOPT on large static models we solved the OPEC model from Section 11.1 as if it had been a static model. In the first experiments we started both the static and the dynamic model from the same infeasible point. The first feasible points turned out to be quite different and the characteristics of the solution path seemed to be rather sensitive to these first feasible points, exactly as we saw in Section 11.2. The following 3-step procedure was therefore used to create results that were comparable:

1. CONOPT was first applied to the dynamic model to find a feasible solution. This solution was written to a file in CONOPT input format using the WRITE-verb. The USER-verb was then called. The USER subroutine contained a matrix generator that created the MODEL file for the static model including an INITIAL bound set with the feasible solution.

2. CONOPT was called to solve the dynamic model starting from the feasible solution generated in the first step.

3. CONOPT was finally called to solve the static model from the same feasible solution.

Table 6 shows the results from the last two calls of CONOPT. The latest version of CONOPT (without packing) was used, so the working storage numbers are larger than in Table 1. It was not possible to run the static version of the 200 period model due to core storage limitations. The static 175 period model was also close to the limit; only 6000 words were left. The results seem to indicate that there can be considerable gains from exploiting the dynamic structure of large models. They also show, however, that the gains are insignificant for 'small' models, i.e. models with less than 100–200 constraints.

## 12. Conclusions

It has been known for some time that sparse matrix techniques used in LP with minor modifications could be used to build large-scale GRG codes. This paper has described one such code, CONOPT, and has indicated several areas where it has been useful to modify LP-techniques to deal with the characteristics of a nonlinear problem. A new phase-1 algorithm for almost-feasible problems has also been described.

The paper has mainly considered large static problems but it has also described techniques for handling dynamic problems with many bounds. The key features here are the inversion routine and the associated Newton routine.

Thus far, CONOPT has been successfully used to solve static models with over 1000 constraints and variables and dynamic models with over 2000 constraints and variables; by doing so, it has proved the viability of large-scale GRG codes, particularly on problems involving almost as many constraints as variables. There are still many research areas to pursue, however, the most important of which seem to be:

(1) Subspace minimization strategies or strategies for releasing variables from their bounds;

(2) Automatic scaling. Although (4) contains rather disappointing results about automatic scaling, we hope that dynamic problems will be easier to scale because better and more stable scale factors can be derived by averaging over all time periods. Other user-defined groupings may also be useful in deriving good average scale factors;

(3) Dynamic setting of parameters and tolerances. Many parameters should have one value for almost linear problems and another for very nonlinear problems. The values of these parameters should be adjusted as the optimization progresses and more information is gained on the problem characteristics;

Table 6

Comparison of static and dynamic implementation of the same model

| | Static | Dynamic | Static | Dynamic | Static | Dynamic | Static | Dynamic |
|---|---|---|---|---|---|---|---|---|
| Periods | 20 | | 50 | | 100 | | 175 | |
| Constraints | 121 | 120 | 301 | 300 | 601 | 600 | 1051 | 1050 |
| Variables | 141 | 140 | 351 | 350 | 701 | 700 | 1226 | 1225 |
| Line searches | 14 | 14 | 64 | 65 | 102 | 118 | 215 | 236 |
| Steps | 29 | 31 | 138 | 138 | 187 | 230 | 413 | 444 |
| Steps/line search | 2.07 | 2.21 | 2.16 | 2.12 | 1.83 | 1.95 | 1.92 | 1.88 |
| FCOMP calls[a] | 51 | 53.1 | 247 | 221.9 | 366 | 300.8 | 804 | 521.4 |
| FCOMP/step | 1.76 | 1.71 | 1.79 | 1.61 | 1.96 | 1.31 | 1.95 | 1.17 |
| Ave. Newton iterations[a] | 0.76 | 0.71 | 0.79 | 0.61 | 0.96 | 0.31 | 0.95 | 0.17 |
| Optimization time | 0.59 | 0.56 | 8.29 | 5.84 | 42.28 | 17.66 | 271.31 | 59.15 |
| Total time | 1.33 | 0.82 | 10.52 | 6.31 | 48.63 | 18.23 | 289.18 | 60.59 |
| Working storage | 7168 | 3072 | 18 432 | 6144 | 39 936 | 14 336 | 75 776 | 31 744 |
| Ratios—Static/Dynamic: | | | | | | | | |
| Optimization time | 1.05 | | 1.42 | | 2.39 | | 4.59 | |
| Total time | 1.62 | | 1.67 | | 2.63 | | 4.77 | |
| Working storage | 2.33 | | 3.00 | | 2.79 | | 2.39 | |
| FCOMP calls | 0.96 | | 1.11 | | 1.22 | | 1.54 | |

[a]The number of FCOMP calls and the average number of Newton iterations has for the dynamic model been divided by the number of periods to get comparable numbers. The numbers do not include FCOMP calls for the Jacobian evaluation.

(4) Comparisons with other large-scale codes such as MINOS 5.0 [26], sequential linear programming, and sequential quadratic programming codes. We hope that the advances of high-level modeling systems will make this task easier. So far, a link has been built between the General Algebraic Modeling System GAMS [5] and CONOPT. Once links to other algorithms are developed, comparisons will become very easy, though not necessarily cheap. The ultimate goal in this area is to be able to predict from within a modeling system which code will be best suited for a particular problem, and to choose that code automatically.

## 13. Acknowledgement

I would like to thank two anonymous referees for their careful reading of earlier versions of the paper and for their many constructive comments. I would also like to thank Alex Meeraus for encouragement and helpful discussions during the many years it took to develop CONOPT.

