IPSC-MATLAB
REFERENCE MANUAL

Shirish Chinchalkar

September 1992

Contents

List of Tables

List of Figures

1 Introduction

IPSC-MATLAB is a programming environment for running MATLAB programs on the Intel iPSC/860 hypercube. The system is designed such that the user can execute computationally intensive portions of MATLAB programs on the hypercube, whereas all other code is executed on a Sun-4 workstation (SPARCstation). The workstation acts as a remote host for the hypercube. MATLAB variables can migrate from the front-end (workstation) to the back-end (hypercube) and vice versa. IPSC-MATLAB combines the flexibility of MATLAB programming with the speed of the Intel hypercube.

Writing parallel programs in Fortran or C for the hypercube is a tedious task. Testing and debugging programs is difficult and time consuming. On the other hand, MATLAB, which runs on workstations, provides a convenient means of writing programs. However, because of the limited speed and memory of workstations, large problems cannot be solved. The hypercube is fast and has more memory. IPSC-MATLAB attempts to use the workstation as well as the hypercube to provide the user with a fast, flexible environment for programming.

2 Overall Design

There is only one data type in dense MATLAB: the complex $m \times n$ matrix. IPSC-MATLAB uses two data types: (a) a real vector (includes scalars) which is replicated on all processors and (b) a real matrix which is distributed using the torus wrap mapping (see for eg. [Geijn 91]). Such a representation helps in providing a simple interface to the user, similar to sequential MATLAB, since the user does not have to deal with mapping each variable individually. Unfortunately, because of this simple data structure, it is not possible to exploit symmetry or sparsity in matrices, so there can be considerable waste of storage space, which is scarce on the Intel hypercube.

The torus wrap mapping was selected because it is suitable for a wide variety of parallel matrix computations such as the LU and QR factorizations, the Cholesky decomposition, and the reduction to tridiagonal form. Moreover, row and column wrap mappings are special cases of the torus wrap mapping. In the torus wrap mapping, the processors are logically arranged in a 2-dimensional grid of size $r \times c$, where r is the number of rows in the processor grid, c is the number of columns, and $p = rc$ is the number of processors. Figure 1 shows 8 processors arranged in a 4×2 grid. Each processor is identified uniquely by a tuple $(i, j), i = 0..r-1, j = 0..c-1$. For any matrix, the entry (k, l) of the matrix is stored on processor $(k\%r, l\%c)$. Figure 2 shows how an 8×8 matrix is mapped onto a processor grid of size 2×2. A more general scheme is the block torus wrap mapping, where blocks (instead of matrix elements) are distributed among processors. In general, the efficiency of parallelization is a function of r and c. Moreover, the best values of r and c for a given value of p, the number of processors, depend on the algorithm. However, a reasonably good choice can be made *a-priori*. Table 1 shows the values of r and c chosen for different values of p. These choices are coded into IPSC-MATLAB and the user is not expected to modify

0	1
2	3
4	5
6	7

Figure 1: The arrangement of 8 processors in the form of a 4×2 grid

$$\begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 2 & 3 & 2 & 3 & 2 & 3 & 2 & 3 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 2 & 3 & 2 & 3 & 2 & 3 & 2 & 3 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 2 & 3 & 2 & 3 & 2 & 3 & 2 & 3 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 2 & 3 & 2 & 3 & 2 & 3 & 2 & 3 \end{bmatrix}$$

Figure 2: The torus wrap mapping for an 8×8 matrix. Each number indicates the processor assigned to that entry of the matrix.

them.

Figure 3 shows the overall organization of IPSC-MATLAB. IPSC-MATLAB consists of two major components; one runs on a SUN SPARCstation and the other runs on the Intel iPSC/860 hypercube. The SPARCstation acts as a remote host for the hypercube. The hypercube SRM was not chosen as a host because it is slow (a 80386 PC) and because the MATLAB license was available only for the SPARCstation. It is also difficult to re-implement MATLAB in its entirety in a short time frame. Hence it was decided to use MATLAB on the SPARCstation and make it communicate with the hypercube using

p	r	c
1	1	1
2	2	1
4	2	2
8	4	2
16	4	4
32	8	4

Table 1: Choices of r and c for different values of p

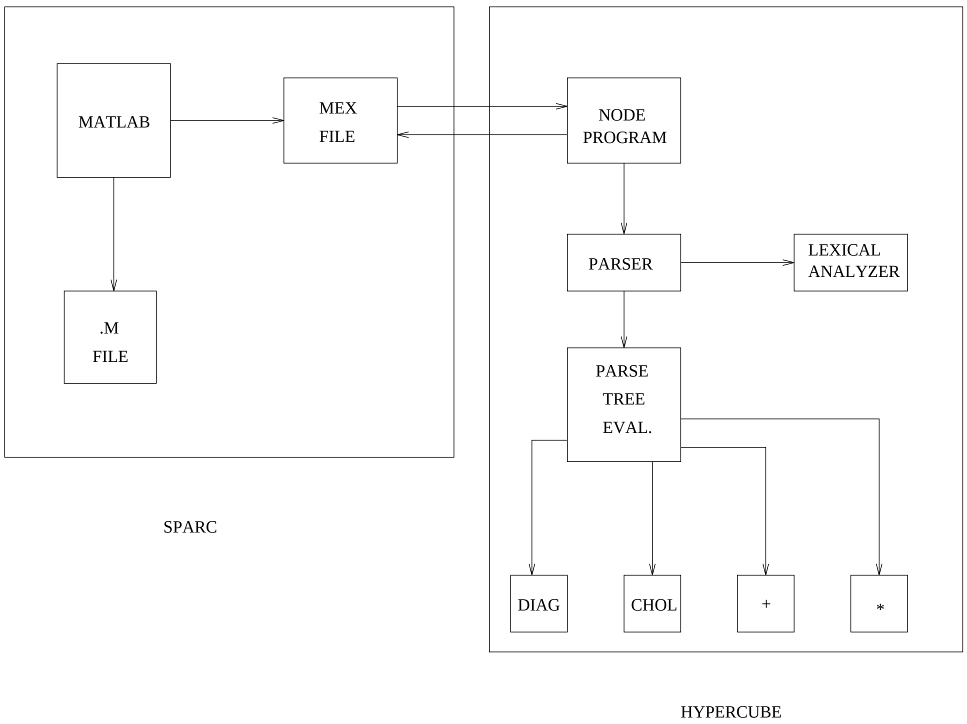

Figure 3: Block diagram showing various components of IPSC-MATLAB

Intel's remote communications software. Figure 4 compares the message passing speed between the remote host and the hypercube, the SRM and the hypercube, and two nodes of the hypercube. The time for 1000 messages (and acknowledgments) was measured and the average is shown in figure 4. It can be seen that message passing between the remote host and the nodes of the hypercube is 3 orders of magnitude slower than node-node communication, when the messages are short. It is also an order of magnitude slower than SRM-node communication. The speed of remote host to node communication is a strong function of the traffic on the network, which is shared by several machines. Consequently, the communication time may vary, however, the curves in figure 4 are representative of typical communication overhead. The case studies in sections 5 and 6 show that it can result in serious degradation in performance if the number of messages is high.

