pMatlab v2.0 Function Reference

Hahn Kim, Nadya Travinin, Jeremy Kepner
{hgk, nt, kepner}@ll.mit.edu

Table of Contents

pMatlab **6**
pRUN 6
Np 6
Pid 7
pMATLAB *Deprecated* 7
pMatlab_Init *Deprecated* 7
pMatlab_Finalize *Deprecated* 8
pMatlab_ver 8
MPI_Abort 8
MatMPI_Delete_all 9
MPI_Run *Deprecated* 9
SendMsg 10
RecvMsg 10
Distributed matrices and matrix manipulation **11**
Elementary distributed matrices 11
map/map 11
map/zeros 15
map/ones 16
map/rand 17
Basic array information 18
dmat/size 18
dmat/ndims 18
dmat/display 19
map/display 19
Distributed array information 20
dmat/global_block_range 20
dmat/global_block_ranges 21

This work is sponsored by the Department of the Air Force under Air Force contract FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government. MATLAB® is a registered trademark of The Mathworks, Inc.

dmat/global_ind 22
dmat/global_inds 23
dmat/global_range 24
dmat/global_ranges 26
map/inmap 28
Matrix manipulation 29
dmat/find 29
Distributed matrix manipulation 30
dmat/agg 30
dmat/agg_all 31
dmat/local 31
dmat/put_local 32
dmat/synch 33
dmat/sync2 33

.... 34
remap 34
dmat/subsasgn 34
dmat/subsref 36
map/subsasgn 37
map/subsref 37
transpose_grid 38
Elementary math functions 39
Trigonometric 39
dmat/sin, dmat/cos, dmat/tan, dmat/sec, dmat/csc, dmat/cot 39
dmat/sind, dmat/cosd, dmat/tand, dmat/secd, dmat/cscd, dmat/cotd 40
dmat/sinh, dmat/cosh, dmat/tanh, dmat/sech, dmat/csch, dmat/coth 41
dmat/asin, dmat/acos, dmat/atan, dmat/asec, dmat/acsc, dmat/acot 42
dmat/asind, dmat/acosd, dmat/atand, dmat/asecd, dmat/acscd, dmat/acotd 43
dmat/asinh, dmat/acosh, dmat/atanh, dmat/asech, dmat/acsch, dmat/acoth 44
Exponential 45
dmat/exp 45
dmat/expm1 45

dmat/log 45
dmat/log1p 46
dmat/log10 46
dmat/log2 46
dmat/pow2 47
dmat/realpow 47
dmat/reallog 47
dmat/realsqrt 48
dmat/sqrt 48
Complex 49
dmat/abs 49
dmat/angle 49
dmat/complex 49
dmat/conj 50
dmat/imag 50
dmat/real 50
Rounding and remainder 51
dmat/fix 51
dmat/floor 51
dmat/ceil 51
dmat/round 52
dmat/sign 52
Operators and special characters 53
dmat/plus 53
dmat/minus 53
dmat/mtimes 54
dmat/times 54
dmat/power 55
dmat/ldivide 55
dmat/rdivide 56
dmat/eq 56
map/eq 57

dmat/gt 57
map/ne 57
dmat/transpose 58
dmat/transpose 58
Sparse matrices 59
map/sparse 59
map/spalloc 60
dmat/sparse 60
Data analysis and Fourier transforms 61
dmat/conv2 61
dmat/fft 61
Data types and structures 63
dmat/double 63
dmat/single 63
dmat/uint8, dmat/uint16, dmat/uint32, dmat/uint64 64
dmat/int8, dmat/int16, dmat/int32, dmat/int64 64
Signal Processing Toolbox 65
dmat/dct 65
dmat/idct 65
Index 66

MATLAB® is a registered trademark of The Mathworks, Inc.

Introduction

This document is meant to be a reference for functions that are of use to pMatlab *users*, i.e. application developers There are a number of additional functions included in pMatlab, but those functions are used internally by pMatlab and should **not** be called by pMatlab applications.

The functions described in this reference are divided into sections that approximately match Mathwork's own categorization of MATLAB® functions. Most sections describe overloaded MATLAB functions; some sections contain additional functions that are unique to pMatlab, but are related to the overloaded MATLAB functions in that section.

- **pMatlab** describes general pMatlab functions required by all pMatlab applications.
- **Distributed matrices and matrix manipulation** describes functions related to creating and obtaining information about distributed matrices. Because this section contains a large number of overloaded MATLAB functions and a number of new pMatlab functions, it is further divided into subsections.
  - **Elementary distributed matrices** describes functions related to the creation of distributed matrices.
  - **Basic array information** and **Distributed array information** describe functions that obtain information about a distributed matrix.
  - **Matrix manipulation** and **Distributed matrix manipulation** describe functions used to manipulate distributed matrices.
- **Elementary math functions**, **Operators and special characters**, **Sparse matrices**, and **Data analysis and Fourier transforms** describe functions and operators that have been overloaded in pMatlab.

Most functions are *class* functions. Consequently, the names of these functions have been prefaced with the name of class they are a part of. For example, the `fft` function for the `dmat` class is listed as `dmat/fft`. Help for every function can be obtained from the MATLAB command prompt by running `help class/function`. For example, to get the help documentation for the `fft` function overloaded for `dmat`, run:

```
help dmat/fft.
```

For some overloaded functions, it may be useful to refer to the help documentation for the original MATLAB function by running `help function` at the MATLAB command prompt. To get the help documentation for the original MATLAB `fft` function, run:

```
help fft.
```

pMatlab

pRUN{ XE "pRUN" }

Used to run a pMatlab v2 program.

Syntax

```
eval(pRUN(mfile, Ncpus, cpus))
```

Description

`mfile` is a string that contains the name of the pMatlab program to be launched, without the `.m` suffix.

`Ncpus` is an integer that specifies the number of processors to launch `mfile` onto.

`cpus` specifies what machines to launch `mfile` onto:

- `cpus = {};` Run all MATLAB processes on the local machine.
- `cpus = {'machine1' 'machine2' ...};` Specify names of machines on which to run. To run interactively, `machine1` must be the name of local machine.
- `cpus = {'machine1:dir1' 'machine2:dir2' ...};` Specify machines names and which directory to use for communication on each machine. Directories must be visible to both machines, i.e. crossmounted. Directories should be located on the local disk of their respective machines.
- `cpus = {'machine1:type' 'machine2:type'};` Specify machine names and the type of each machine. `type` can be either `'unix'` or `'pc'`. Default is `'unix'` (can be changed in `MatMPI_Comm_settings.m`)
- `cpus = {'machine1:type:dir1' 'machine2:type:dir2'};` Specify machine names, communication directories, and the type of each machine.

Np{ XE "Np" }

Returns the total number of Matlab instances currently running (i.e. Ncpus).

Syntax

```
Np
```

Description

`Np` is an accessor function to the `pMATLAB` global variable and returns the total number of Matlab instances launched by `pRUN`.

Pid{ XE "Pid" }

Returns the Pid of the current instance of Matlab.

Syntax

```
Pid
```

Description

`Pid` is an accessor function to the `pMATLAB` global variable and returns the current instance of Matlab. `Pid` ranges from `0` to `Np`-1.

---

*pMATLAB{ XE "pMATLAB" } *Deprecated**

Data structure created by `pRUN`. Contains information necessary for communication. See `pMatlab_Init` for more details.

---

*pMatlab_Init{ XE "pMatlab_Init" } *Deprecated**

Called by `pRUN` to initialize pMatlab environment.

Syntax

```
pMatlab_Init
```

Description

Initializes variables required by the pMatlab library, such as number of processors, current processor's rank and which processor is the leader. All of the variables necessary for communication are stored in the `pMATLAB` structure.

Fields of the `pMATLAB` structure:

- `comm` - contains the MatlabMPI communicator
- `comm_size` - size of communicator, i.e. number of processors
- `my_rank` - rank of the local processor
- `leader` - indicates which rank is the leader, by default set to 0
- `pList` - list of ranks of participating processors
- `tag` - current message tag
- `tag_num` - number of messages sent; synchronized across all processors in `pList`

MATLAB® is a registered trademark of The Mathworks, Inc.

Fields to be potentially added in the future:

- `num_tasks` - number of tasks (scopes) created from the beginning of the program
- `curr_task` - current task (scope)
- `scopes` - contains a cell array of communication scopes; each entry is a struct with the current fields of the `pMATLAB` structure plus the `task_num` field

---

*pMatlab_Finalize{ XE "pMatlab_Finalize" } *Deprecated**

Called by `pRUN` to terminate pMatlab environment.

Syntax

```
pMatlab_Finalize
```

Description

Terminates pMatlab environment, i.e. exits non-leader MATLAB processes. This ensures that MATLAB processes are not orphaned on remote machines while leaving the leader process running.

---

pMatlab_ver{ XE "pMatlab_ver" }

Display version number for pMatlab

Syntax

```
v = pMatlab_ver
```

Description

`v = pMatlab_ver` returns a string `v` containing the pMatlab version.

MPI_Abort{ XE "MPI_Abort" }

Aborts any currently running pMatlab or MatlabMPI program and blocks returning until all processes have ended. Automatically prepended to pMatlab scripts by `pRUN`.