## References

[1] J. Abadie, "Optimization problems with coupled blocks", *Economic Cybernetics Studies and Research* (1970b).

[2] J. Abadie, "Application of the GRG algorithm to optimal control problems", in: J. Abadie, ed., *Nonlinear and integer programming* (North-Holland, Amsterdam, 1972) pp. 191-211.

[3] J. Abadie and J. Carpentier, "Generalization of the Wolfe reduced gradient method to the case of nonlinear constraints", in: R. Fletcher, ed., *Optimization* (Academic Press, New York, 1969) pp. 37-47.

[4] P.O. Beck and L.S. Lasdon, "Scaling nonlinear programs", *Operations Research Letters* 1 (1981) 6-9.

[5] J. Bisschop and A. Meeraus, "On the development of a general algebraic modeling system in a strategic planning environment", *Mathematical Programming Study* 20 (1982) 1-29.

[6] T. Cauchois, "The world coffee model", M.Sc. Diss., Massachusetts Institute of Technology (Cambridge, MA, 1980).

[7] C.F. Coleman and J.J. More, "Estimation of sparse jacobian matrices and graph coloring problems", *SIAM Journal of Numerical Analysis* (1983) 187-209.

[8] A.R. Colville, "A comparative study of nonlinear programming codes", in: H.W. Kuhn, ed., *Proceedings of the Princeton Symposium on Mathematical Programming* (Princeton University Press, 1970).

[9] A.R. Curtis, M.J.D. Powell and J.K. Reid, "On the estimation of sparse jacobian matrices", *Journal of the Institute of Mathematics and its Applications* 13 (1974) 117-119.

[10] R.S. Dembo and S. Sahi, "A globally convergent framework for linearly constrained nonlinear optimization", Working Paper B69, Yale School of Organization and Management, Yale University (New Haven, CT, 1983).

[11] A. Drud, "Optimization in large partly nonlinear systems", in: J. Cea, ed., *Optimization techniques. Modeling and optimization in the service of Man*, Part 2, Lecture notes in computer science, Vol. 41 (Springer-Verlag, Berlin, Heidelberg, New York, 1976) 312-329.

[12] A Drud and A. Meeraus, "CONOPT - A system for large-scale dynamic optimization - User's guide", Technical note 16, Development Research Center, World Bank (Washington, DC, 1980).

[13] R. Fourer, "Solving staircase linear programs by the simplex method, Part 1: Inversion", *Mathematical Programming* 23 (1983) 274-313.

[14] C.B. Garcia and W.I. Zangwill, *Pathways to solutions, fixed points, and equilibria* (Prentice-Hall, NJ, 1983).

[15] A. Gelb, "Oil rent and development strategies: A model for Indonesia", Development Research Department, World Bank (Washington, DC, 1983).

[16] P.M.J. Harris, "Pivot selection methods of the devex LP code", *Mathematical Programming* 5 (1973) 1-28.

[17] E. Hellerman and D. Rarick, "Reinversion with the preassigned pivot procedure", *Mathematical Programming* 1 (1971) 195-216.

[18] E. Hellerman and D. Rarick, "The partitioned preassigned pivot procedure", in: D.J. Rose and R.A. Willoughby, eds., *Sparse matrices and their applications* (Plenum Press, New York, 1972) pp. 67-76.

[19] J.E. Kalan, "Aspects of large-scale in-core linear programming", in: *Proceedings of ACM conference, Chicago, 1971*, pp. 304-313.

[20] L.S. Lasdon, A.D. Waren, A. Jain and M. Ratner, "Design and testing of a generalized reduced gradient code for nonlinear programming", *ACM Transactions on Mathematical Software* 4 (1978) 34-50.

[21] L.S. Lasdon and N.H. Kim, "SLP User's Guide", Department of General Business, School of Business Administration, University of Texas (Austin, Texas, 1983).

[22] J.B. Mantell and L.S. Lasdon, "A GRG algorithm for econometric control problems", *Annals of Economic and Social Measurement* 6 (1978) 581-597.

[23] B.A. Murtagh and M.A. Saunders, "Large-scale linearly constrained optimization", *Mathematical Programming* 14 (1978) 41-72.

[24] B.A. Murtagh and M.A. Saunders, "MINOS/AUGMENTED user's manual", Report SOL 80-14 (1980), Department of Operations Research, Stanford University, Stanford, CA.

[25] B.A. Murtagh and M.A. Saunders, "A projected lagrangian algorithm and its implementation for sparse nonlinear constraints", *Mathematical Programming Study* 16 (1982) 84-117.

[26] B.A. Murtagh and M.A. Saunders, "MINOS 5.0 User's Guide", Report SOL 83-20 (1983), Department of Operations Research, Stanford University, Stanford, CA.

[27] R.S. Pindyck, "Gains to producers from the cartelization of exhaustible resources", *Review of Economics and Statistics* 60 (1978) 238-251.

[28] M.J.D. Powell, "A hybrid method for nonlinear equations", and "A FORTRAN subroutine for solving systems of nonlinear algebraic equations", in: P. Rabinowitz, ed., *Numerical methods for nonlinear algebraic equations* (Gordon and Breach, London, 1970).

[29] K. Schittkowski, *Nonlinear programming codes*, Lecture Note in Economics and Mathematical Systems, vol. 183 (Springer-Verlag, Berlin, Heidelberg, New York, 1980).

[30] "APEX III Reference Manual Version 1.2", CDC Manual 76070000.

[31] "Mathematical Programming System-Extended (MPSX), and Generalized Upper Bounding (GUB)", IBM manual SH20-0968-1.