LASCALA - A LANGUAGE FOR LARGE SCALE LINEAR ALGEBRA*

by

A.W. Westerberg & T.J. Berna

DRC-06-4-79

January 1979

Department of Chemical Engineering
Carnegie-Melion University
Pittsburgh, PA 15213

^Presented at SIAM Symposium on Sparse Matrix Computations, Knoxville, TN, November 1978.

This work supported by NSF Grant ENG-76-80149.

LASCALA — A Language for Large Scale Linear Algebra

A.W. Westerberg and T.J. Berna

Abstract. A problem-oriented language is described which is capable of directing the calculation sequence associated with solving large, sparse, linear algebra systems which in general will require the use of mass memory. Although the concept of LASCALA arose in connection with the problem of optimizing large chemical processes, its application is suitable for use with any large matrices having a loosely connected block diagonal structure.

1. Introduction. One approach used to develop a model of a chemical process is to write all of the constraints associated with the process and to solve them simultaneously. These constraints include linear and nonlinear equality and inequality algebraic constraints, and for a typical process there are often several thousand such constraints. Once the model has been generated, one would like to find the set of feasible (or optimal feasible) solutions which describe the process. Unfortunately, it is frequently undesirable, if not impossible, to handle all of the constraints simultaneously without resorting to the use of mass memory devices for auxiliary storage.

The purpose of our work has been to develop an optimization scheme capable of finding optimal feasible solutions for large chemical processes. Our first step was to develop a scheme for computing the L/U factors of the large Jacobian matrix associated with the process constraints. Our algorithm (Westerberg and Berna, 1978) performs the factorization in a block-by-block manner to avoid using an excessive amount of core storage. In order to implement the ideas presented in that paper and in order to extend these ideas for use with our optimization algorithm (Berna, Locke and Westerberg, 1978), we advocate the use of a problem-oriented language such as LASCALA (Large Scale Linear Algebra).

We are in the early stages of developing LASCALA. Although the sample problem presented here has a chemical engineering parentage, LASCALA is well-suited for use in solving any problems that give rise to a large bordered block diagonal (BBD) Jacobian matrix. Presently LASCALA programs must be written manually, but its full potential can be realized only when the programs are generated automatically. Our primary objective in writing this paper is to introduce LASCALA in concept and to show the types of commands that such a language should have.

Throughout the discussion we assume that the user has available a sparse matrix package capable of performing an L/U factorization of a sparse matrix and of performing the forward and backward substitutions required to solve linear algebraic systems. We have given this hypothetical package the name SPARSE and assume it contains four routines: ANALYSE, FACTOR, FWD and BACK. The ANALYSE step, which determines the pivot sequence, must have some provision for handling nonpivot flags. FACTOR performs the elimination for a given pivot sequence. The routines FWD and BACK must be able to perform separately the forward and backward steps of the Gaussian elimination where the coefficient matrix is the factored matrix or its transparse. Our intention in designing LASCALA has been to keep the problem-oriented language independent of any particular sparse matrix code.

The discussion which follows is divided into six sections. The first section describes the nature of the chemical process design problem. Following a statement of the design problem, there is a statement of the optimization algorithm and a description of the method used to generate the process constraints. Section 5 describes some of the sparse matrix manipulations required in our work. In Section 6 we describe LASCALA in greater detail, and we discuss future work in Section 7.

2. The Design Process. A typical design problem is stated as follows: Design a process for producing 1 million kilograms of a chemical "C" per year. The product must be 99.9% pure, and available raw materials are chemicals "A" and "B". In addition to the explicit

requirements stated above, there are several implicit constraints requiring that the final design must: minimize annual operating expenses, comply with local pollution codes, meet certain safety standards, etc. A designer faced with solving this problem normally begins by considering each of the elements which might go into the process: the reactor, purification units, heat exchangers, etc. Before analyzing the behavior of the entire process the designer must analyze the behavior of each element. In considering the reactor, for example, the designer must consider various operating conditions available and alternative reaction schemes. As each of the elements is analyzed, the designer begins to link them together and to analyze the behavior of the entire network. The designer then continues to work with the network until an acceptable (perhaps an optimal) process is discovered.