The user needs to call only one function called `ipsc()` to introduce parallelism into MATLAB. There are 5 ways of calling this function, as explained below. This function is implemented as a `mex` program. A `mex` program is a C function linked with a MATLAB library. When this function is called from within MATLAB, MATLAB dynamically links the `mex` file to itself before calling the function. The `mex` program acts as a host program. When `ipsc()` is called for the first time, it loads a node program on the hypercube. Whenever a MATLAB command needs to be executed on the hypercube, it is sent to all nodes of the hypercube. The node program consists of a lexical analyzer, a parser, a parse tree

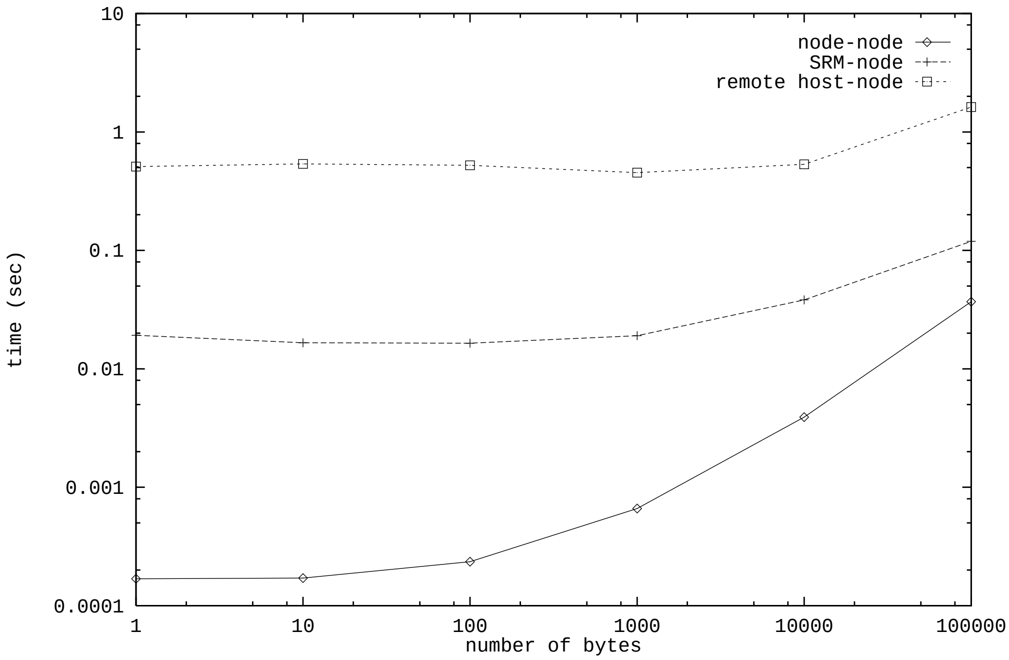

Figure 4: Elapsed time for sending a message of length n bytes and receiving an acknowledgment of length 0 bytes

evaluator, and a linear algebra module. The remote host and the cube use different internal representations for floating point numbers. Hence, it is necessary to reorder the bytes in messages sent to and from the remote host. All nodes are responsible for parsing the given command. If there is a syntax error or a semantic error, node 0 sends an error message to the host program, which prints the error message on the screen.

Currently, IPSC-MATLAB does not print the exact cause of a syntax error. If it finds a syntax error, it returns the offending portion of the input statement to MATLAB, which prints it on the screen.

There are 5 different ways of calling `ipsc()`, of which 1, 2, and 3 are most commonly used. Methods 4 and 5 are useful, though not essential. All other commands that do not involve a call to `ipsc()` are executed on the front-end.

1. `ipsc('name', var)`
   Where `var` is a variable residing on the front-end and `name` is a valid MATLAB variable name. This command sends the variable `var` from the front-end to the hypercube where it is called `name`. For example:

   ```
   >> x = rand(5,1)
   ```

```
x =

      0.2190
      0.0470
      0.6789
      0.6793
      0.9347

>> ipsc('x', x)
```

sends the vector `x` to the hypercube.

2. `ipsc('command')`
   where `command` is a MATLAB command. The command is executed on the hypercube. The command reference lists all commands that can be executed on the hypercube. For example, after executing the statements in 1, you can type:

   ```
   ipsc('y = 2*x')
   ```

   This will automatically create a vector `y` with the value `2*x` on the hypercube.

3. `var = ipsc('name');`
   This function call retrieves the variable `name` from the cube and assigns it to the front-end variable `var`. For example, after executing commands shown in 1 and 2, you can type:

   `>> y = ipsc('y')` which will produce the output

   ```
   y =

         0.4379
         0.0941
         1.3577
         1.3586
         1.8694
   >>
   ```

4. `ipsc(0)`
   This prints the number of calls to the function `ipsc()`, the number of variables sent to the hypercube, and the number of variables received from the hypercube. Commands listed in 1, 2, and 3 are counted in the number of calls. Commands of type 1 count as variables sent and those of type 3 count as variables received. This is a simple way of determining how dependent an algorithm is on host-node communication.

5. `ipsc(';')`
   This will print the amount of time the iPSC has spent on executing commands sent from the remote host. It *excludes* the time spent sending messages to and from the remote host.

3 What cannot be done in IPSC-MATLAB

In addition to the limitations described in the reference pages in the appendix, the following is yet to be implemented in IPSC-MATLAB:

- complex numbers
  IPSC-MATLAB cannot represent complex numbers; `sqrt()` will fail if called with a negative argument, `acos()` will fail if called with an argument greater than 1, etc.

- empty matrices in operations such as addition and subtraction.
  IPSC-MATLAB is capable of producing empty matrices and sending them to the front-end, however, use of such a matrix on the back-end will result in an error.

- operations on sparse matrices other than those mentioned in section 7.

- operations on strings.

- functions implemented using .m files, such as the optimization toolbox and the FFT toolbox.

- control-flow such as `if ...then ...else ...end` and `for ...end`.

- matrices as subscripts of matrices, subscripted arrays on the left hand side of an expression.
  This is perhaps the most serious limitation.