Syntax

```
MPI_Abort
```

Description

Will abort any currently running pMatlab/MatlabMPI program by looking for leftover MATLAB processes and killing them. Cannot be used after `MatMPI_Delete_all`. Must be run in the directory from which the pMatlab/MatlabMPI programs was launched.

MatMPI_Delete_all{ XE "MatMPI_Delete_all" }

Deletes the MatMPI directory and its contents. Automatically prepended to pMatlab scripts by `pRUN`.

Syntax

```
MatMPI_Delete_all
```

---

*MPI_Run{ XE "MPI_Run" } *Deprecated**

Called by `pRUN` to launches a pMatlab v2.0 program. Used directly to run a pMatlab v1.0or MatlabMPI program.

Syntax

```
eval(MPI_Run(mfile, Ncpus, cpus))
```

Description

`mfile` is a string that contains the name of the pMatlab/MatlabMPI program to be launched, without the `.m` suffix.

`Ncpus` is an integer that specifies the number of processors to launch `mfile` onto.

`cpus` specifies what machines to launch `mfile` onto:

- `cpus = {};` Run all MATLAB processes on the local machine.
- `cpus = {'machine1' 'machine2' ...};` Specify names of machines on which to run. To run interactively, `machine1` must be the name of local machine.
- `cpus = {'machine1:dir1' 'machine2:dir2' ...};` Specify machines names and which directory to use for communication on each machine. Directories must be visible to both machines, i.e. crossmounted. Directories should be located on the local disk of their respective machines.
- `cpus = {'machine1:type' 'machine2:type'};` Specify machine names and the type of each machine. `type` can be either `'unix'` or `'pc'`. Default is `'unix'` (can be changed in `MatMPI_Comm_settings.m`)

`cpus = {'machine1:type:dir1' 'machine2:type:dir2'};` Specify machine names, communication directories, and the type of each machine.

SendMsg{ XE "SendMsg" }

Sends a set of variables from the current Pid to the Pid denoted by dest.

Syntax

```
SendMsg(dest,tag,var1,var2,...)
```

Description

`dest` is the numeric Pid of the destination.

`tag` is an integer or string used to differentiate multiple message being sent between the same Pids..

`var1,var2,...` are the variables to be sent.

RecvMsg{ XE "RecvMsg" }

Receives a set of variables sent from the Pid denoted by source.

Syntax

```
[var1,var2,...] = SendMsg(source,tag)
```

Description

`source` is the numeric Pid of the sender.

`tag` is an integer or string used to differentiate multiple message being sent between the same Pids..

`var1,var2,...` are the variables to be received.

Distributed matrices and matrix manipulation

Elementary distributed matrices

---

map/map{ XE "map:map" }

Map class constructor.

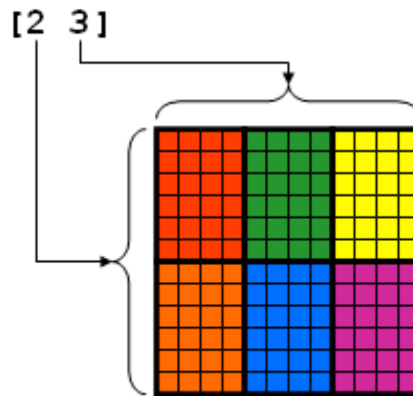
Syntax

```
p = map(GRID_SPEC, DIST_SPEC, PROC_LIST)
p = map(GRID_SPEC, DIST_SPEC, PROC_LIST, OVERLAP_SPEC)
```

Description

`map(GRID_SPEC, DIST_SPEC, PROC_LIST, OVERLAP_SPEC)` constructs a map object to be used as an input to a `dmat` constructor.

- `GRID_SPEC`: Vector of integers specifying how each dimension of a `dmat` is broken up. For example, if `GRID_SPEC = [2 3]`, the first dimension is broken up between 2 processors and the second dimension is broken up between 3 processors. The following figure illustrates how this grid example would break up a `dmat` given 6 processors using a block distribution.

  The length of `GRID_SPEC` can be 2, 3, or 4 and must match the number of dimensions in the `dmat`.

- `DIST_SPEC`: Array of structures with two possible fields, `dist` and `b_size`, specifying the `dmat` distribution.

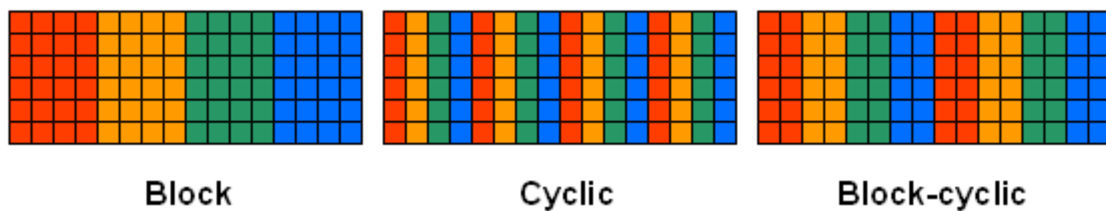
  `DIST_SPEC.dist` is a string specifying the type of data distribution the `dmat` should use. Each entry in the array must have the `dist` field defined. The `dist` field can have three possible values:

  - `'b'`: block
  - `'c'`: cyclic
  - `'bc'`: block-cyclic

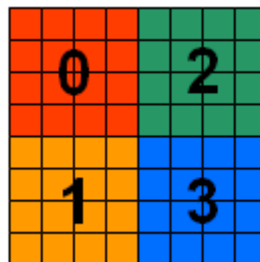
Setting `DIST_SPEC` to `{}` uses block distribution for all dimensions.

`DIST_SPEC.b_size` specifies the block size for block-cyclic distributions. If `DIST_SPEC.dist` is set to `'bc'`, then `DIST_SPEC.b_size` must also be defined. If `DIST_SPEC.dist` is set to `'b'` or `'c'`, then `DIST_SPEC.b_size` does not have to be defined.

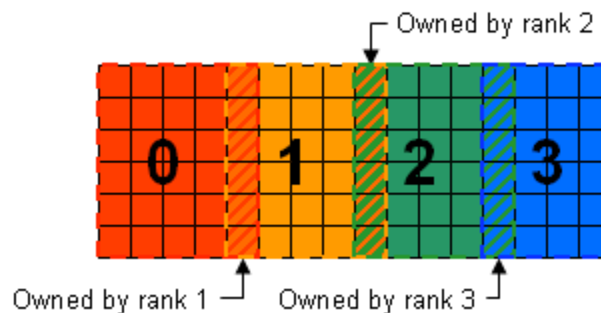
The following figure shows an example of the same `dmat` distributed over 4 processors using each of the three types of data distributions:



- `PROC_LIST`: Array of processor ranks specifying on which ranks the object should be distributed. Ranks are assigned column-wise (top-down, then left-right) to grid locations in sequential order.



- `OVERLAP_SPEC`: Optional. Vector of integers specifying amount of overlap between processors for each dimension. The following figure shows an example of a `dmat` distributed across four processors with 1 column of overlap between adjacent processors.



The length of `OVERLAP_SPEC` can be 2, 3, or 4 and must match the number of dimensions in the `dmat`. Only block distributions can have overlap.

`map` returns a data structure `p` which contains the following fields:

- `DIM`: the number of dimensions of the map (must equal the dimension of the `dmat`)
- `PROC_LIST`: the list of processor ranks on which the object should be distributed
- `DIST_SPEC`: the distribution specification for each dimension
- `GRID`: array of length `DIM` specifying how the object should be distributed

Examples

2D map, 2x2 grid, block-cyclic along rows and columns, block size 2 along rows, block size 3 along columns:

```
grid1 = [2 2];            % 2x2 grid
dist1(1).dist = 'bc';     % block-cyclic along dim 1 (rows)
dist1(1).b_size = 2;      % block size 2 along dim 1 (rows)
dist1(2).dist = 'bc';     % block-cyclic along dim 2 (columns)
dist1(2).b_size = 3;      % block size 3 along dim 2 (columns)
proc1 = [0:3];            % list of ranks 0 through 3
map1 = map(grid1, dist1, proc1);
```

2D map, 2x3 grid, cyclic along both rows and columns:

```
grid2 = [2 3];            % 2x3 grid
dist2(1).dist = 'c';      % cyclic along dim 1 (rows)
dist2(2).dist = 'c';      % cyclic along dim 2 (columns)
proc2 = [0:5];            % list of ranks 0 through 5
map2 = map(grid2, dist2, proc2);
```

2D map, 1x2 grid, block along rows, cyclic along columns:

```
grid3 = [1 2];            % 1x2 grid
dist3(1).dist = 'b';      % block along dim 1 (rows)
dist3(2).dist = 'c';      % cyclic along dim 2 (columns)
proc3 = [0:1];            % list of ranks 0 and 1
map3 = map(grid3, dist3, proc3);
```

3D map, 2x3x2 grid, block-cyclic along rows and columns with block size 2, cyclic along third dimension:

```
grid4 = [2 3 2];          % 2x3x2 grid
dist4(1).dist = 'bc';     % block-cyclic along dim 1 (rows)
dist4(1).b_size = 2;      % block size 2 along dim 1 (rows)
dist4(2).dist = 'bc';     % block-cyclic along dim 2 (columns)
dist4(2).b_size = 2;      % block size 2 along dim 2 (columns)
dist4(3).dist = 'c';      % cyclic along dim 3
proc4 = [0:11];           % list of ranks 0 through 12
map4 = map(grid4, dist4, proc4);
```

2D map, 1x4 grid, block along rows, cyclic along columns:

```
grid5 = [1 4];            % 1x4 grid
dist5(1).dist = 'b';      % block along dim 1 (rows)
dist5(2).dist = 'c';      % cyclic along dim 2 (columns)
proc5 = [0:3];            % list of ranks 0 through 3
map5 = map(grid5, dist5, proc5);
```

2D map, block along both dimensions, overlap in the column dimension of size 1 (1 column overlap):

```
grid6 = [2 2];            % 2x2 grid
dist6 = {};               % block along all dimensions
proc6 = [0 1];            % list of ranks 0 and 1
overlap6 = [0 1];         % overlap of 0 along dim 1 (rows)
                          % overlap of 1 along dim 2 (columns)
map6 = map(grid6, dist6, proc6, overlap6);
```

These examples show only how to create `map` objects. Refer to `dmat/ones`, `dmat/rand`, and `dmat/zeros` on how to create `dmat` objects using `map` objects.

map/zeros{ XE "map:zeros" }

Create a `dmat` of zeros.