A desirable feature of any design package is that the user must be able to place arbitrary specifications on the process. For example, the original problem statement might specify the temperature, pressure, composition or flowrate at any point in the process. The specifications might require that some function of these variables be satisfied. In any event, the final design must satisfy all specifications imposed on the process.

When the entire chemical process is considered there are n+r+q variables and n equality constraints. The user specifies values for q of the variables; effectively, these variables become constants for the remainder of the problem. The Jacobian matrix for the equality constraints is factored into the product of a lower triangular matrix $\underline{\underline{L}}$ and an upper triangular matrix $\underline{\underline{U}}$. The n variables corresponding to the pivots of the factored Jacobian matrix become the dependent variables $\underline{x}$ and the remaining variables become the decision (independent) variables $\underline{u}$. The r decision variables represent the actual degrees of freedom for optimizing the chemical process design.

Before concluding our discussion of the chemical process design problem, we would like to say a few words about the structure of the Jacobian matrix corresponding to a typical chemical process. The Jacobian has a bordered block diagonal structure; the blocks correspond to the various elements in the process, and the overlap between blocks is due to the connections among the elements in the network. One can expect to find Jacobian matrices having this structure whenever the constraints are related to a set of loosely connected modules. We say more about the role that this structure plays in the design process later in this paper. Now let us consider the optimization problem as a mathematician might see it.

3. <u>The Optimization Algorithm</u>. The problem of finding an optimal design for a particular chemical process can be stated as follows:

$$\text{Min } \$(z)$$

subject to

(PI)

$$g(2) = 0$$
$$h(z) * 0$$
$$z \; e R^{n+r}$$
$$g{:}R^{n+r} - R^{n}$$
$$hsR^{***} \rightarrow R^{m}$$
$$\Phi : R^{n+r} \rightarrow R$$

where § may be the net annual operating expenses, and g and h are the constraints described in Section 2. For problems of industrial significance n and m range from about 1000 to 50,000 and r ranges from 1 to about 50. Typical values for n and r are 10,000 and 10, respectively. In order to solve (Pi) within a reasonable amount of core storage and execution time, we have modified an algorithm published by Powell (1977). Powell's algorithm relies heavily on some work published by Han (1975). In the remainder of this section we sketch the development of the relevant details of our optimization algorithm (Berna, Locke and Westerberg, 1978).

In order to develop the conditions necessary for z* to be an optimal feasible solution to (Pi), we define the Lagrange function

(1) $$L(z,X,uO = \S(z) - X^{T}g(z) - n^{T}h(z)$$

The necessary conditions for optimality then become

(2) $$\frac{\partial L}{\partial z} = 0 = \frac{\partial \Phi}{\partial z} - \frac{\partial g^T}{\partial z}\lambda - \frac{\partial h^T}{\partial z}\mu$$

(3) $$g(z) = 0 \quad ; \quad h(z) \geq 0$$

(4) $$\mu^T h = 0 \quad ; \quad \mu \geq 0$$

The Newton-Raphson scheme for solving (2) gives

(5) $$\frac{\partial L}{\partial z} + \frac{\partial^2 L}{\partial z^2}\Delta z = 0 = \frac{\partial \Phi}{\partial z} + \frac{\partial^2 L}{\partial z^2}\Delta z - \frac{\partial g^T}{\partial z}\lambda - \frac{\partial h^T}{\partial z}\mu$$

In a similar fashion, the constraints in (3) can be linearized to give

(6) $$\begin{aligned} g + \frac{\partial g}{\partial z^T}\Delta z &= 0 \\ h + \frac{\partial h}{\partial z^T}\Delta z &\geq 0 \end{aligned}$$

At this point, we define the following quadratic function

(7) $$Q(\Delta z) = \Phi + \frac{\partial \Phi}{\partial z^T}\Delta z + \frac{1}{2}\Delta z^T \frac{\partial^2 L}{\partial z \partial z^T}\Delta z$$