- graphics.
  All graphical operations must be performed on the front-end.

- special values such as `pi, eps, clock,` and `flops`.

- programming tools such as .m files, keyboard input, functions, and operating system level support.

- various factorizations such as `eig` and `svd`, special matrices and elementary matrix functions such as `expm` and `logm`.

Most or all of these limitations can be overcome at the expense of more programming effort. Currently only a limited functionality is added; more can be added as and when necessary.

4 An Example

Here is an example of how to generate a random symmetric positive definite matrix (`H`) with a given condition number ($10^{\mathtt{condn}}$) on the hypercube. Mathematically, the matrix H is obtained by multiplying 3 matrices as follows:

$$H = \left(I - \frac{2vv'}{v'v}\right) \Lambda \left(I - \frac{2vv'}{v'v}\right)$$

where Λ is a diagonal matrix with the given condition number and v is a random column vector of the correct size.
Sequential MATLAB code:

```
v = ones(n,1) - 2*rand(n,1);
c = log(10)*condn/(n-1);
for i=1:n, d(i,1) = (i-1)*c; end,
b = exp(d);
z = v.*b; t = v'*v;
H = diag(b) - (2/t)*z*v';
H = H - (2/t)*v*z';
H = H + (4/t/t)*(v'*z)*v*v';
```

IPSC-MATLAB code (Note: the first few lines are executed on the front-end as they do not call the function `ipsc()`):

```
v = ones(n,1) - 2*rand(n,1);
c = log(10)*condn/(n-1);
for i=1:n, d(i,1) = (i-1)*c; end,
b = exp(d);
ipsc('v',v);
ipsc('b',b);
ipsc('z = v.*b; t = v"*v');
ipsc('H = diag(b) - (2/t)*z*v"');
ipsc('H = H - (2/t)*v*z"');
ipsc('H = H + (4/t/t)*(v"*z)*v*v"');
```

5 Case study: Box-constrained dense positive definite quadratic programming

This section describes the use of IPSC-MATLAB for box-constrained positive definite quadratic programming (QP). A new algorithm for solving the following problem is presented in [Coleman 92b] and [Coleman 92c]:

$$\min_x \{q(x) \stackrel{\text{def}}{=} g^T x + \frac{1}{2} x^T H x : l \leq x \leq u\}$$

The algorithm works for any symmetric H; this section only describes the dense positive definite case. The most computationally intensive task in this algorithm is a Cholesky factorization of a large dense positive definite matrix. The original program was written in MATLAB and could not be used for solving large problems on a SUN workstation.

The program was parallelized by calling the function `ipsc()`. The following operations performed in the QP algorithm were executed in parallel on the back-end:

- Cholesky factorization
- Solution of triangular system of equations
- Matrix vector multiply
- Lower triangular matrix vector multiply
- Matrix scaling
- Addition of a diagonal matrix to a dense matrix

Initially, a random symmetric positive definite matrix H is generated in parallel on the iPSC. Later a matrix M is obtained from H at every iteration. Both these matrices reside permanently on the back end. All operations that deal with H and M are performed on the back-end, whereas almost all operations on scalars and vectors are performed on the front-end in MATLAB. The program uses several dozen variables; most of which reside on the front end. A few vectors are sent from the front-end to the back-end and vice versa at every iteration. Approximately 30 vectors needed to be sent from the front-end to the back-end and 20 from the back-end to the front-end at every iteration. This proved to be very expensive. So one MATLAB function (`startls()`), which performed line search and required the maximum amount of data movement, was re-written in C and moved to the back-end. This reduced the number of sends and receives to about half their original values.

Table 2 shows the amount of time taken for solving problems of size 1000 and 2000. The term 'number of calls' refers to the number of times the function `ipsc()` was called by the MATLAB program. Real time is the total turnaround time (wall clock time), whereas iPSC time is the time the algorithm would have taken if the remote host to node communication were instantaneous.

A separate C program for the iPSC was also written. This program did not use MATLAB. Table 3 compares the execution time for MATLAB, IPSC-MATLAB, and iPSC, for two problems. It can be seen that IPSC-MATLAB is significantly faster than plain MATLAB, but is slower than C code that runs on the iPSC only.

6 Case Study: Sparse box-constrained linear least squares

The algorithm in [Coleman 92b] can also be used for solving sparse box-constrained least square problems of the form:

$$\min_x \{ \|Ax - b\|_2 : l \le x \le u \}$$

n	1000	2000
iter.	15	14
time(iPSC) (sec)	68	207
time(real) (sec)	470	631
#calls	451	425
#sends	198	186
#recvs	144	135

Table 2: Dense positive definite QP using IPSC-MATLAB on 32 processors

method	$n = 1000$	$n = 2000$
MATLAB	144.6	1046.5
IPSC-MATLAB	31.3	45.1
iPSC	4.6	14.2

Table 3: Wall clock time (seconds) per iteration for 3 methods of solving dense positive definite QP (32 processors)

where A is an m-by-n sparse matrix, and m is much larger than n. This problem arises in several contexts; the one being explored is turbulent combustion where data in a 3D grid is approximated by a linear function of unknown parameters.

The above problem reduces to the following box-constrained quadratic programming problem:

$$\min_x \{q(x) \stackrel{\text{def}}{=} (-A^T b)^T x + \frac{1}{2} x^T (A^T A) x : l \leq x \leq u\}$$

Because of the underlying complexity of the sparse parallel code, it is difficult to provide a general interface as in the dense case. Moreover, for the dense case, the torus wrap mapping appears to result in efficient algorithms for a wide range of linear algebraic operations. It is not the case with sparse linear algebra. Different operations require different mappings of the same matrix. It was therefore decided to implement only those functions required by the QP solver as special-purpose functions in IPSC-MATLAB. The lexical analyzer, the parser, and the parse tree evaluator were modified to incorporate the sparse multifrontal solver from [Pothen 91]. A function `S_startls()` was written for the back-end. This function is identical to `startls()`, but handles the sparse case.

