Writing Parallel Parameter Sweep Applications with pMatlab

Hahn Kim, Albert Reuther, Jeremy Kepner
{hgk, reuther, kepner}@ll.mit.edu
MIT Lincoln Laboratory, Lexington, MA 02144

Abstract

Parameter sweep applications execute the same piece of code multiple times with unique sets of input parameters. This type of application is extremely amenable to parallelization. This document describes how to parallelize parameter sweep applications with pMatlab by introducing a simple serial parameter sweep application written in MATLAB®, then parallelizing the application using pMatlab.

1. Introduction

Parameter sweep applications are a class of application in which the same code is run multiple times using unique sets of input parameter values. This includes varying one parameter over a range of values or varying multiple parameters over a large multidimensional space. Examples of parameter sweep applications are Monte Carlo simulations or parameter space searches.

In parameter sweep applications, each individual run is independent of all other runs. This property is important for parallelizing parameter sweep applications because it means we can formulate this type of problem in a *leader-worker* paradigm. The SETI@Home project is a well-known leader-worker parallel application [1]. The SETI@Home servers at UC-Berkeley distribute jobs to computers around the world. None of the jobs communicate with each other; they only communicate their results back to the SETI@Home servers when they are done computing the job. Because each job is independent, it does not matter if the 415th job completes after the 420th job completes. It is only important that each job completes and its results are recorded. This type of application, in which individual processors do not communicate with each other during processing, is known as *embarrassingly parallel*. Parameter sweep applications are also embarrassingly parallel.

The rest of this paper is structured as follows. Section 2 describes the basic concept behind parallelizing parameter sweep applications. Section 3 introduces a basic serial parameter sweep application written in MATLAB. Section 4 demonstrates how to parallelize the serial code using pMatlab. Section 5 compares the results of the serial and parallel codes and Section 6 concludes with a summary.

This document assumes that the reader has a basic understanding of pMatlab. Before continuing, it is recommended that the reader first read [2], which provides an overview of parallel programming and introduction to pMatlab.

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.

2. Parallelization Process

A typical parameter sweep application consists of a for loop which repeatedly executes the same code, usually in a function. A unique set of arguments is supplied to the function in each iteration. Since the loop iterator has a unique value for each iteration, it can be used to compute the set of arguments. A more advanced parameter sweep application may nest multiple for loops and use multiple iterators to compute the input arguments.

This is the model used to parallelize parameter sweep applications with pMatlab. Each iteration is computed entirely on one processor; different loop iterations are processed on different processors. The advantage of using pMatlab to parallelize parameter sweep applications is its ability to abstract the mechanism for distributing data and computation across multiple processors. When the user creates a distributed matrix, i.e. a dmat, in pMatlab, he need only specify a map to describe how to distribute the dmat. After dmat is distributed, each processor operates on only its local section of the dmat.

3. Serial Code

This section presents code for a serial MATLAB program that implements a basic parameter sweep application. The serial parameter sweep application consists of two files:

- **`param_sweep_serial.m`** – This MATLAB script repeatedly calls `sample_function` within a for loop, supplying a unique set of input arguments in each iteration.
- **`sample_function.m`** – This MATLAB script contains the function the user wishes to parameter sweep.

In this example, `sample_function` is called 16 times within a for loop. `sample_function` accepts three inputs arguments and returns three output values. The output values for each iteration are stored in a 16x3 matrix, where each row stores the output of a unique iteration and each column stores one of the three outputs for a given iteration.

3.1. param_sweep_serial.m

This section describes `param_sweep_serial.m`. See Figure 1.

Lines 4-5 set the number of iterations to perform, `n`, and number of output arguments of `sample_function`, `m`. These values will be used to set the dimensions of the matrix that will store the output of all loop iterations. `n` specifies the number of rows in the matrix and `m` specifies the number of columns. Each row will store the outputs of a single iteration and each column will store one output. In the example, `sample_function` computes three outputs and is called 16 times.

Line 8 creates the output matrix, `z`, with size `m×n`. The matrix `z` will store the results of `sample_function`.

Lines 11-18 contain the for loop that calls `sample_function` `n` times, supplying different inputs for each iteration `ii`. Line 13 computes an argument value, `my_other_arg`, based on `ii`. Line 17 calls `sample_function`, passing it three input arguments. The first argument is

the loop iterator, `ii`. The second argument is set to 0. The third argument is `my_other_arg`. The reason for setting the second argument to 0 will become clear when the code is parallelized. The results of `sample_function` are returned and written to row `ii` in the output matrix `z`.

Line 21 indicates that the program completely successfully.

Line 24 displays the results of all `n` calls to `sample_function`.

3.2. sample_function.m

This section describes `sample_function`. See Figure 2.