Now consider the quadratic programming problem (QPP) formed by minimizing $Q(\Delta z)$ subject to the linearized constraints (6):

(QPP) $$\begin{aligned} &\min_{\Delta z} Q(\Delta z) \\ \text{s.t.}\quad & g + \frac{\partial g}{\partial z^T}\Delta z = 0 \\ & h + \frac{\partial h}{\partial z^T}\Delta z \geq 0 \end{aligned}$$

The necessary conditions for Δz^* to be a solution to (QPP) are that (5) and (6) hold and that the following constraints be satisfied:

(8) $$\begin{aligned} &\mu^T\left(h + \frac{\partial h}{\partial z^T}\Delta z^*\right) = 0 \\ &\mu \geq 0 \end{aligned}$$

Based on the observation that (5), (6) and (8) are satisfied by solving (QPP) and that these constraints represent the Newton-Raphson iteration from z_k to z_{k+1}, Powell (1977) developed an algorithm which solves (Pi) by generating and solving a sequence of QPP's. Instead of actually computing $\frac{\partial^2 L}{\partial z \partial z^T}$ the Hessian of the Lagrange function, Powell uses a series of pairwise rank-one updates to approximate this matrix. The basic algorithm is given as follows:

Step 1. Guess C = I and z $\left(C \text{ is } \frac{\partial^2 L}{\partial z \partial z^T} \right)$

Step 2. Evaluate $, g, g, h, $\frac{\partial}{\partial z^T}$, $\frac{\partial}{\partial z}$, g

Step 3, Solve QPP for Az*, X*, p,*

Step 4. Estimate

$$\left(\frac{\partial L}{\partial \Delta z}\right)_{z+\Delta z,est} = \left(\frac{\partial \Phi}{\partial z}\right)_z + \left(\frac{\partial^2 L}{\partial z \partial z^T}\right)_z \Delta z - \left(\frac{\partial g^T}{\partial z}\right)_z \lambda^* - \left(\frac{\partial h^T}{\partial z}\right)_z \mu^*$$

Step 5. Move to z_{nexc} = z + Az and repeat Step 2. Using the same X* and |A* from Step 4, evaluate

$$(fe)_{next,act} = \left(\frac{\partial \phi}{\partial z}\right)_{next} \quad - \quad {}^T{}_{next} \quad - \quad {}^T{}_{next} \quad {}^*{}_{next}$$

(At z_{nexu}., Az = 0 so the term $\frac{\partial^2 L}{\partial z \partial z}$ Az is zero.) Then use

$$\delta = \left[\left(\frac{\partial L}{\partial \Delta z}\right)_{est} - \left(\frac{\partial L}{\partial \Delta z}\right)_{act} \right]$$ to update C.

Step 6. Iterate from Step 3 until ||Az|| is small.

For relatively small problems (n £ 50) this algorithm works extremely well. For very large problems (n ^ 1000) the size of C exceeds the core storage space available on most machines; furthermore, the computational requirements involved in updating C (by computing $w_k w_k^T$) are prohibitive for large values of n. Our algorithm, which extends the above algorithm, uses the linearized equality constraints in (6) to set up a reduced (QPP) where the size of the Hessian matrix is rxr instead of (nfr)x(nfr). A second advantage associated with the extended algorithm is that we never compute W, $W_K,{}_K^T$; instead, we always compute the appropriate scalar products associated with the pre- and post-multiplication operators of C. In other words, we never need C alone. Rather, we need terms of the form $p^T Cq$ where these terms are computed (after *I* rank-one updates of C, *A* « n) as follows

$$P^T Cq - p^T q + \sum_{k=1}^{\ell} (p^T W_k)(W_k^T q)$$

Uiis operation requires (2X4-1) *tthA* multiplications; if C is treated as a full matrix, this operation requires n^2+n multiplications. This difference does not even include the *A*n^2 multiplications (for the *A* outer products W_K«W_K^T) required to compute C.