C functions for the iPSC were written by Chunguang Sun to generate the matrix $H = A^T A$ and the vector $g = -A^T b$ according to discretization of a cubical domain by a regular grid. A $k \times k \times k$ grid results in a QP problem with k^3 unknowns. As with the dense case, all operations on the matrices H and M were performed in parallel on the back-end. Table 4 shows the timing for 2 different problems solved using IPSC-MATLAB. Since

n	$25 \times 25 \times 25$	$26 \times 26 \times 26$
iter.	17	17
time(iPSC) (sec)	393	466
time(real) (sec)	1170	1310
#calls	499	499
#sends	202	202
#recvs	147	147

Table 4: Sparse box-constrained linear least squares problems solved using IPSC-MATLAB on 32 processors

method	$n = 22 \times 22 \times 22$	$n = 24 \times 24 \times 24$
MATLAB	466.4	756.7
IPSC-MATLAB	61.8	73.8
iPSC	15.8	21.5

Table 5: Wall clock time (seconds) per iteration for 2 methods of solving sparse box-constrained least squares problems (32 processors)

all sparse code was implemented as special purpose functions, there was little memory overhead. Therefore, large problems could be solved, unlike the dense case, where symmetry of matrices was not exploited. A C program that ran entirely on the back-end was also written for comparison with IPSC-MATLAB. Table 5 compares the execution time for MATLAB, IPSC-MATLAB, and iPSC, for two problems. As with the dense case, the message-passing overhead is high.

7 Special-purpose operations

Currently, only 8 special purpose operations have been implemented on the hypercube (see sections 5 and 6). Each of these operations has been implemented as a function call. All arguments to these functions are passed by reference (as opposed to MATLAB, which passes variables by value). The functions are:

- `ipsc('S_generate_H_and_g(k, m, H, g, dim)')`
  Computes `H` $\leftarrow$ `A'A` and `g` $\leftarrow$ `-A'b` for the least squares problem on a 2-D or 3-D regular grid. The grid is of size k^{dim}, `m` is the number of rows in the matrix `A`. The matrix `H` is of size k^{dim}. `dim` is the problem dimension ($2 \equiv$ 2-D, $3 \equiv$ 3-D).

- `ipsc('S_matrix_vec_mul(A, x, y)')`
  Performs the operation `y` $\leftarrow$ `A*x`

- `ipsc('S_scale_and_add(A, v1, v2, B)')`
  Performs the operation `B` ← `diag(v1)*A*diag(v1) + diag(v2)`

- `ipsc('S_extract_row(A, i, x)')`
  Performs the operation `x` ← `A(i,:)`

- `ipsc('S_chol(num_calls, A, info, R)')`
  Performs the operation `[R, info] = chol(A)`. `num_calls` is the number of times you expect to perform Cholesky factorization for matrices with the same structure (`num_calls` is set to either 1 or greater than 1). If `num_calls` is greater than 1, then some data structures are retained for later use.

- `ipsc('S_solve(R, x, y)')`
  Performs the operation `y` ← `(R'*R)\x` where `R` is the Cholesky factor obtained above.

- `ipsc('[ox, osig, alpha_add, nstrg] =`
  `startls(s, H, b, strg, x, y, sigma, u, l, count, oval')`
  Performs line search for dense quadratic programming.

- `ipsc('[ox, osig, alpha_add, nstrg] =`
  `S_startls(s, H, b, strg, x, y, sigma, u, l, count, oval')`
  Performs line search for sparse quadratic programming.

In the expressions that have the prefix '`S_`', all matrices are represented using a sparse data structure, whereas all vectors are represented as dense vectors. Sparse matrices cannot be sent to or received from the cube.

8 Conclusion

IPSC-MATLAB appears to be a useful tool for running MATLAB programs on large problems in parallel. *However, its usefulness is limited by the slow communication links from the remote host to the hypercube and by the limited memory on each processor of the hypercube.* When compared with the current generation of fast workstations, it does not appear to have significant advantages; however, with the next generation of hypercubes, IPSC-MATLAB can be a useful tool for solving large problems.

References

[Coleman 92a] Coleman, T. F., and Plassmann, P. E., "A parallel nonlinear least-squares solver: theoretical analysis and numerical results", *SIAM Journal for Scientific and Statistical Computing*, V13, N3, pp. 771-793, May 1992.

[Coleman 92b] Coleman, T. F., and Li, Y., "A reflective Newton method for minimizing a general quadratic function subject to bounds on variables", *in preparation*, 1992.

[Coleman 92c] Coleman, T. F., and Li, Y., "Global and quadratic convergence of reflective Newton methods for nonlinear minimization subject to bounds", *in preparation*, 1992.

[Geijn 91] van de Geijn, R. A., "Massively parallel LINPACK benchmark on the Intel Touchstone DELTA and iPSC/860 systems", *Progress Report*, Department of Computer Sciences, University of Texas, Austin, Texas, 1991.

[Hendrickson 92] Hendrickson, B., "Parallel QR factorization on a hypercube using the torus wrap mapping", *Technical Report SAND91-0874*, Sandia National Laboratories, Albuquerque, NM 87185, Jan 1992.

[Li 89] Li, G. and Coleman, T. F., "A New Method for Solving Triangular Systems on Distributed-Memory Message-Passing Multiprocessors", *SIAM Journal on Scientific and Statistical Computing*, V10, N2, pp. 382-396, 1989.

[MathWorks 90] The MathWorks, Inc., "PRO-MATLAB for Sun workstations", Jan 1990.

[Morgan 84] Morgan, B. J. T., "Elements of Simulation", Chapman and Hall, 1984.

[Pothen 91] Pothen, A. and Sun, C., "A distributed multifrontal algorithm using clique trees", *Technical Report CTC91TR72*, Advanced Computing Research Institute, Cornell Theory Center, Cornell University, August 1991.

[Romine 88] Romine, C. H. and Ortega, J. M., "Parallel Solution of Triangular Systems of Equations", *Parallel Computing*, V6, pp. 109-114, 1988.

A Command Reference

This appendix serves as a reference for all commands implemented in IPSC-MATLAB. The meaning of these commands can be found in [MathWorks 90]. It is not repeated here. The parallel algorithms used are briefly described and any differences in syntax or unimplemented portions of commands and operations are explained.

abs

Synopsis:
`ipsc('y = abs(x)')`

Algorithm:
Each processor independently computes the absolute values of all entries of `x` belonging to it.

any,all

Synopsis:

```
ipsc('y = any(x)')
ipsc('y = all(x)')
```

Algorithm:

If `x` is a vector, each processor independently performs the operation `any()` or `all()` on the copy it has.
If `x` is a matrix, the following operations are performed:

1. Each processor performs the operation on its portion of `x`.
2. Processors $(0, j)$, $j = 0..c-1$ collect contributions from processors (i, j), $i = 0..r-1$ and perform the operation `any()` or `all()` on the components received.
3. Processor (0,0) gathers results from processors $(0, j)$, $j = 0..c-1$.
4. Processor 0 broadcasts the result to all processors.

chol

Synopsis:
`ipsc('[R, p] = chol(A)')`

Notes:
`chol()` computes the Cholesky factorization of a real symmetric matrix, `A`. If the matrix `A` is positive definite, the Cholesky factor `R` is returned and `p` is set to 0. If the matrix `A` is not positive definite, `p` is non-zero and `R` is of size `p-1 by n`, where `n` is the size of `A`. `R` is the partial Cholesky factor of `A`, that is, `R(:,1:p-1)'*R(:,1:p-1) = A(1:p-1,1:p-1)`.