Syntax

```
Y = zeros(N, P)
Y = zeros(M, N, P)
Y = zeros(M, N, Q, P)
Y = zeros(M, N, Q, R, P)
```

Description

`zeros(N, P)` returns an `N`-by-`N` `dmat` of zeros mapped according to the map specified by `P`.

`zeros(M, N, P)` returns an `M`-by-`N` `dmat` of zeros mapped according to the map specified by `P`.

`zeros(M, N, Q, P)` returns an `M`-by-`N`-by-`Q` `dmat` of zeros mapped according to the map specified by `P`.

`zeros(M, N, Q, R, P)` returns an `M`-by-`N`-by-`Q`-by-`R` `dmat` of zeros mapped according to the map specified by `P`.

Remarks

Dimension of the `dmat` must be consistent with the dimension of the map's grid.

map/ones{ XE "map:ones" }

Create a `dmat` of all ones

Syntax

```
Y = ones(N, P)
Y = ones(M, N, P)
Y = ones(M, N, Q, P)
Y = ones(M, N, Q, R, P)
```

Description

`ones(N, P)` returns an `N`-by-`N` `dmat` of ones mapped according to the map specified by P.

`ones(M, N, P)` returns an `M`-by-`N` `dmat` of ones mapped according to the map specified by `P`.

`ones(M, N, Q, P)` returns an `M`-by-`N`-by-`Q` `dmat` of ones mapped according to the map specified by `P`.

`ones(M, N, Q, R, P)` returns an `M`-by-`N`-by-`Q`-by-`R` `dmat` of ones mapped according to the map specified by `P`.

Remarks

Dimension of the `dmat` must be consistent with the dimension of the map's grid.

map/rand{ XE "map:rand" }

Create a `dmat` of uniformly distributed random numbers.

Syntax

```
Y = rand(N, P)
Y = rand(M, N, P)
Y = rand(M, N, Q, P)
Y = rand(M, N, Q, R, P)
```

Description

The `rand` function generates `dmats` of random numbers between 0 and 1 distributed uniformly.

`rand(N, P)` returns `N`-by-`N` `dmat` of random numbers mapped according to the map specified by `P`.

`rand(M, N, P)` returns `M`-by-`N` `dmat` of random numbers mapped according to the map specified by `P`.

`rand(M, N, Q, P)` returns an `M`-by-`N`-by-`Q` `dmat` of random numbers mapped according to the map specified by `P`.

`rand(M, N, Q, R, P)` returns an `M`-by-`N`-by-`Q`-by-`R` `dmat` of random numbers mapped according to the map specified by `P`.

Remarks

Dimension of the `dmat` must be consistent with the dimension of the map's grid.

Calls the MATLAB `rand` function to create each local part of the `dmat`. Thus, the resulting array will **not** be the same as a `double` random array of the same dimensions.

Basic array information

---

dmat/size{ XE "dmat:size" }

Size of the `dmat`.

Syntax

```
d = size(X)
[m, n] = size(X)
[d1, d2, d3, ..., dn] = size(X)
```

Description

`d = size(X)` returns the size of each dimension of `dmat X` in vector `d`.

`[m,n] = size(X)` returns the size of `dmat X` in separate variables `m` and `n`.

`[d1,d2,d3,...,dn] = size(X)` returns the sizes of each dimension of `X` in separate variables.

Remarks

If `A = zeros(m, n, q, p1)` and `B = zeros(m, n, q, p2)`, where `p1` and `p2` are different maps, `size(A)` and `size(B)` return the same results.

---

dmat/ndims{ XE "dmat:ndims" }

Number of dimension of the dmat.

Syntax

```
n = ndims(A)
```

Description

`n = ndims(A)` returns the number of dimensions in the dmat `A`. The number of dimensions in a dmat is always greater than or equal to 2.

Remarks

`ndims(A)` is `length(size(A))`.

dmat/display{ XE "dmat:display" }

Display `dmat`.

Syntax

```
display(D)
```

Description

`display(D)` aggregates the `D` onto the leader process and displays the entire contents of `D` on the leader process. On remote processes, `display(D)` displays only the local portion of `D`.

`display(D)` is also called for `D` when a semicolon is not used to terminate a statement.

Remarks

Note that `display` incurs communication overhead to aggregate `D` onto the leader processor.

map/display{ XE "map:display" }

Display map object.

Syntax

```
display(M)
```

Description

`display(M)` displays the contents of the map object.

Remarks

`display(M)` is also called for `M` when a semicolon is not used to terminate a statement.

Distributed array information

---

dmat/global_block_range{ XE "dmat:global_block_range" }

Returns the ranges of global indices local to the current processor for a given `dmat`.

Syntax

```
I = global_block_range(D, DIM)
[I1, I2, ..., IN] = global_block_range(D)
```

Description

`I = global_block_range(D, DIM)` Returns the global index range of the `dmat` `D` local to the current processor in the specified dimension, `DIM`.

`[I1, I2, ..., IN] = global_block_range(D)` Returns the global index range of the `dmat` `D` local to the current processor for all `N` dimensions of `D`.

The global index range for each dimension is returned as a 2-element vector. The first element in the vector represents the starting global index and the second element represents the ending index.

Examples

Let `Ncpus` be 4:

```
P = map([1 Ncpus], {}, 0:Ncpus-1);
D = zeros(100, 100, P);
[I1, I2] = global_block_range(D);
```

For each rank, `I1` contains:

Rank	`I1(1)`	`I1(2)`
0	1	50
1	51	100
2	1	50
3	51	100

For each rank, `I2` contains:

Rank	`I2(1)`	`I2(2)`
0	1	50
1	1	50
2	51	100
3	51	100

dmat/global_block_ranges{ XE "dmat:global_block_ranges" }

Returns the ranges of global indices for all processors in the map of `dmat D`.

Syntax

```
I = global_block_ranges(D, DIM)
[I1, I2, ..., IN] = global_block_ranges(D)
```

Description

`I = global_block_ranges(D, DIM)` Returns the global index ranges of the `dmat D` for all processors in the specified dimension, `DIM`.

`[I1, I2, ..., IN] = global_block_ranges(D)` Returns the global index range of the `dmat D` for all processors in all dimensions of `D`.

For each dimension, the indices are returned as a matrix `I` of size `NUM_PROCS_IN_GRIDx3`. Each line of the returned matrix, `I(i,:)` contains the following information:

```
[PROCESSOR_RANK   START_INDEX   END_INDEX]
```

Examples

Let `Ncpus` be 4:

```
P = map([1 Ncpus], {}, 0:Ncpus-1);
D = zeros(100, 100, P);
[I1, I2] = global_block_ranges(D);
```

On every rank, `I1` contains:

I1(1)	I1(2)	I1(3)
0	1	50
2	1	50
1	51	100
3	51	100

On every rank, `I2` contains:

I2(1)	I2(2)	I2(3)
0	1	50
2	51	100
1	1	51
3	51	100

Remarks

The difference between `global_block_range` and `global_block_ranges` is subtle, but important. `global_block_range` returns a single vector containing the index range for *only* that particular processor. `global_block_ranges` returns a matrix that contains the index ranges for *every* processor.

dmat/global_ind{ XE "dmat:global_ind" }

Returns the global indices local to the current processor.

Syntax

```
I = global_ind(D, DIM)
[I1, I2, ..., IN] = global_ind(D)
```

Description

`I = global_ind(D, DIM)` Returns the global indices of the `dmat D` local to the current processor in the specified dimension, `DIM`.

`[I1, I2, ..., IN] = global_ind(D)` Returns the global indices of the `dmat D` local to the current processor in all dimensions of `D`.

The global indices for each dimension are returned as a vector.

Examples

Let `Ncpus` be 4:

```
P = map([1 Ncpus], {}, 0:Ncpus-1);
D = zeros(100, 100, P);
[I1, I2] = global_ind(D);
```

For each rank, `I1` contains:

Rank	I1(:)
0	1 2 3 … 49 50
1	51 52 53 … 99 100
2	1 2 3 … 49 50
3	51 52 53 … 99 100

For each rank, `I2` contains:

Rank	I2(:)
0	1 2 3 … 49 50
1	1 2 3 … 49 50
2	51 52 53 … 99 100
3	51 52 53 … 99 100

dmat/global_inds{ XE "dmat:global_inds" }

Returns the global indices for all processors in the map of `dmat D`.