The basic approach of our algorithm is to partition the original variable set z into xSR^n and ueR^r. This partitioning is accomplished by performing an L/U factorization of the matrix —^-_. The variables

corresponding to nonzero pivots are labeled x; the remaining variables are labeled u. At each iteration we use these factors to set up a reduced QPP which is solved to obtain Au and p,. The values for Ax and X are computed by performing the back substitution based on Au and *\x.* Figure 1 illustrates this process. The details of the optimization algorithm are published elsewhere (Berna, Locke, Westerberg, 1978), but in the present discussion we wish to focus on the sparse matrix manipulations required to carry out the optimization procedure. Before discussing these manipulations we first need to describe the process for generating the Jacobian matrices and residuals associated with each of the constraints in the process. Section 4 describes this procedure, and we continue our discussion of the sparse matrix operations in Section 5.

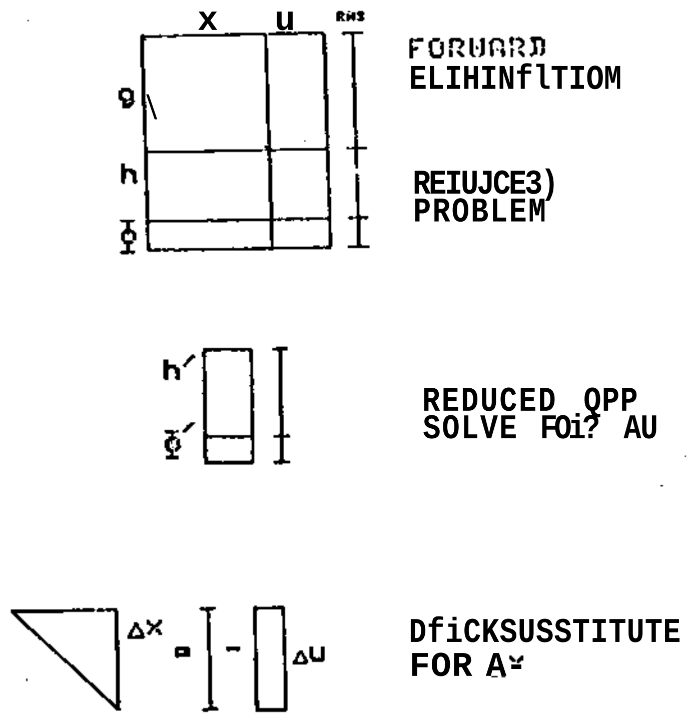


Figure 1. Schematic diagram of procedure used to solve very large quadratic programming problems (QPP's).

4. Process Model: Packets and Generators, The discussion in this section centers around the approach adopted for modeling a chemical process. The job of the process model is to generate the constraints in (Pi) that describe a process. Conventional chemical process simulation packages visualize the process model as a set of interconnected unit subroutines which operate on inlet stream values and produce outlet stream values. In addition to the process streams associated with each unit, there may be one or more parameters which the user is required to specify. As we mentioned earlier it is far more desirable for a process simulator to accept arbitrary constraints on the process; such is the case with equation based simulation packages. We are interested in developing a process simulator capable of generating the Jacobian elements and right-hand sides associated with the constraints in (Pi). Jacobian elements are computed by "generators" which compute the elements based on the values of variables in the associated "variable packets." The rows are identified as those belonging to the "equation packets" associated with the

particular generator. Each generator has a set of packets associated with it; Figure 2 illustrates.

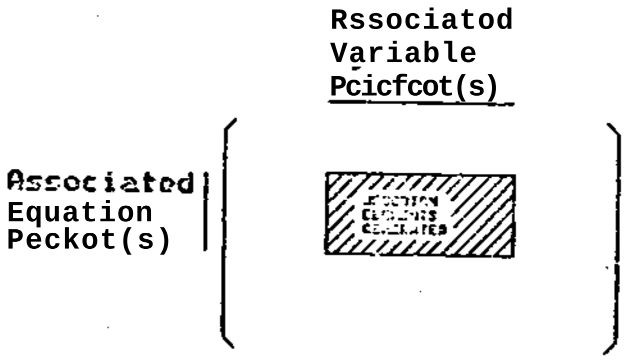


Figure 2. Jacobian elements are generated by a generator in the particular rows and columns identified as the associated equation and variable packet(s).