Algorithm:
`chol()` uses the inner product algorithm. The original code was designed to use the lower triangular portion of `A`. To use this code as is, the processor grid had to be reordered in column major order. The inner product algorithm was used because inner products can be performed faster than outer products.
At each step of the algorithm, the messages are tagged with an integer indicating whether the factorization is to be abandoned or not. The reason this needs to be done is that a negative diagonal is detected only by the processor that has that diagonal element. By tagging messages with an integer error code, within a few steps, all processors are informed about the indefiniteness of `A`.
If `R` is a row vector or a scalar, it is made available to all processors after the factorization terminates.

clear

Synopsis:
`ipsc('clear x')`

Notes:
Only 1 variable can be cleared at a time.

Algorithm:
Each processor frees memory used by the variable `x` and eliminates it from the list of variables in the workspace.

diag

Synopsis:

```
ipsc('y = diag(x)')
```

Algorithm:

When `x` is a vector, no message passing is required because each processor has a copy of `x`. Each processor then extracts the elements of `x` it needs and fills them in `y`. When `x` is a matrix, each processor initializes `y` to zero and copies its diagonal entries of `x` into `y`. A global sum of `y` gives the final result.

exp,log,log10,sqrt

Synopsis:

```
ipsc('y = exp(x)')
ipsc('y = log(x)')
ipsc('y = log10(x)')
ipsc('y = sqrt(x)')
```

Algorithm:

Each processor performs the given operation on its entries of `x`. No message passing is required.

max,min

Synopsis:

```
ipsc('y = max(x)')
ipsc('z = max(x,y)')
ipsc('y = min(x)')
ipsc('z = min(x,y)')
```

Algorithm:

When `min()` and `max()` are called with 1 argument, the operations `min()` and `max()` are performed columnwise, similar to the operations `any()` and `all()`.
When `min()` and `max()` are called with 2 arguments, the operations are performed elementwise, similar to the operations `exp()` and `log()`.

norm,fnorm

Synopsis:

```
ipsc('y = norm(x)')
ipsc('y = norm(x,p)')
ipsc('y = fnorm(x)')
```

Notes:

For matrices, 2-norm cannot yet be computed.
The function `fnorm()` computes the Frobenius norm.

Algorithm:

When the argument `x` is a vector, the norm is computed independently by each processor.
When the argument `x` is a matrix:
If the Frobenius norm is desired, each processor computes its contribution to the norm by computing the sum of the squares of all entries. A global sum followed by a square root gives the final result.
If the 1-norm is desired, the maximum column sum is computed as follows:

1. Each processor computes the sum of the absolute values for each of the columns of `x` with it.
2. For each column, a partial sum is performed along the column direction of the processor grid, placing the result in row 0 of the processor grid.
3. Processors $(0, j)$, $j = 0..c-1$ compute the maximum column sum among the columns with them.
4. Processor (0,0) gathers results from processors $(0, j)$, $j = 0..c-1$ and finds the maximum of all numbers received.
5. Processor 0 broadcasts the result to all processors.

If the infinity norm is desired, the maximum row sum is computed in a manner similar to the computation of the maximum column sum.

ones,zeros,eye

Synopsis:

```
ipsc('y = ones(N)')
ipsc('y = ones(M,N)')
ipsc('y = zeros(N)')
ipsc('y = zeros(M,N)')
ipsc('y = eye(N)')
ipsc('y = eye(M,N)')
```

Notes:
The usage `ones(A)`, where `A` is a matrix, is considered obsolete. Use `ones(size(A))` instead.

Algorithm:
Message passing may be needed for determining the dimensions of the resulting matrix. Once the dimensions are determined, each processor can independently fill the matrix with zeros, ones, or the identity matrix.
Let `y` be of size `m`$\times$`n`. The following rules are used to determine `m` and `n`:

1. If the function is called with only 1 argument:
   - If `N` is a scalar, `m = n = N`.
   - If `N` is a vector, `m = N(0)` and `n = N(1)`.
   - If `N` is a matrix, it results in an error.
2. If the function is called with 2 arguments, `m = M(0,0)` and `n = N(0,0)`. If `M` and `N` are matrices, processor 0 broadcasts the (0,0) entries of both matrices to all processors.

qr

Synopsis:

```
ipsc('[Q,R] = qr(A)')
ipsc('[Q,R,E] = qr(A)')
```

Notes:

This qr factorization does not produce `Q` and `R` identical to MATLAB. However, the product `Q*R` will be equal to `A` or `A*E` (depending on whether column pivoting is performed or not).

Algorithm:

See [Coleman 92a] for a row oriented qr factorization algorithm with pivoting and [Hendrickson 92] for a torus-wrapped qr factorization algorithm without pivoting. Here, a torus-wrapped algorithm is used which performs pivoting only if desired. The algorithm uses Householder transformations. Column pivoting, based on the column norms, is performed at every step. The norm of each column is updated as the factorization proceeds. `Q` is formed explicitly using backward accumulation of the Householder vectors, which are temporarily stored in the lower triangular portion of `A`.

rand

Synopsis:

```
ipsc('A = rand(n)')
ipsc('A = rand(m, n)')
ipsc('A = rand(seed, n)')
ipsc('A = rand(normal)')
ipsc('A = rand(uniform)')
```

Notes:
The syntax is modified slightly to avoid having to use strings within strings. `seed`, `normal`, and `uniform` are new keywords and cannot be used as IPSC-MATLAB variable names.

Algorithm:
The size of the resulting matrix `A` is determined in a manner similar to that used in the functions `ones()` and `zeros()`.
Uniform random numbers are generated by calling the math library function `drand48()`. Normal random numbers are generated by the rejection method (see [Morgan 84]). If `A` is a matrix, all processors generate random numbers independently. If `A` is a vector, only processor 0 generates `A`. Processor 0 then broadcasts `A` to all other processors. This ensures that the vector `A` is identical on all processors.

round,fix,ceil,floor,sign,rem

Synopsis:

```
ipsc('y = round(x)')
ipsc('y = fix(x)')
ipsc('y = ceil(x)')
ipsc('y = floor(x)')
ipsc('y = sign(x)')
ipsc('z = rem(x,y)')
```

Algorithm:
Since the operations above are performed componentwise, they can be performed independently on each processor. No message passing is necessary.

size,length

Synopsis:

```
ipsc('y = size(x)')
ipsc('y = length(x)')
```

Algorithm:

Since each processor knows the size of each variable, the above operations can be performed independently with no interprocessor communication.

solve

Synopsis:
`ipsc('x = solve(R, y)')`

Notes:
This command solves a system of equations using forward elimination and back substitution, given an upper triangular matrix `R`.
`x` $\leftarrow$ `(R'*R)` $\backslash$ `y`.
Typically, `R` is the Cholesky factor of a symmetric positive definite matrix.