Syntax

```
I = global_inds(D, DIM)
[I1, I2, ..., IN] = global_inds(D)
```

Description

`global_inds(D, DIM)` Returns global indices of the `dmat D` for all processors in the specified dimension, `DIM`.

`global_inds(D)` Returns global indices of the `dmat D` for all processors in all dimensions of `D`.

For each dimension, the indices are returned as a matrix `I` of size `NUM_PROCS_IN_GRIDXMAX_LOCAL_INDS`. Each line of the returned matrix `I`, `I(i,:)`, contains the following information:

```
[PROCESSOR_RANK   IND1   IND2   ...   INDn]
```

To ensure that all rows in the return index are the same, the indices matrix is appended with extra zeros where there are not enough indices.

Examples

Let `Ncpus` be 4:

```
P = map([1 Ncpus], {}, 0:Ncpus-1);
D = zeros(100, 100, P);
[I1, I2] = global_ind(D);
```

On every rank, `I1` contains:

I1(1)	I1(2:end)
0	1 2 3 … 49 50
2	1 2 3 … 49 50
1	51 52 53 … 99 100
3	51 52 53 … 99 100

On every rank, `I2` contains:

I2(1)	I2(2:end)
0	1 2 3 … 49 50
2	51 52 53 … 99 100
1	1 2 3 … 49 50
3	51 52 53 … 99 100

Remarks

The difference between `global_ind` and `global_inds` is subtle, but important. `global_ind` returns a single vector containing the indices for only that particular processor. `global_ind` returns a matrix that contains the indices for every processor.

MATLAB® is a registered trademark of The Mathworks, Inc.

dmat/global_range{ XE "dmat:global_range" }

Returns the ranges of global indices `dmat D` of local to the current processor. Returns the same range as `global_block_range` if `D` is block distributed, returns subranges for block-cyclic and cyclic distributions.

Syntax

```
I = global_range(D, DIM)
[I1, I2, ..., IN] = global_range(D)
```

Description

`I = global_range(D, DIM)` Returns the global index range of the `dmat D` local to the current processor in the specified dimension, `DIM`.

`[I1, I2, ..., IN] = global_range(D)` Returns the global index range of the `dmat D` local to the current processor in all dimensions of `D`.

For each dimension, the indices are returned as a matrix `I`. Each line of the returned matrix, `I(i,:)`, contains the following information:

`[START_INDEX_1 END_INDEX_1 START_INDEX_2 END_INDEX_2 ...]`

Examples

Let `Ncpus` be 4:

```
dist(1).dist = 'b';
dist(2).dist = 'b';
P = map([Ncpus/2 Ncpus/2], dist, 0:Ncpus-1);
D = zeros(100, 100, P);
[I1, I2] = global_range(D);
```

For each rank, `I1` contains:

Rank	`I1`
0	[1 50]
1	[51 100]
2	[1 50]
3	[51 100]

For each rank, `I2` contains:

Rank	`I2`
0	[1 50]
1	[1 50]
2	[51 100]
3	[51 100]

Let `Ncpus` be 4:

```
dist(1).dist = 'c';
dist(2).dist = 'b';
P = map([Ncpus/2 Ncpus/2], dist, 0:Ncpus-1);
D = zeros(100, 100, P);
[I1, I2] = global_range(D);
```

For each rank, `I1` contains:

Rank	`I1`
0	[1 1 3 3 5 5 … 97 97 99 99]
1	[2 2 4 4 6 6 … 98 98 100 100]
2	[1 1 3 3 5 5 … 97 97 99 99]
3	[2 2 4 4 6 6 … 98 98 100 100]

For each rank, `I2` contains:

Rank	`I2`
0	[1 50]
1	[1 50]
2	[51 100]
3	[51 100]

Let `Ncpus` be 4:

```
dist(1).dist = 'bc';
dist(1).b_size = 4;
dist(2).dist = 'b';
P = map([Ncpus/2 Ncpus/2], dist, 0:Ncpus-1);
D = zeros(100, 100, P);
[I1, I2] = global_range(D);
```

For each rank, `I1` contains:

Rank	`I1`
0	[1 4 9 12 … 89 92 97 100]
1	[5 8 13 16 … 93 96]
2	[1 4 9 12 … 89 92 97 100]
3	[5 8 13 16 … 93 96]

For each rank, `I2` contains:

Rank	`I2`
0	[1 50]
1	[1 50]
2	[51 100]
3	[51 100]

dmat/global_ranges{ XE "dmat:global_ranges" }

Returns the ranges of global indices for all processors in the map of `dmat D`. Returns the same range as `global_block_ranges` if `D` is block distributed, returns subranges for block-cyclic and cyclic distributions.

Syntax

```
I = global_ranges(D, DIM)
[I1, I2, ..., IN] = global_ranges(D)
```

Description

`I = global_ranges(D, DIM)` Returns the global index ranges of the `dmat D` for all processors in the specified dimension, `DIM`.

`[I1, I2, ..., IN] = global_ranges(D)` Returns the global index range of the `dmat D` for all processors in all dimensions of `D`.

For each dimension, the indices are returned as a matrix `I` of size

`NUM_PROCS_IN_GRIDxNUM_BLOCK_BOUNDARIES`. Each line of the returned matrix, `I(i,:)`, contains the following information:

`[PROCESSOR_RANK START_INDEX_1 END_INDEX_1 START_INDEX_2 END_INDEX_2 ...]`

Examples

Let `Ncpus` be 4:

```
dist(1).dist = 'b';
dist(2).dist = 'b';
P = map([Ncpus/2 Ncpus/2], dist, 0:Ncpus-1);
D = zeros(100, 100, P);
[I1, I2] = global_ranges(D);
```

On every rank, `I1` contains:

I1(1)	I1(2:end)
0	1 50
2	1 50
1	51 100
3	51 100

On every rank, `I2` contains:

I2(1)	I2(2:end)
0	1 50
2	51 100
1	1 50
3	51 100

Let `Ncpus` be 4:

```
dist(1).dist = 'c';
dist(2).dist = 'b';
P = map([Ncpus/2 Ncpus/2], dist, 0:Ncpus-1);
D = zeros(100, 100, P);
[I1, I2] = global_ranges(D);
```

On every rank, `I1` contains:

I1(1)	I1(2:end)
0	1 1 3 3 5 5 … 97 97 99 99
2	1 1 3 3 5 5 … 97 97 99 99
1	2 2 4 4 6 6 … 98 98 100 100
3	2 2 4 4 6 6 … 98 98 100 100

On every rank, `I2` contains:

I2(1)	I2(2:end)
0	1 50
2	51 100
1	1 50
3	51 100

Let `Ncpus` be 4:

```
dist(1).dist = 'bc';
dist(1).b_size = 4;
dist(2).dist = 'b';
P = map([Ncpus/2 Ncpus/2], dist, 0:Ncpus-1);
D = zeros(100, 100, P);
[I1, I2] = global_ranges(D);
```

On every rank, `I1` contains:

I1(1)	I1(2:end)
0	1 4 9 12 … 89 92 97 100
2	1 4 9 12 … 89 92 97 100
1	5 8 13 16 … 93 96 0 0
3	5 8 13 16 … 93 96 0 0

On every rank, `I2` contains:

I2(1)	I2(2:end)
0	1 50
2	51 100
1	1 50
3	51 100

Remarks

If processors in the same dimension have different number of blocks, the block boundaries are padded with zeros for the processors that have fewer blocks.

map/inmap{ XE "map:inmap" }

Checks if a processor is in the map.

Syntax

```
b = inmap(m, r)
```

Description

`b = inmap(m, r)` checks if processor rank `r` is in map `m`. Returns TRUE for Boolean if rank `r` is in the map, FALSE otherwise.

dmat/find{ XE "dmat:find" }

Find indices of nonzero elements in a `dmat`.

Syntax

```
[I, J] = find(X)
```

Description

`[I, J] = find(X)` returns the row and column indices of nonzero elements of the `dmat X`.

Remarks

Currently supports only `[I, J] = find(X)` calling convention. Only works on 2D `dmats`.

`find` requires every processor to send its results to every other processor, thus can incur a significant amount of communication overhead.

Distributed matrix manipulation

---

dmat/agg{ XE "dmat:agg" }

Aggregates the parts of a `dmat` on the leader processor.

Syntax

```
A = agg(D)
```

Description

`A = agg(D)` aggregates the parts of a `dmat D` into a whole and returns it as a regular `double` matrix, `A`. If the current processor is the leader, returns the aggregated matrix. Otherwise, returns the local part of `D`.

Remarks

Currently, it doesn't matter if the leader is in the map – the global matrix is returned on the leader, regardless.

Note that `agg` incurs communication overhead to aggregate `D` onto the leader processor.

Since `A` on the leader processor contains the entire contents of `D` but on all other processors contains only the local portion of `D`, `A` will have different values and sizes on each processor. Thus, `agg` should be used with caution.

dmat/agg_all{ XE "dmat:agg_all" }

Aggregates the parts of a `dmat` onto all processors.