In order to clarify the presentation of these concepts, we have developed a Generator, Equation and Variable (GEV) diagram which schematically illustrates the connection between the various generators and packets in a given process model. Figure 3 illustrates the GEV diagram and corresponding Jacobian matrix for a simple process. Variable packets are illustrated by labeled solid horizontal lines, equation packets by labeled solid vertical lines. Generators are represented by labeled boxes, and the packets associated with each generator are those connected by a dashed line to that generator. In this example variable packets "S2" and "Cost" are associated with two generators while the equation packet, "Cost Function," is associated with two generators. This diagram illustrates the fact that each generator may be associated with many packets and that any packet may be associated with more than one generator. From a chemical engineering viewpoint this concept in modeling offers several advantages: one is that the physical property calculations may be included as separate generators, another is that these physical properties can be associated with a process stream instead of using the less natural association of physical properties with process units. To those unfamiliar with chemical process models, this latter distinction may seem to be of little importance. The significance of the statement is more clearly understood when one considers that: (1) all process

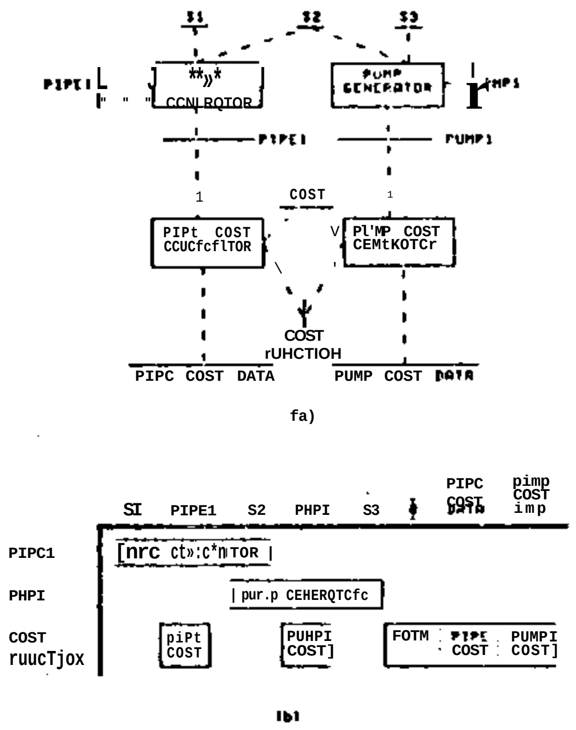


Figure 3. GEV diagram (a) for process containing a pump and a length of pipe; (b) shows Jacobian matrix associated with this GEV.

simulators currently associate physical property calculations with a process unit and not with a process stream, and (2) because a process stream is usually associated with two process units there is the possibility that two different values for a computed physical property could be assigned to the same process stream. Our convention can eliminate the possibility of having this discrepancy arise.

5. Manipulations for Large Problems. In Section 3 we alluded to an optimization algorithm which solves (P1) by solving a series of QPP's in the degrees of freedom, Δu, only. In order to arrive at this reduced QPP one must be able to compute the L/U factors of the Jacobian matrix $\frac{\partial g}{\partial x^T}$. For large problems, those of industrial significance, the Jacobian matrix must be factored in a block-by-block manner; in this section we discuss the manipulations required to solve (P1) in a reasonable amount of core storage space. For convenience we introduce the following notation:

$$J_x = \frac{\partial g}{\partial x^T} \qquad J_u = \frac{\partial g}{\partial u^T}$$

$$A = J_x^{-1} J_u \qquad v = J_x^{-1} g$$

$$K_x = \frac{\partial h}{\partial x^T} \qquad K_u = \frac{\partial h}{\partial u^T}$$

$\hat{C}$ = Quasi-Newton Approximation to $\frac{\partial^2 L}{\partial z \partial z^T}$

w_k = k^{th} rank-one update to $\hat{C}$

Although the ideas presented in this section first arose in connection with chemical process models, we wish to stress that they may be applied to any loosely connected network of modules giving rise to a bordered block diagonal (BBD) Jacobian matrix.