Algorithm:
The gather-scatter algorithm in [Romine 88] is modified to use the torus wrap mapping. See [Geijn 91] for an efficient implementation of the cyclic algorithm in [Li 89] using a block torus wrap mapping. The block torus wrap mapping can improve performance considerably as fewer messages are required.

sum,prod,cumsum,cumprod

Synopsis:

```
ipsc('y = sum(x)')
ipsc('y = prod(x)')
ipsc('y = cumsum(x)')
ipsc('y = prod(x)')
```

Algorithm:

The operations `sum()` and `prod()` are performed similar to the operations `any()` and `all()`.

The operations `cumsum()` and `cumprod()` are performed as follows:

- If the argument `x` is not a matrix, the processors independently perform the operation since they have a copy of `x`.
- If the argument `x` is a matrix
  1. Processors $(0, j)$, $j = 0..c-1$ initialize row 0 of `y`.
  2. Row i of `y`, $i = 0..m-2$ is sent by processors $(i\%r, j)$, $j = 0..c-1$ to processors $(i+1\%c, j)$, $j = 0..c-1$.
  3. When a processor receives a portion of row i, it performs the desired operation and stores the result in row $i+1$ of `y` before going to step 2.

Thus there is no parallelism in this operation, but the operation is performed on a distributed matrix.

trig

Synopsis:

```
ipsc('y = sin(x)')
ipsc('y = cos(x)')
ipsc('y = tan(x)')
ipsc('y = asin(x)')
ipsc('y = acos(x)')
ipsc('y = atan(x)')
ipsc('z = atan2(y,x)')
```

Algorithm:
These operations are performed componentwise and do not require any interprocessor communication.

+ - * / \ ^ ”

Synopsis:

```
ipsc('C = A + B')
ipsc('C = A - B')
ipsc('C = A * B')
ipsc('C = A / B')
ipsc('C = A \ B')
ipsc('C = A"')
ipsc('C = A .* B')
ipsc('C = A ./ B')
ipsc('C = A .\ B')
ipsc('C = A .^ B')
ipsc('C = A."')
```

Notes:

All operations are similar to those in sequential MATLAB except for the following limitations and differences:

- / and \ can be used only if the divisor is a scalar.
- The operator ^ is not available. If a scalar needs to be raised to another scalar, use `.^` instead.
- To avoid quotes within quotes, the single-quote for transpose is replaced by a double-quote.
- Since complex numbers are not implemented, .” and ” are equivalent.
- For some unknown reason, MATLAB refuses to accept the statement `C = A+-B`. IPSC-MATLAB accepts this statement and gives the expected answer.

Algorithm:

The operations +, -, .*, ./, .\, and `.^` can be performed independently by all processors.

Since / and \ are implemented only when the divisor is a scalar, they too can be performed independently on all processors.

`C = A*B`:

If neither `A` nor `B` is a matrix, the operation can be performed independently by all processors.

If `A` is a scalar and `B` is a matrix, or if `A` is a matrix and `B` is a scalar, the operation can be performed independently by all processors.
If one is a vector and the other is a matrix,

- `C` is initialized to 0.
- Processors independently perform their portion of `A*B`.
- A global sum of the product gives C.

For matrix-matrix multiplication, the outer product algorithm is used. Each processor performs the outer product `A(:,i)*B(i,:)`. Processors that have `A(:,i)` and `B(i,:)` are responsible for broadcasting that information to all processors that require it. Note, because of the torus wrap mapping, the entire row and column is not needed on every processor.
`B = A":`
For scalars and vectors, each processor performs the operation independently.
When `A` is a matrix, at step `i`, row `i` of `A` is assembled on the processor that owns entry `(i,i)` of `A`. Subsequently, this row is broadcast to all processors that need a portion of it.

< <= > >= == = & | ˜

Synopsis:

```
ipsc('C = A < B')
ipsc('C = A <= B')
ipsc('C = A > B')
ipsc('C = A >= B')
ipsc('C = A == B')
ipsc('C = A  = B')
ipsc('C = A & B')
ipsc('C = A | B')
ipsc('C = ~A')
```

Algorithm:
All these operations can be are performed elementwise, and hence, independently.

[ ] ( ) = , . ;

Synopsis:

```
ipsc('C = [A ; B]')
ipsc('C = [A , B]')
ipsc('C = A(i,j)')
ipsc('C = 1.234')
ipsc('C = A + B ; C = 2*C')
ipsc('C = 3*(A + B)')
```

Notes:

Brackets are used as array constructors. To simplify parsing, elements inside the brackets must be separated either by a comma (which places the elements in the same row) or by a semi-colon (which places the elements in different rows). MATLAB also allows spaces to separate elements inside the brackets, but this results in some ambiguity in parsing.

Parentheses are used to indicate operator precedence and also for subscripting arrays. Matrices cannot yet be used as subscripts of other matrices. Subscripted arrays on the left hand side of an assignment statement are yet to be implemented.

Algorithm:

`C = [A ; B]`:

This proceeds in two stages:

- Each processor determines which processor to send its portion of `B` to; sends it, receives a new portion of `B` and stores it in `C`.
- Processors copy `A` into `C` appropriately.

All communication takes place along the column direction of the processor grid.

`C = [A , B]`:

This works similar to the semi-colon except that all communication takes place along the row direction of the processor grid.

Subscripted arrays:

If the array is a vector or a scalar, all processors perform the operation independently.

If a matrix is subscripted by two scalars, the processor that has that entry of the matrix broadcasts it to all other processors.

If the matrix is subscripted by a vector and a scalar, the particular row or column of the matrix is assembled on all processors, after which all processors perform the operation independently.

If the matrix is subscripted by two vectors, each processor has a copy of the vectors used as subscripts. Each processor therefore determines which processor needs some or all of the entries of the matrix it has and sends those entries to it. This results in a completely general communication pattern.