Syntax

```
A = agg_all(D)
```

Description

`A = agg_all(D)` aggregates the parts of a `dmat D` onto all processors in the map of `D` and returns a regular `double` matrix `A`.

Remarks

Unlike `agg`, `agg_all` creates a result that is consistent in size and values across all processors . However, because `agg_all` causes all processors to communicate with all processors, `agg_all` can incur a significant amount of communication and should be used with caution.

---

dmat/local{ XE "dmat:local" }

Returns the local part of the `dmat`.

Syntax

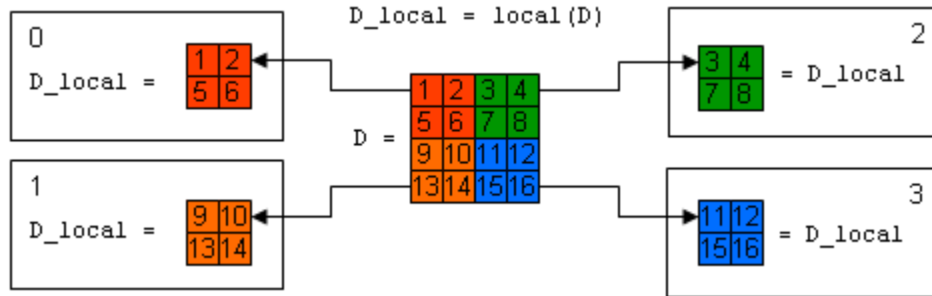
```
D_local = local(D)
```

Description

`D_local = local(D)` Returns the local part of the `dmat D` on the current processor.

Examples

The following diagram shows four processors obtaining their respective local parts of the `dmat`, `D`, and copying the contents into a local variable, `D_local`. Note that `D_local` exists on each processor but contains different data.



dmat/put_local{ XE "dmat:put_local" }

Assigns new data to the local part of the `dmat`.

Syntax

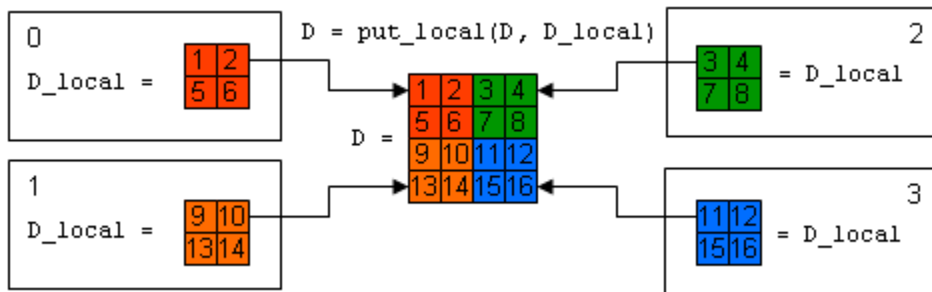
```
D = put_local(D, D_LOCAL)
```

Description

`D = put_local(D, D_LOCAL)` assigns `D_LOCAL` to the local part of the `dmat D`.

Examples

The following diagram shows four processors each writing a local matrix, `D_local`, into their respective parts of a `dmat`, `D`. Note that `D_local` on each processor must be the same size as the local portion of their respective parts of `D`.



dmat/synch{ XE "dmat:synch" }

Synchronize the overlapped data in a `dmat`.

Syntax

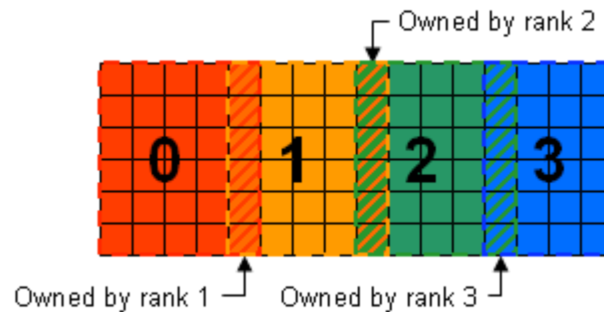
```
D = synch(D)
```

Description

`D = synch(D)` If overlap is present, the owner processor of the overlapping data sends its data to the processor that has a copy of the overlapping data. No-op if there is no overlap.

Remarks

The owner is the processor with the higher index in the grid in the corresponding dimension. For example, if the overlap is in the second dimension the owner is the processor in the column of the grid with the higher index.



---

dmat/sync2{ XE "dmat:synch" }

Perform an operation on the overlapped data with the result that both processors hold the final data locally.

Syntax

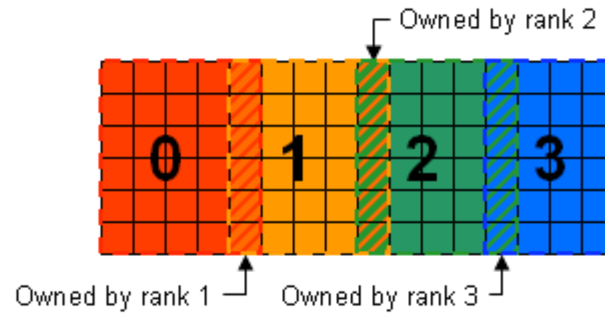
```
D = synch2(D)
```

Description

`D = synch2(D)` If overlap is present, each processor sends it's overlapping data to it's neighbor. Currently only the addition operator is supported, so each processor adds the received data to its given data, the result being that all processors have the sum of their overlap. No-op if there is no overlap.

Remarks

The owner is the processor with the higher index in the grid in the corresponding dimension. For example, if the overlap is in the second dimension the owner is the processor in the column of the grid with the higher index.



---

remap{ XE "remap" }

Remaps a `dmat` with a new map.

Syntax

```
remap(X, NEW_MAP)
```

Description

`remap(X, NEW_MAP)` takes a `dmat` `X` and redistributes it according to the specified `map NEW_MAP`.

dmat/subsasgn{ XE "dmat:subsasgn" }

Subscripted assignment to a distributed object. Overloaded method for `A(I)=B`. Should not be called directly.

Syntax

```
A = subsasgn(A, S, B)
```

Description

`A = subsasgn(A, S, B)` Subscripted assignment of `B` (right hand side) to `A` (left hand side). `A` is of type `dmat`. `B` can be of type `dmat` or `double`. `S` is a structure array with the fields:

- `type`: String containing `'()'`, `'{}'`, or `'.'` specifying the subscript type. Currently only supports `'()'`.
- `subs`: Cell array or string containing the actual subscripts.

Remarks

In cases where `A(I)` and/or `B` are distributed across multiple processors, `subsasgn` will automatically transfer the appropriate data between processors.

dmat/subsref{ XE "dmat:subsref" }

Subscripted reference. Overloaded method for `A(I)`. Should not be called directly.

Syntax

```
B = subsref(A, S)
```

Description

`B = subsref(A, S)` Subscripted reference on a `dmat` A. S is a structure array with the fields:

- `type`: String containing `'()'`, `'{}'`, or `'.'` specifying the subscript type.
- `subs`: Cell array or string containing the actual subscripts.

Remarks:

Currently, `subsref` will only return a standalone `dmat` in the following cases:

- `A(:,:)` – Refers to the entire contents of the `dmat` `A`
- `A(i,j)` – Refers to a single element in the `dmat` `A`. Returns a new dmat containing the value at (i,j) stored on the processor which contained that element in `A`.

In all other cases, `subsref` will not produce a "standalone" `dmat`, i.e. the resulting `dmat` can not be directly used as an input to any pMatlab function, with the exception of `local`.

map/subsasgn{ XE "map:subsasgn" }

Subscripted assignment. Should not be called directly.

Syntax

```
A = subsasgn(A, S, B)
```

Description

`A = subsasgn(A, S, B)` Subscripted assignment of `B` (right hand side) to `A` (left hand side).

`A.FIELD = B` allows the fields of a `map` object `A` to be assigned using the `'.'` notation (complies with structure behavior).

Remarks

This functionality might be deprecated from the final API, to limit control the user has of private members of the MAP object.

---

map/subsref{ XE "map:subsref" }

Subscripted reference. Should not be called directly.

Syntax

```
B = subsref(A, S)
```

Description

`A = subsref (A, S, B)` Subscripted assignment of `A` (right hand side) to `B` (left hand side).

`B = A.FIELD` allows the fields of a `map` object `A` to be referenced using the `'.'` notation (complies with structure behavior).

Remarks

This functionality might be deprecated from the final API, to limit control the user has of private members of the MAP object. `subsref` might be replaced by `get` functions.

transpose_grid{ XE "transpose_grid" }

Redistributes a `dmat` by transposing its grid.

Syntax

```
A = transpose_grid(B)
```

Description

`A = transpose_grid(B)` creates a `dmat` `A` that has the same contents as the `dmat` `B`, except that the grid for `A`'s map is the transpose of the grid for `B`'s map. The contents of `B` are automatically redistributed to `A`. `transpose_grid` is only supported for 2D `dmats`.

`transpose_grid` is optimized to redistribute row-distributed `dmats` into column-distributions and vice versa. For example, suppose `B`'s map had a grid specification of `[1 4]`. Then `A`'s map will have a grid specification of `[4 1]`. Note that `B` must be block distributed in both dimensions with no overlap.

For all other distributions, e.g. `dmats` with row and column distributions, overlap, etc., `subsasgn` will be used to redistribute `B`.