The structure of a typical chemical process naturally gives rise to a bordered block diagonal (BBD) Jacobian matrix. Our first goal is to compute $A = J_x^{-1} J_u$ and $v = J_x^{-1} g$ in a block-by-block fashion. The simplest case is illustrated in Figure 4a. In this discussion a single prime (') will denote Jacobian and right-hand side elements associated with equation packet "a" and a double prime (") will be used for those associated with "b". The Jacobian matrix associated with this process is shown in Figure 4b. The first step in factoring this matrix is to generate the Jacobian elements associated with unit "a". Nonpivot flags are placed on the equation packet a' and on the variable packets ab' and ba' because these packets are shared with a generator that has not yet been processed, and therefore there may be other nonzeros in these columns. The active system is shown in Figure 5a. We then pivot to compute the L/U factors of this matrix to get the structure shown in Figure 5b. The factored block (the non-crosshatched portion in Figure 5b) is stored in one area of mass memory, the residual block (the crosshatched portion) in another. We next repeat this process for "b". The active matrix with the appropriate nonpivot flags is shown in Figure 5c. Pivoting on the

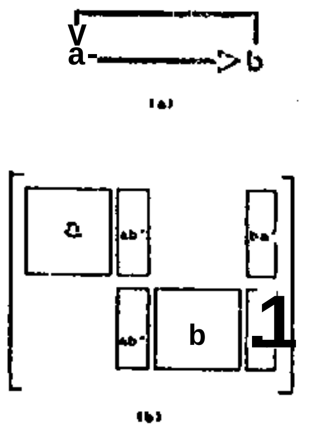


Figure 4. Process containing two interconnected units (a) and its associated Jacobian matrix (b).

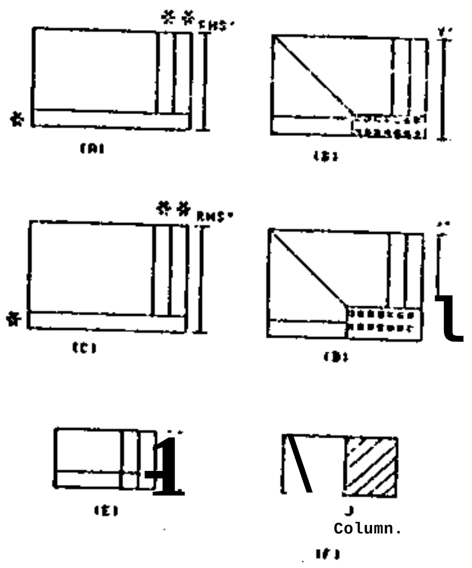
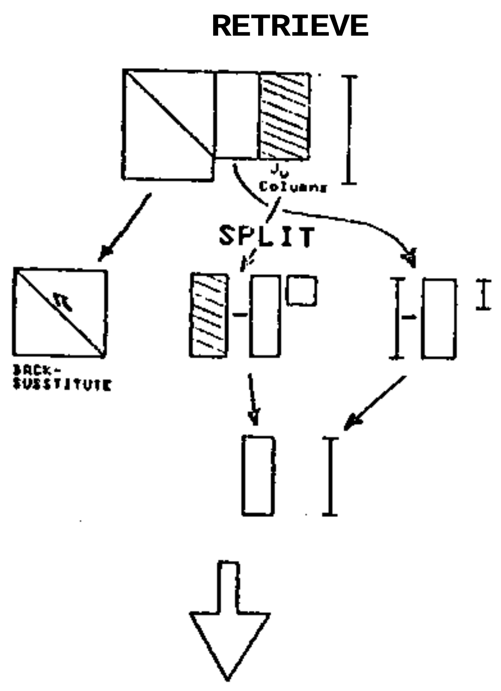


Figure 5. Steps (a) through (f) illustrate the sequence of operations used to factor the Jacobian matrix of Figure 4.