`B = A` :

If `A` is a temporary variable, it is simply renamed to `B` and the old copy of `B`, if any, is destroyed. If `A` is not a temporary variable, `B` is resized, if necessary, and then the assignment is carried out independently by all processors.

:

Synopsis:

```
ipsc('C = i :  j')
ipsc('C = i :  j :  k')
```

Notes:

If `i, j, k` are matrices, only the first entry in each is used.
The precedence rules may not be the same as those in MATLAB. Use of more than 3 colons in a single expression is discouraged.

Algorithm:

Since the colons generate vectors, as long as the first entry in `i, j, k` is made available to all processors, they can perform the operation independently.

B Lexical Analyzer

This section lists the lexical analyzer used for extracting tokens from commands sent to IPSC-MATLAB.

```
D                       [0-9]
L                       [a-zA-Z_]
E                       [Ee][+-]?{D}+

%{
#include <stdio.h>
#include "parser.h"
#include "y.tab.h"

#define my_return(x) strcpy(yylval.val, yytext); \
                     strcat(comm_line, " "); \
                     strcat(comm_line, yytext); return(x)
#undef input
extern int input();
#undef unput
extern int unput();

%}

%%

"clear"                 { my_return(CLEAR); }
"_M_RECV"               { my_return(RECV); }
"_M_SEND"               { my_return(SEND); }
"zeros"                 { my_return(ZEROS); }
"ones"                  { my_return(ONES); }
"diag"                  { my_return(DIAG); }
"chol"                  { my_return(CHOL); }
"asin"                  { my_return(ASIN); }
"acos"                  { my_return(ACOS); }
"atan"                  { my_return(ATAN); }
"sin"                   { my_return(SIN); }
"cos"                   { my_return(COS); }
"tan"                   { my_return(TAN); }
"log10"                 { my_return(LOG10); }
"log"                   { my_return(LOG); }
```

```
"exp"                { my_return(EXP); }
"sqrt"               { my_return(SQRT); }
"ceil"               { my_return(CEIL); }
"floor"              { my_return(FLOOR); }
"round"              { my_return(ROUND); }
"fix"                { my_return(FIX); }
"sign"               { my_return(SIGN); }
"abs"                { my_return(ABS); }
"rem"                { my_return(REM); }
"atan2"              { my_return(ATAN2); }
"min"                { my_return(MIN); }
"max"                { my_return(MAX); }
"any"                { my_return(ANY); }
"all"                { my_return(ALL); }
"sum"                { my_return(SUM); }
"prod"               { my_return(PROD); }
"size"               { my_return(SIZE); }
"length"             { my_return(LENGTH); }
"norm"               { my_return(NORM); }
"inf"                { my_return(INF); }
"eye"                { my_return(EYE); }
"cumsum"             { my_return(CUMSUM); }
"cumprod"            { my_return(CUMPROD); }
"fnorm"              { my_return(FNORM); }
"rand"               { my_return(RAND); }
"seed"               { my_return(SEED); }
"uniform"            { my_return(UNIFORM); }
"normal"             { my_return(NORMAL); }
"solve"              { my_return(SOLVE); }
"qr"                 { my_return(QR); }
"startls"            { my_return(STARTLS); }
"S_generate_H_and_g" { my_return(S_GENERATE_H_AND_G); }
"S_matrix_vec_mul"   { my_return(S_MATRIX_VEC_MUL); }
"S_scale_and_add"    { my_return(S_SCALE_AND_ADD); }
"S_startls"          { my_return(S_STARTLS); }
"S_extract_row"      { my_return(S_EXTRACT_ROW); }
"S_chol"             { my_return(S_CHOL); }
"S_solve"            { my_return(S_SOLVE); }

{L}({L}|{D})*        { my_return(IDENTIFIER); }
```

```
{D}+({E})?              { my_return(CONSTANT); }
{D}*"."{D}+({E})?       { my_return(CONSTANT); }
{D}+"."{D}*({E})?       { my_return(CONSTANT); }

"<="                    { my_return(LE_OP); }
">="                    { my_return(GE_OP); }
"=="                    { my_return(EQ_OP); }
"~="                    { my_return(NE_OP); }
".*"                    { my_return(DOT_STAR); }
".^"                    { my_return(DOT_CARET); }
"./"                    { my_return(DOT_SLASH); }
".\\"                   { my_return(DOT_BACKSLASH); }
"\\"                    { my_return(BACKSLASH); }
".\""                   { my_return(DOT_QUOTE); }
"\""                    { my_return(QUOTE); }
";"                     { my_return(';'); }
","                     { my_return(','); }
":"                     { my_return(':'); }
"="                     { my_return('='); }
"("                     { my_return('('); }
")"                     { my_return(')'); }
"["                     { my_return('['); }
"]"                     { my_return(']'); }
"&"                     { my_return('&'); }
"!"                     { my_return('!'); }
"~"                     { my_return('~'); }
"-"                     { my_return('-'); }
"+"                     { my_return('+'); }
"*"                     { my_return('*'); }
"/"                     { my_return('/'); }
"<"                     { my_return('<'); }
">"                     { my_return('>'); }
"^"                     { my_return('^'); }
"|"                     { my_return('|'); }
"#"                     { my_return('#'); }
\n                      { my_return('\n'); }

[ \t\v\f]               { }
.                       { my_return(GARBAGE); }

%%
```

```
int yycolumn = 0;

yywrap()
{
    return(1);
}

int input()
{
    yylineno = 1;
    yycolumn = command_index;
    yytchar = command[command_index];
    if (command_index == command_length)
    {
        yytchar = EOF;
        command_index++;
        return 0;      /* EOF */
    }

    command_index++;
    return (int) command[command_index-1];
}

int unput(c)
int c;
{
    command_index--;
}
```