Elementary math functions

Trigonometric

---

dmat/sin{ XE "dmat:sin" }, dmat/cos{ XE "dmat:cos" }, dmat/tan{ XE "dmat:tan" }, dmat/sec{ XE "dmat:sec" }, dmat/csc{ XE "dmat:csc" }, dmat/cot{ XE "dmat:cot" }

Performs the trigonometric operation in radians.

Syntax

```
Y = sin(X)
Y = cos(X)
Y = tan(X)
Y = sec(X)
Y = csc(X)
Y = cot(X)
```

Description

`Y = sin(X)` is the sine of the elements of the distributed array `X`, expressed in radians.

`Y = cos(X)` is the cosine of the elements of the distributed array `X`, expressed in radians.

`Y = tan(X)` is the tangent of the elements of the distributed array `X`, expressed in radians.

`Y = sec(X)` is the secant of the elements of the distributed array `X`, expressed in radians.

`Y = csc(X)` is the cosecant of the elements of the distributed array `X`, expressed in radians.

`Y = cot(X)` is the cotangent of the elements of the distributed array `X`, expressed in radians.

dmat/sind{ XE "dmat:sind" }, dmat/cosd{ XE "dmat:cosd" }, dmat/tand{ XE "dmat:tand" }, dmat/secd{ XE "dmat:secd" }, dmat/cscd{ XE "dmat:cscd" }, dmat/cotd{ XE "dmat:cotd" }

Performs the trigonometric operation in degrees.

Syntax

```
Y = sind(X)
Y = cosd(X)
Y = tand(X)
Y = secd(X)
Y = cscd(X)
Y = cotd(X)
```

Description

`Y = sind(X)` is the sine of the elements of the distributed array `X`, expressed in degrees.

`Y = cosd(X)` is the cosine of the elements of the distributed array `X`, expressed in degrees.

`Y = tand(X)` is the tangent of the elements of the distributed array `X`, expressed in degrees.

`Y = secd(X)` is the secant of the elements of the distributed array `X`, expressed in degrees.

`Y = cscd(X)` is the cosecant of the elements of the distributed array `X`, expressed in degrees.

`Y = cotd(X)` is the cotangent of the elements of the distributed array `X`, expressed in degrees.

dmat/sinh{ XE "dmat:sinh" }, dmat/cosh{ XE "dmat:cosh" }, dmat/tanh{ XE "dmat:tanh" }, dmat/sech{ XE "dmat:sech" }, dmat/csch{ XE "dmat:csch" }, dmat/coth{ XE "dmat:coth" }

Performs the hyperbolic trigonometric operation.

Syntax

```
Y = sinh(X)
Y = cosh(X)
Y = tanh(X)
Y = sech(X)
Y = csch(X)
Y = coth(X)
```

Description

`Y = sinh(X)` is the hyperbolic sine of the elements of the distributed array `X`.

`Y = cosh(X)` is the hyperbolic cosine of the elements of the distributed array `X`.

`Y = tanh(X)` is the hyperbolic tangent of the elements of the distributed array `X`.

`Y = sech(X)` is the hyperbolic secant of the elements of the distributed array `X`.

`Y = csch(X)` is the hyperbolic cosecant of the elements of the distributed array `X`.

`Y = coth(X)` is the hyperbolic cotangent of the elements of the distributed array `X`.

dmat/asin{ XE "dmat:asin" }, dmat/acos{ XE "dmat:acos" }, dmat/atan{ XE "dmat:atan" }, dmat/asec{ XE "dmat:asec" }, dmat/acsc{ XE "dmat:acsc" }, dmat/acot{ XE "dmat:acot" }

Performs the inverse trigonometric operation in radians.

Syntax

```
Y = asin(X)
Y = acos(X)
Y = atan(X)
Y = asec(X)
Y = acsc(X)
Y = acot(X)
```

Description

`Y = asin(X)` is the inverse sine of the elements of the distributed array `X`, expressed in radians. Complex results are obtained if `abs(x) > 1.0` for some element.

`Y = acos(X)` is the inverse cosine of the elements of the distributed array `X`, expressed in radians. Complex results are obtained if `abs(x) > 1.0` for some element.

`Y = atan(X)` is the inverse tangent of the elements of the distributed array `X`, expressed in radians.

`Y = asec(X)` is the inverse secant of the elements of the distributed array `X`, expressed in radians.

`Y = acsc(X)` is the inverse cosecant of the elements of the distributed array `X`, expressed in radians.

`Y = acot(X)` is the inverse cotangent of the elements of the distributed array `X`, expressed in radians.

dmat/asind{ XE "dmat:asind" }, dmat/acosd{ XE "dmat:acosd" }, dmat/atand{ XE "dmat:atand" }, dmat/asecd{ XE "dmat:asecd" }, dmat/acscd{ XE "dmat:acscd" }, dmat/acotd{ XE "dmat:acotd" }

Performs the inverse trigonometric operation in degrees.

Syntax

```
Y = asind(X)
Y = acosd(X)
Y = atand(X)
Y = asecd(X)
Y = acscd(X)
Y = acotd(X)
```

Description

`Y = asin(X)` is the inverse sine of the elements of the distributed array `X`, expressed in degrees.

`Y = acos(X)` is the inverse cosine of the elements of the distributed array `X`, expressed in degrees.

`Y = atan(X)` is the inverse tangent of the elements of the distributed array `X`, expressed in degrees.

`Y = asec(X)` is the inverse secant of the elements of the distributed array `X`, expressed in degrees.

`Y = acsc(X)` is the inverse cosecant of the elements of the distributed array `X`, expressed in degrees.

`Y = acot(X)` is the inverse cotangent of the elements of the distributed array `X`, expressed in degrees.

dmat/asinh{ XE "dmat:asinh" }, dmat/acosh{ XE "dmat:acosh" }, dmat/atanh{ XE "dmat:atanh" }, dmat/asech{ XE "dmat:asech" }, dmat/acsch{ XE "dmat:acsch" }, dmat/acoth{ XE "dmat:acoth" }

Performs the inverse hyperbolic trigonometric operation.

Syntax

```
Y = asinh(X)
Y = acosh(X)
Y = atanh(X)
Y = asech(X)
Y = acsch(X)
Y = acoth(X)
```

Description

`Y = asinh(X)` is the inverse hyperbolic sine of the elements of the distributed array `X`.

`Y = acosh(X)` is the inverse hyperbolic cosine of the elements of the distributed array `X`.

`Y = atanh(X)` is the inverse hyperbolic tangent of the elements of the distributed array `X`.

`Y = asech(X)` is the inverse hyperbolic secant of the elements of the distributed array `X`.

`Y = acsch(X)` is the inverse hyperbolic cosecant of the elements of the distributed array `X`.

`Y = acoth(X)` is the inverse hyperbolic cotangent of the elements of the distributed array `X`.

Exponential

---

dmat/exp{ XE "dmat:exp" }

Exponential.

Syntax

```
Y = exp(X)
```

Description

`exp(X)` is the exponential of the elements of `X`, `e` to the `X`.

For complex `Z=X+i*Y`, `exp(Z) = exp(X)*(cos(Y)+i*sin(Y))`.

---

dmat/expm1{ XE "dmat:expm1" }

Compute `exp(x)-1` accurately.

Syntax

```
Y = expm1(X)
```

Description

`expm1(X)` computes `exp(X)-1`, compensating for the roundoff in `exp(X)` for distributed array `X`. For small `X`, `expm1(X)` is approximately `X`, whereas `exp(X)-1` can be zero.

---

dmat/log{ XE "dmat:log" }

Natural logarithm.

Syntax

```
Y = log(X)
```

Description

`log(X)` is the natural logarithm of the elements of distributed array `X`. Complex results are produced if `X` is not positive.

dmat/log1p{ XE "dmat:log1p" }

Compute `log(1+x)` accurately.

Syntax

```
Y = log1p(X)
```

Description

`log1p(X)` computes `log(1+x)`, compensating for the roundoff in `1+x` for distributed array `X`. For small `X`, `log1p(X)` is approximately `X`, whereas `log(1+x)` can be zero.

---

dmat/log10{ XE "dmat:log10" }

Compute common (base 10) logarithm.

Syntax

```
Y = log10(X)
```

Description

`log10(X)` is the base 10 logarithm of the elements of distributed array `X`. Complex results are produced if `X` is not positive.

---

dmat/log2

Compute base 2 logarithm.

Syntax

```
Y = log2(X)
```

Description

`log2(X)` is the base 2 logarithm of the elements of distributed array `X`.

dmat/pow2{ XE "dmat:pow2" }

Compute base 2 power.

Syntax

```
Y = pow2(X)
```

Description

`pow2(X)` for each element of distributed array `X` is 2 raised to the power `X`.

---

dmat/realpow{ XE "dmat:realpow" }

Element by element power that will error out on complex result.

Syntax

```
R = realpow(P,Q)
```

Description

`realpow(P,Q)` denotes element-by-element powers. The array dimensions must be the same, unless the non-distributed argument is a scalar. If both `P` and `Q` are `dmats`, their maps must be the same. The `dmat` input must not be the only scalar, since operations with a distributed scalar incur significant communication overhead.

---

dmat/reallog{ XE "dmat:reallog" }

Natural log of real number.