allowable rows and columns leads to the structure represented in Figure 5d. *The* factored block is sent to mass memory, and the two residual blocks (that is, this one and the one from $^{fl}a^{lf}$) are combined to give the matrix shown in Figure 5e. There is no need to prevent pivoting in any rows or columns, therefore pivoting leads to the matrix illustrated in Figure 5f. The nonpivoted columns are equivalent to $L"^{1}J_{u}$ so A is obtained by performing only the backward substitution on these columns. In order to calculate $A = J_{x}^{-1}J_{u}$ we move the columns corresponding to $L\sim^{\pm} J_u$ to the right hand side of the equality and backsubstitute. Once the backsubstitution has been performed for the residual block, we retrieve the factored block corresponding to 'b^1. We identify those columns associated with J_u and move these to the right hand side of the equality. This procedure is shown schematically in Figure 6. The process is separated for $^f a^f$.



REPEfiT FOR^{v}a' TO GET

f»=j"x$^!$ J_u

$v=J_x^{-1}\ g$

Figure 6. Schematic representation of backward substitution step associated with the Jacobian matrix in Figure 4.

It is a simple matter to show that the following operations can also be performed in a block-by-block manner:

$$K_u + K_x A$$

$$CA \quad ; \quad Cv$$

$$A^T A$$

We have illustrated some of the manipulations that are required; we now examine LASCALA in greater detail keeping in mind the capabilities it must exhibit.

6, LASCALA Commands, As stated in previous sections of this paper, our objective is to develop a problem-oriented language capable of describing certain sparse matrix manipulations. Table 1 contains a list of commands currently proposed for LASCALA. In the present section we describe each of these commands and introduce the computational environment developed to handle large sparse matrices.

TABLE 1

LASCALA Conmands

Memory Management	Transfer
Conversion	Local Global Convert
Generating	Set GNGF JGEN KGEN JRGEN KRGEN
Solving	Set PNPF Analyze Factor FWD BKWD
Other Arithmetic	Add Subt Mult
QPP	QPP

The first command for discussion is the TRANSFER command. To a large extent the efficiency of a sparse matrix scheme is determined by how effectively it uses mass memory and core storage space. On the other hand, one would like to reduce the number of data transfers to mass memory devices because these are very costly operations. On the other hand, one would like to avoid cluttering up core space with data that is not needed for the current matrix calculation (but which may be required later)

Figure 7 illustrates one computational environment which attempts to use core space efficiently while organizing data to reduce the number of transfers to mass memory. The core storage space is divided

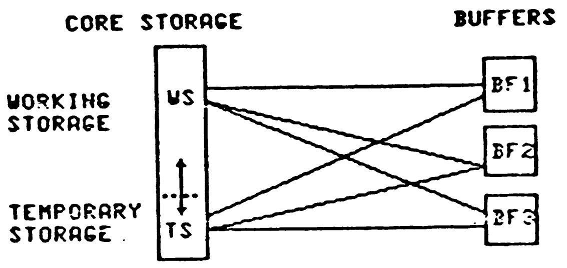


Figure 7. Data structure used with large systems.

into a working store (WS) and a temporary store (TS); the fraction of core allocated to either category can be changed as required by the executive routines. Data which can be sent to mass memory is stored in one of three buffers according to its type. The syntax of the TRANSFER command is as follows:

```
TRANSFER 'type' 'from' 'to' ('list of names')
```

where acceptable symbols for "type", and "from" and "to" are given in Table 2.

TABLE 2

Acceptable Parameters for the TRANSFER Command

Data Type Designator	
VP	Variable Packet
FB	Factored Block
RB	Residual Block
VC	Vector
Location Designators	
BF1, BF2, BF3	Buffers 1, 2 and 3, respectively
WS	Working Storage Space
TS	Temporary Storage Space

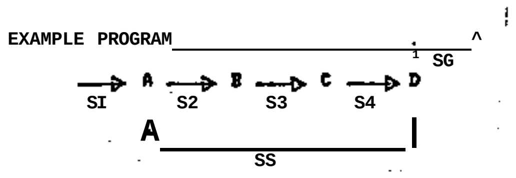
Within WS is a matrix structure used for core-resident matrix operations. This matrix has a set of LOCAL row and column indices which differ from the set of GLOBAL row and column indices. The commands LOCAL and GLOBAL convert a given set of indices <u>into</u> LOCAL and GLOBAL indices, respectively.