C Grammar

This section lists the yacc grammar used for parsing commands sent to IPSC-MATLAB. It should help the reader unambiguously decide what syntax is legal and what isn't. Unfortunately, not all syntactically correct constructs are executed; some are yet to be implemented. When run, IPSC-MATLAB will produce semantic errors under such circumstances. See the reference pages for further details on what is yet to be implemented.
The actions taken while parsing the input are not shown.

```
%{
#include "parser.h"

extern char yytext[];
extern int yyleng;

%}
%token <val> CLEAR CONSTANT LE_OP GE_OP EQ_OP NE_OP DOT_STAR
%token <val> DOT_CARET DOT_SLASH DOT_BACKSLASH DOT_QUOTE QUOTE
%token <val> IDENTIFIER GARBAGE RECV SEND ZEROS ONES DIAG
%token <val> BACKSLASH CHOL CEIL FLOOR FIX ROUND SIN COS TAN
%token <val> ASIN ACOS ATAN LOG10 LOG EXP SQRT SIGN ABS REM ATAN2
%token <val> MIN MAX ANY ALL SUM PROD SIZE LENGTH NORM INF EYE
%token <val> CUMSUM CUMPROD FNORM RAND SEED UNIFORM NORMAL SOLVE
%token <val> STARTLS QR S_GENERATE_H_AND_G S_MATRIX_VEC_MUL
%token <val> S_SCALE_AND_ADD S_STARTLS S_EXTRACT_ROW S_CHOL S_SOLVE
%token <val> '*' '/' '+' '-' '<' '>' '&' '|' '=' '~' ':' ',' ';'

%start program
%%

mtx_cols
        : expr
        | mtx_cols ',' expr
        ;

mtx_rows
        : mtx_cols
        | mtx_rows ';' mtx_cols
        ;
```

```
array_constructor
        : '[' mtx_rows ']'
        ;

arg
        : IDENTIFIER
        ;

subscript
        : expr
        ;

subscripted_array
        : IDENTIFIER
        | IDENTIFIER '(' subscript ')'
        | IDENTIFIER '(' subscript ',' subscript ')'
        ;

primary_expr
        : CONSTANT
        | '(' expr ')'
        | subscripted_array
        | array_constructor
        | function_call
        ;

unary_expr
        : primary_expr QUOTE
        | primary_expr DOT_QUOTE
        | '+' primary_expr
        | '-' primary_expr
        | '~' primary_expr
        | primary_expr
        ;

multiplicative_expr
        : unary_expr
        | multiplicative_expr '*' unary_expr
        | multiplicative_expr '/' unary_expr
        | multiplicative_expr BACKSLASH unary_expr
        | multiplicative_expr DOT_STAR unary_expr
```

```
        | multiplicative_expr DOT_SLASH unary_expr
        | multiplicative_expr DOT_BACKSLASH unary_expr
        | multiplicative_expr DOT_CARET unary_expr
        ;

additive_expr
        : multiplicative_expr
        | additive_expr '+' multiplicative_expr
        | additive_expr '-' multiplicative_expr
        ;

relational_expr
        : additive_expr
        | relational_expr '<' additive_expr
        | relational_expr '>' additive_expr
        | relational_expr LE_OP additive_expr
        | relational_expr GE_OP additive_expr
        ;

equality_expr
        : relational_expr
        | equality_expr EQ_OP relational_expr
        | equality_expr NE_OP relational_expr
        ;

and_expr
        : equality_expr
        | and_expr '&' equality_expr
        ;

inclusive_or_expr
        : and_expr
        | inclusive_or_expr '|' and_expr
        ;

index_expr
        : inclusive_or_expr
        | index_expr ':' inclusive_or_expr
        ;

function_call
```

```
: ONES '(' expr ')'
| ONES '(' expr ',' expr ')'
| ZEROS '(' expr ')'
| ZEROS '(' expr ',' expr ')'
| RAND '(' expr ')'
| RAND '(' expr ',' expr ')'
| EYE '(' expr ')'
| EYE '(' expr ',' expr ')'
| DIAG '(' expr ')'
| DIAG '(' expr ',' expr ')'
| SIN '(' expr ')'
| COS '(' expr ')'
| TAN '(' expr ')'
| ASIN '(' expr ')'
| ACOS '(' expr ')'
| ATAN '(' expr ')'
| LOG10 '(' expr ')'
| LOG '(' expr ')'
| EXP '(' expr ')'
| SQRT '(' expr ')'
| CEIL '(' expr ')'
| FIX '(' expr ')'
| FLOOR '(' expr ')'
| ROUND '(' expr ')'
| SIGN '(' expr ')'
| ABS '(' expr ')'
| REM '(' expr ',' expr ')'
| ATAN2 '(' expr ',' expr ')'
| MIN '(' expr ',' expr ')'
| MAX '(' expr ',' expr ')'
| ANY '(' expr ')'
| ALL '(' expr ')'
| SUM '(' expr ')'
| PROD '(' expr ')'
| MIN '(' expr ')'
| MAX '(' expr ')'
| SIZE '(' expr ')'
| LENGTH '(' expr ')'
| NORM '(' expr ')'
| NORM '(' expr ',' expr ')'
| NORM '(' expr ',' INF ')'
```

```
        | NORM '(' expr ',' '-' INF ')'
        | FNORM '(' expr ')'
        | CUMSUM '(' expr ')'
        | CUMPROD '(' expr ')'
        | SOLVE '(' expr ',' expr ')'
        ;

multiple_retval_function
        : '[' subscripted_array ',' subscripted_array ']'
          '=' CHOL '(' expr ')'
        | '[' subscripted_array ',' subscripted_array ']'
          '=' QR '(' expr ')'
        | '[' subscripted_array ',' subscripted_array ','
           subscripted_array ']' '=' QR '(' expr ')'
        | '[' subscripted_array ',' subscripted_array ','
          subscripted_array ',' subscripted_array ']' '='
          STARTLS '(' expr ',' expr ',' expr ',' expr ','
          expr ',' expr ',' expr ',' expr ',' expr ',' expr
          ',' expr ')'
        | '[' subscripted_array ',' subscripted_array ','
          subscripted_array ',' subscripted_array ']' '='
          S_STARTLS '(' expr ',' expr ',' expr ',' expr ','
          expr ',' expr ',' expr ',' expr ',' expr ',' expr
          ',' expr ')'
        ;

expr
        : index_expr
        ;

assignment_expr
        : subscripted_array '=' expr
        | multiple_retval_function
        ;

statement
        : assignment_expr termin
        | CLEAR IDENTIFIER termin
        | RECV IDENTIFIER termin
        | SEND IDENTIFIER CONSTANT CONSTANT termin
        | RAND '(' SEED ',' expr ')' termin
```

```
        | RAND '(' UNIFORM ')' termin
        | RAND '(' NORMAL ')' termin
        | S_GENERATE_H_AND_G '(' arg ',' arg ',' arg ','
          arg ',' arg ')'
        | S_MATRIX_VEC_MUL '(' arg ',' arg ',' arg ')'
        | S_SCALE_AND_ADD '(' arg ',' arg ',' arg ',' arg ')'
        | S_EXTRACT_ROW '(' arg ',' arg ',' arg ')'
        | S_CHOL '(' arg ',' arg ',' arg ',' arg ')'
        | S_SOLVE '(' arg ',' arg ',' arg ')'
        | termin
        ;

statement_list
        : statement
        | statement_list statement
        ;

program
        : statement_list
        ;

termin
        : ';'
        | '\n'
        ;
%%
```