Syntax

```
Y = reallog(X)
```

Description

`reallog(X)` is the natural logarithm of the elements of distributed array `X`. An error is produced if `X` is not positive.

dmat/realsqrt{ XE "dmat:realsqrt" }

Square root of number greater than or equal to zero.

Syntax

```
Y = realsqrt(X)
```

Description

`realsqrt(X)` is the square root of the elements of distributed array `X`. An error is produced if `X` is not positive.

---

dmat/sqrt{ XE "dmat:sqrt" }

Square root.

Syntax

```
Y = sqrt(X)
```

Description

`sqrt(X)` is the square root of the elements of distributed array `X`. Complex results are produced if `X` is not positive.

Complex

---

dmat/abs{ XE "dmat:abs" }

Absolute value.

Syntax

```
Y = abs(X)
```

Description

`abs(X)` is the absolute values of the elements of distributed array `X`.

---

dmat/angle{ XE "dmat:angle" }

Phase angle

Syntax

```
Y = angle(X)
```

Description

`angle(X)` returns the phase angles, in radians, of a distributed array `X` with complex elements.

---

dmat/complex{ XE "dmat:complex" }

Construct complex `dmat` from real `dmat`.

Syntax

```
C = complex(A)
C = complex(A, B)
```

Description

`C = complex(A)` for real `A` returns the complex `dmat` `C` with real part `A` and all zero imaginary part.

`C = complex(A, B)` returns the complex `dmat` `A + Bi`. `A` and `B` must have the same mapping.

---

dmat/conj{ XE "dmat:conj" }

Complex conjugate.

Syntax

```
Y = conj(X)
```

Description

`conj(X)` is the complex conjugate of distributed array `X`.

For a complex `X`, `CONJ(X) = REAL(X) - i*IMAG(X)`.

---

dmat/imag{ XE "dmat:imag" }

Complex imaginary part.

Syntax

```
Y = imag(X)
```

Description

`imag(X)` is the imaginary part of distributed array `X`.

---

dmat/real{ XE "dmat:real" }

Complex real part.

Syntax

```
Y = real(X)
```

Description

`real(X)` is the real part of distributed array `X`.

Rounding and remainder

---

dmat/fix{ XE "dmat:fix" }

Round towards zero.

Syntax

```
Y = fix(X)
```

Description

`fix(X)` rounds the elements of distributed array `X` to the nearest integers towards zero.

---

dmat/floor{ XE "dmat:floor" }

Round towards minus infinity.

Syntax

```
Y = floor(X)
```

Description

`floor(X)` rounds the elements of the distributed array `X` to the nearest integers towards minus infinity.

---

dmat/ceil{ XE "dmat:ceil" }

Round towards plus infinity.

Syntax

```
Y = ceil(X)
```

Description

`ceil(X)` rounds the elements of distributed array `X` to the nearest integers towards infinity.

dmat/round{ XE "dmat:round" }

Round towards nearest integer.

Syntax

```
Y = round(X)
```

Description

`round(X)` rounds the elements of distributed array `X` to the nearest integers.

---

dmat/sign

Signum function.

Syntax

```
Y = sign(X)
```

Description

`sign(X)` For each element of distributed array `X`, `sign(X)` returns 1 if the element is greater than zero, 0 if it equals zero and -1 if it is less than zero.

For the nonzero elements of complex `X`, `sign(X) = X ./ ABS(X)`.

Operators and special characters

---

dmat/plus{ XE "dmat:plus" }

\+ Plus.

Syntax

```
R = P + Q
R = plus(P, Q)
```

Description

`R = P + Q` or `R = plus(P, Q)` performs an element-wise addition between `P` and `Q`. If both `P` and `Q` are `dmats`, then their dimensions and maps must be equal. If only one of `P` and `Q` is a `dmat`, then their dimensions must be the same or the non-`dmat` input must be a scalar. `dmat` inputs can only be scalar if both inputs are `dmats` and have the same map.

---

dmat/minus{ XE "dmat:minus" }

\- Minus.

Syntax

```
R = P - Q
R = minus(P, Q)
```

Description

`R = P - Q` or `R = minus(P, Q)` performs an element-wise subtraction between `P` and `Q` (subtract `Q` from `P`). If both `P` and `Q` are `dmats`, then their dimensions and maps must be equal. If only one of `P` and `Q` is a `dmat`, then their dimensions must be the same or the non-`dmat` input must be a scalar. `dmat` inputs can only be scalar if both inputs are `dmats` and have the same map.

dmat/mtimes{ XE "dmat:mtimes" }

`*` Matrix multiply.

Syntax

```
C = A * B
C = mtimes(A, B)
```

Description

`C = A * B` or `C = mtimes(A, B)` multiples two matrices together. `A` and/or `B` may be a `dmat` with any type of distribution. `C` will have a distribution of the distributed matrix operand, or `A` if both `A` and `B` are distributed matrices.

Remarks

Overlaps have not been tested.

---

dmat/times{ XE "dmat:times" }

`.*` Array multiply.

Syntax

```
C = A .* B
C = times(A, B)
```

Description

`C = A .* B` or `C = times(A, B)` performs an element-wise multiplication between `A` and `B`. If both `A` and `B` are `dmats`, then their dimensions and maps must be equal. If only one of `A` and `B` is a `dmat`, then their dimensions must be the same or the non-`dmat` input must be a scalar. `dmat` inputs can only be scalar if both inputs are `dmats` and have the same map.

dmat/power{ XE "dmat:power" }

`.^` Array power.

Syntax

```
R = P .^ Q
R = power(P, Q)
```

Description

`R = P .^ Q` or `R = power(P, Q)` performs an element-wise power operation between `P` and `Q`. If both `P` and `Q` are `dmats`, then their dimensions and maps must be equal. If only one of `P` and `Q` is a `dmat`, then their dimensions must be the same or the non-`dmat` input must be a scalar. `dmat` inputs can only be scalar if both inputs are `dmats` and have the same map.

---

dmat/ldivide{ XE "dmat:ldivide" }

`.\` Left array divide.

Syntax

```
R = P .\ Q
R = ldivide(P, Q)
```

Description

`R = P .\ Q` or `R = ldivide(P, Q)` performs an element-wise left divide between `P` and `Q` (divide `Q` by `P`). If both `P` and `Q` are `dmats`, then their dimensions and maps must be equal. If only one of `P` and `Q` is a `dmat`, then their dimensions must be the same or the non-`dmat` input must be a scalar. `dmat` inputs can only be scalar if both inputs are `dmats` and have the same map.

dmat/rdivide{ XE "dmat:rdivide" }

`./` Right array divide.

Syntax

```
R = P ./ Q
R = rdivide(P, Q)
```

Description

`R = P ./ Q` or `R = rdivide(P, Q)` performs an element-wise right divide between `P` and `Q` (divide `P` by `Q`). If both `P` and `Q` are `dmats`, then their dimensions and maps must be equal. If only one of `P` and `Q` is a `dmat`, then their dimensions must be the same or the non-`dmat` input must be a scalar. `dmat` inputs can only be scalar if both inputs are `dmats` and have the same map.

---

dmat/eq{ XE "dmat:eq" }

== Equal.

Syntax

```
A == B
```

Description

`A == B` compares the dimensions, maps, sizes and data of `A` and `B`.

If `A` and `B`'s maps, dimensions, and sizes agree then the output is a `dmat` with 0 where elements are not equal and 1 where elements are equal (similar to serial MATLAB).

If the maps are not equal, then a 0 is returned regardless of any other properties.

If the maps agree but the dimensions and/or sizes are not equal, then an error is thrown (analogous to serial MATLAB behavior).

map/eq{ XE "map:eq" }

== Equal.

Syntax

```
A == B
```

Description

`A == B` compares member variables of maps `A` and `B`. If all are the same then TRUE is returned, otherwise FALSE is returned.

---

dmat/gt{ XE "dmat:gt" }

> Greater than.

Syntax

```
A > B
```

Description

`A > B` compares each element of `dmat A` to scalar `B`. Returns a `dmat` with each entry equal to 0 if original entry was < `B`, and 1 otherwise. Calls the MATLAB `gt` on the local part of A.

---

map/ne{ XE "map:ne" }

~= Not equal.

Syntax

```
A ~= B
```

Description

`A ~= B` Returns FALSE if two maps are equal, TRUE otherwise. Two maps are equal if their grids are equal.

dmat/transpose{ XE "dmat:transpose" }

`.′` Transpose

Syntax

```
A = B.′
A = transpose(B)
```

Description

`B.'` is the non-conjugate transpose of a distributed matrix. The input distributed array must be a matrix. The distribution of the distributed array is limited to block in 1, 2nd, or both dimensions.

---

dmat/transpose{ XE "dmat:transpose" }

`′` Complex conjugate transpose

Syntax

```
A = B′
A = ctranspose(B)
```

Description

`B.'` is the complex conjugate transpose of a distributed matrix. The input distributed array must be a matrix. The distribution of the distributed array is limited to block in 1st, 2nd, or both dimensions.

Sparse matrices

map/sparse{ XE "map:sparse" }

Create a sparse `dmat`.