The CONVERT command is used to convert from one sparse matrix data structure to another. At present a matrix may use any of four data structures: full, sparse, rank-one or the data structure peculiar to the user's sparse matrix package. The sparse data structure is a list of the nonzeros in the matrix along with their respective row and column indices. The rank-one data structure contains the k rank-one updates (w_x) where the matrix is given by adding the k outer products $w_i w_i^T$ to the identity matrix. The other two data structures are selfexplanatory.

The constraint generating commands are SET GNGF JGEN, KGEN, JRGEN and KRGEN. The first command is used to set "generate/no-generate" flags in order to suppress and/or ignore generation of certain rows and/or columns of the Jacobian matrix. The commands JGEN and KGEN are used to generate the partial derivatives associated with the equality and inequality constraints, respectively, while JRGEN and KRGEN are used to generate the residuals of these constraints.

The commands associated with solving a linear system are SET PNPF, ANALYZE, FACTOR, FWD and BACK. The first command is used to set "pivot/no-pivot" flags to indicate any rows or columns whose elements may not be considered for pivot variables. The remaining four commands are used to call the appropriate routines of the user-supplied sparse matrix package described in the Introduction. Other commands which should be compatible with the sparse matrix package are: ADD, SUBT and MULT. These commands are used to add, subtract or multiply two matrices where one or both of the matrices is sparse. The QPP command is used to call the appropriate routines to solve the QPP described in Section 3. Due to the nature of our QPP we do not use a sparse matrix code but rather use a full matrix version of Fletcher's generalized QPP algorithm.

In order to give the reader a more concrete example of how LASCALA might be used, we have included a portion of a sample program. In this program (see Figure 8) each of units A, B, C and D are so large that they must be treated separately in core. The sample program illustrates the sequence of LASCALA commands required to perform the elimination and forward substitution on the linearized equality constraints associated with unit A. A similar sequence of commands is required for the other units and for the backward substitution.



```
USER INPUT COMPLETE. ALL VAPIABLES/EQUATICMS INITIALIZED 4 SCALED.
C     UNIT A

      TRANSFER VP B'1 US CS1,SS,Vf»A,S2)
      TRANSFER VC BF3 WS CLGNOFA,LPHPFA)
      SET GNC-F (LGN5FA)
      SET PKPF CL^KPFA)
      JOEN
      LCCAL
      CONVERT SP SPMP1
      ANALYZE
      FIO
      OLOCAL
      TRANSFER TP WS DF2 CVPA)
      TRANSFER PB US TS (PDA)
      TPAUSFER VP W3 TS CSS)
      TPANSFER VP WS LF1 CVPA>
      TRANSFER VP WS EF1 <ST>
```

Figure 8. Typical segment from LASCALA program.

7. Extensions to LASCALA. An obvious extension to the work described thus far, and one which we are currently studying, is to develop a compiler which will automatically generate the necessary sequence of LASCALA commands associated with the user's description of his problem. The compiler should make some attempt to optimize the order in which the variables are generated/eliminated. The compiler should also contain guidelines for deciding what order to use in storing variables for faster retrieval from mass memory.

REFERENCES

Berna, T.J., M.H. Locke and A.W, Westerberg, "A New Approach to Optimization of Chemical Processes," Paper presented at National AlChE Meeting, Miami, FL. (November 1978).

Powell, M.J.D., "A Fast Algorithm for Nonlinear Constrained Optimization Calculations," Paper presented at the 1977 Dundee Conference on Numerical Analysis.

Westerberg, A.W. and T.J. Berna, "Decomposition of Very Large-Scale Newton-Raphson Based Flowsheeting Problems," Computers and Chemical Engineering, 2, 61-63 (1978).