Syntax

```
S = sparse([],[],[],M,N,NZMAX,P)
S = sparse([],[],[],M,N,P)
S = sparse([],[],[],P)
S = sparse(M,N,P)
```

Description

`S = sparse(I,J,S,M,N,NZMAX,P)` generates an `M`-by-`N` sparse `dmat` distributed according to map `P` with space allocated for `NZMAX` nonzeros (note that `NZMAX` applies to the overall `dmat`, not to individual processors. `NZMAX` will be distributed as evenly as possible over all processors).

The rows of `[I,J,S]` are intended to be used to initialize the non-zero values of the matrix. However, `sparse` currently does not support the use of `I,J,` and `S` in pMatlab; they are kept to remain consistent with the `sparse` function built into MATLAB.

There are four ways that `sparse` can be called:

- `S = sparse([],[],[],M,N,NZMAX,P)`
- `S = sparse([],[],[],M,N,P)` uses `NZMAX = 0`.
- `S = sparse([],[],[],P)` uses `M = 0` and `N = 0`. This generates the ultimate sparse matrix, an `M`-by-`N` all zero matrix.
- `S = sparse(M,N,P)` abbreviates `sparse([],[],[],M,N,0,P)`. This also generates an `M`-by-`N` all zero matrix.

Remarks

The recommended method of creating a sparse `dmat` is with `spalloc`.

map/spalloc{ XE "map:spalloc" }

Allocate space for a sparse `dmat`.

Syntax

```
S = spalloc(M, N, NZMAX, P)
```

Description

`S = spalloc(M, N, NZMAX, P)` creates an `M`-by-`N` all zero sparse `dmat` with room to eventually hold `NZMAX` nonzeros.

Remarks

`NZMAX` applies to the overall `dmat`, not individual processors. `NZMAX` is evenly distributed across processors

---

dmat/sparse{ XE "dmat:sparse" }

Converts a `dmat` to a sparse `dmat`

Syntax

```
S = sparse(X)
```

Description

`S = sparse(X)` converts a full `dmat` to sparse form by squeezing out any zero elements. If `X` is already a sparse `dmat`, `sparse(X)` returns `X`.

Data analysis and Fourier transforms

---

dmat/conv2{ XE "dmat:conv2" }

Two dimensional convolution.

Syntax

```
C = conv2(A, B, 'shape')
```

Description

`C = conv2(A, B, 'shape')` performs the 2D convolution of `dmat A` and `double B`. Returns a subsection of the 2D convolution with size specified by `'shape'`.

Remarks

Only `'shape' == 'same'` is supported, which returns the central part of the convolution of the same size as `A`.

---

dmat/fft{ XE "dmat:fft" }

Discrete Fourier transform on a `dmat`.

Syntax

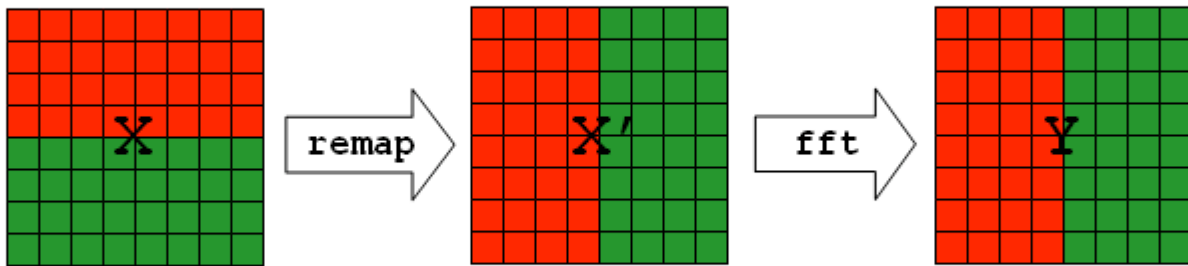
```
Y = fft(X)
Y = fft(X, [], DIM)
Y = fft(X, N, DIM)
```

Description

`fft(X)` is the discrete Fourier transform (DFT) of matrix `X`. The FFT operation is applied to each column. If the matrix `X` is row distributed, `fft` displays a warning and remaps `X`. Calls MATLAB fft on the local part.

`fft(X, [], DIM)` or `fft(X, N, DIM)` applies FFT across dimension `DIM`. `fft(X, N, DIM)` returns the `N` -point DFT. If `X` is distributed along a dimension other than dimension `DIM`, displays a warning and remaps `X` along the dimension `DIM`. Calls the MATLAB `fft` on the local part.

For example, suppose that X is distributed row-wise. Calling `Y = fft(X, [], 1)`, which will perform a FFT on each column, will perform the following remapping:



Remarks

fft supports 2D and 3D dmats.

If fft remaps X, Y has the new map (different from the original map passed in with X).

Data types and structures

---

dmat/double{ XE "dmat:double" }

Converts each local part of the `dmat` to double precision floating point.

Syntax

```
D = double(X)
```

Description

`D = double(X)` returns the double precision value for the local part of `X` in a `dmat` `D` with the same mapping and dimensions as `X`. If the local part of `D` is already double precision, `double` has no effect.

---

dmat/single{ XE "dmat:single" }

Converts each local part of the `dmat` to single precision floating point.

Syntax

```
D = single(X)
```

Description

`D = single(X)` returns the single precision value for the local part of `X` in a `dmat` `D` with the same mapping and dimensions as `X`. If the local part of `D` is already single precision, `double` has no effect.

dmat/uint8{ XE "dmat:uint8" }, dmat/uint16{ XE "dmat:uint16" }, dmat/uint32{ XE "dmat:uint32" }, dmat/uint64{ XE "dmat:uint64" }

Converts each local part of the `dmat` to unsigned integer.

Syntax

```
D = uint8(X)
D = uint16(X)
D = uint32(X)
D = uint64(X)
```

Description

`D = uint*(X)` returns the unsigned integer value for the local part of `X` in a `dmat` `D` with the same mapping and dimensions as `X`. If the local part of `D` is already unsigned integer, `uint*` has no effect.

---

dmat/int8{ XE "dmat:int8" }, dmat/int16{ XE "dmat:int16" }, dmat/int32{ XE "dmat:int32" }, dmat/int64{ XE "dmat:int64" }

Converts each local part of the a `dmat` to signed integer.

Syntax

```
D = int8(X)
D = int16(X)
D = int32(X)
D = int64(X)
```

Description

`D = int*(X)` returns the signed integer value for the local part of `X` in a `dmat` `D` with the same mapping and dimensions as `X`. If the local part of `D` is already signed integer, `int*` has no effect.

Signal Processing Toolbox

---

dmat/dct{ XE "dmat:dct" }

Distributed discrete cosine transform

Syntax

```
B = dct(A)
```

Description

`B = dct(A)` returns the discrete cosine transform of `A`. The vector `B` is the same size as `A` and contains the discrete cosine transform coefficients.

If `A` is a matrix, the `dct` operation is applied to each column. This transform can be inverted using `idct`.

---

dmat/idct{ XE "dmat:idct" }

Distributed inverse discrete cosine transform

Syntax

```
A = idct(B)
```

Description

`A = idct(B)` inverts the `dct` transform, returning the original vector if `B` was obtained using `B = dct(A)`.

If `B` is a matrix, the `idct` operation is applied to each column.

Index

dmat
- abs, 46
- acos, 39
- acosd, 40
- acosh, 41
- acot, 39
- acotd, 40
- acoth, 41
- acsc, 39
- acscd, 40
- acsch, 41
- agg, 28
- agg_all, 29
- angle, 46
- asec, 39
- asecd, 40
- asech, 41
- asin, 39
- asind, 40
- asinh, 41
- atan, 39
- atand, 40
- atanh, 41
- ceil, 48
- complex, 46
- conj, 47
- conv2, 58
- cos, 36
- cosd, 37
- cosh, 38
- cot, 36
- cotd, 37
- coth, 38
- csc, 36
- cscd, 37
- csch, 38
- dct, 62
- display, 17
- double, 60
- eq, 53
- exp, 42
- expm1, 42
- fft, 58
- find, 27
- fix, 48
- floor, 48
- global_block_range, 18
- global_block_ranges, 19
- global_ind, 20
- global_inds, 21
- global_range, 22
- global_ranges, 24
- gt, 54
- idct, 62
- imag, 47
- int16, 61
- int32, 61
- int64, 61
- int8, 61
- ldivide, 52
- local, 29
- log, 42

log10, 43
log1p, 43
minus, 50
mtimes, 51
ndims, 16
plus, 50
pow2, 44
power, 52
put_local, 30
rdivide, 53
real, 47
reallog, 44
realpow, 44
realsqrt, 45
round, 49
sec, 36
secd, 37
sech, 38
sin, 36
sind, 37
single, 60
sinh, 38
size, 16
sparse, 57
sqrt, 45
subsasgn, 32
subsref, 33
synch, 31
tan, 36
tand, 37
tanh, 38
times, 51
transpose, 55
uint16, 61
uint32, 61
uint64, 61
uint8, 61

map
- display, 17
- eq, 54
- inmap, 26
- map, 9
- ne, 54
- ones, 14
- rand, 15
- spalloc, 57
- sparse, 56
- subsasgn, 34
- subsref, 34
- zeros, 13

MatMPI_Delete_all, 8
MPI_Abort, 7
MPI_Run, 8
pMATLAB, 6
pMatlab_Finalize, 7
pMatlab_Init, 6
pMatlab_ver, 7
remap, 31
transpose_grid, 35