`sample_function` simply returns the values of its three input arguments as output values. The first argument is the current loop iteration. The second argument is the rank of the local processor. The third argument is another argument whose value is computed based on the current loop iteration.

```
1:   % basic parameter sweep code (serial)
2:
3:   % Set data sizes.
4:   m = 3; % number of output arguments
5:   n = 16; % number of independent iterations
6:
7:   % Create z – data output matrix.
8:   z = zeros(n, m);
9:
10:  % Loop over the local indices
11:  for ii = 1:size(z, 1)
12:     % Calculate another argument
13:     my_other_arg = 2.5 * ii;
14:
15:     % Call a function with the index, and other arguments, and
16:     % store the result in a row
17:       z(ii, :) = sample_function(ii, 0, my_other_arg);
18:  end % for ii
19:
20:  % Finalize the program
21:  disp('SUCCESS');
22:
23:  % Finally, display the resulting matrix on the leader
24:  disp(z);
```

Figure 1 – Code for `param_sweep_serial.m`, basic serial MATLAB code that implements a parameter sweep application

```
1:   function [out] = sample_function(i_global, my_rank, my_other_arg);
2:
3:   out = zeros(1,3);
4:
5:   out(1) = i_global;
6:   out(2) = my_rank;
7:   out(3) = my_other_arg;
```

Figure 2 – Code for `sample_function.m`, sample function called by the parameter sweep application

4. Parallel Code

In this section, we will modify the serial code presented earlier so that it can run on multiple processors using pMatlab. The parallelized parameter sweep application consists of three files:

- **`RUN.m`** – This is the standard pMatlab script that launches the parameter sweep application. See Figure 3. For an explanation of `RUN.m`, refer to [1].
- **`param_sweep_parallel.m`** – This is a modified version of `param_sweep_serial`, using pMatlab to distribute computation and gather the results of `sample_function` from all processors.
- **`sample_function.m`** – This function is unchanged from the serial version.

Remember that pMatlab follows the single-program multiple-data (SPMD) model, in which the same program runs on all processors but each processor contains different data. Figure 5 graphically depicts the SPMD program flow of `param_sweep_parallel` and compares it with the program flow of `param_sweep_serial`. Each individual loop iteration is depicted, with each processor computing mutually exclusive sets of iterations simultaneously.

```
1:   % RUN is a generic script for running pMatlab scripts.
2:
3:   % Define number of processors to use
4:   Ncpus = 4;
5:
6:   % Name of the script you want to run
7:   mFile = 'param_sweep_parallel';
8:
9:   % Define cpus
10:
11:  % Run on user's local machine
12:  % cpus = {};
13:
14:  % Specify which machines to run on
15:  cpus = {'node-1', 'node-2', 'node-3', 'node-4'};
16:
17:  % Run the script.
18:  ['Running ' mFile ' on ' num2str(Ncpus) ' cpus']
19:  eval(pRUN(mFile, Ncpus, cpus));
```

Figure 3 – Code for `RUN.m`, pMatlab script to launch `param_sweep_parallel.m`

```
1:    % basic parameter sweep code
2:    %
3:    % Want to parallelize the following loop:
4:    % for ii = 1:n
5:    %     z(ii) = f(ii, otherArgs...)
6:    % end % for ii
7:
8:    % Turn parallelism on or off.
9:    PARALLEL = 1;
10:
11:   % Set data sizes.
12:   m = 3; % number of output arguments
13:   n = 16; % number of independent iterations
14:
15:   % Create Maps.
16:   map1 = 1;
17:   if (PARALLEL)
18:   % Break up rows.
19:      map1 = map([Ncpus 1], {}, 0:Ncpus-1 );
20:   end
21:
22:   % Create z - data output matrix.
23:   z = zeros(n, m, map1);
24:
25:   % Get the local portion of the global indices
26:   my_i_global = global_ind(z, 1);
27:
28:   % Get the local portion of the distributed matrix
29:   my_z = local(z);
30:
31:   % Loop over the local indices
32:   for i_local = 1:length(my_i_global)
33:      % Determine the global index for this (local) iteration
34:      i_global = my_i_global(i_local);
35:
36:      % Calculate another argument
37:      my_other_arg = 2.5 * i_global
38:
39:      % Call a function with the global index, and other arguments, and
40:      % store the result in a local row
41:      my_z(i_local, :) = sample_function(i_global, my_rank, my_other_arg);
42:   end % for i_local
43:
44:   % Store the local portion of z into the distributed matrix z
45:   z = put_local(z, my_z);
46:
47:   % Finally, aggregate all of the output onto the leader process
48:   z_final = agg(z);
49:
50:   % Finalize the pMATLAB program
51:   disp('SUCCESS');
52:
53:   % Finally, display the resulting matrix on the leader
54:   disp(z_final);
```

Figure 4 – Code for `param_sweep_parallel.m`, pMatlab version of the serial parameter sweep application in `param_sweep_serial.m`. Red and blue indicate lines that have been added and modified, respectively, in the original serial code.

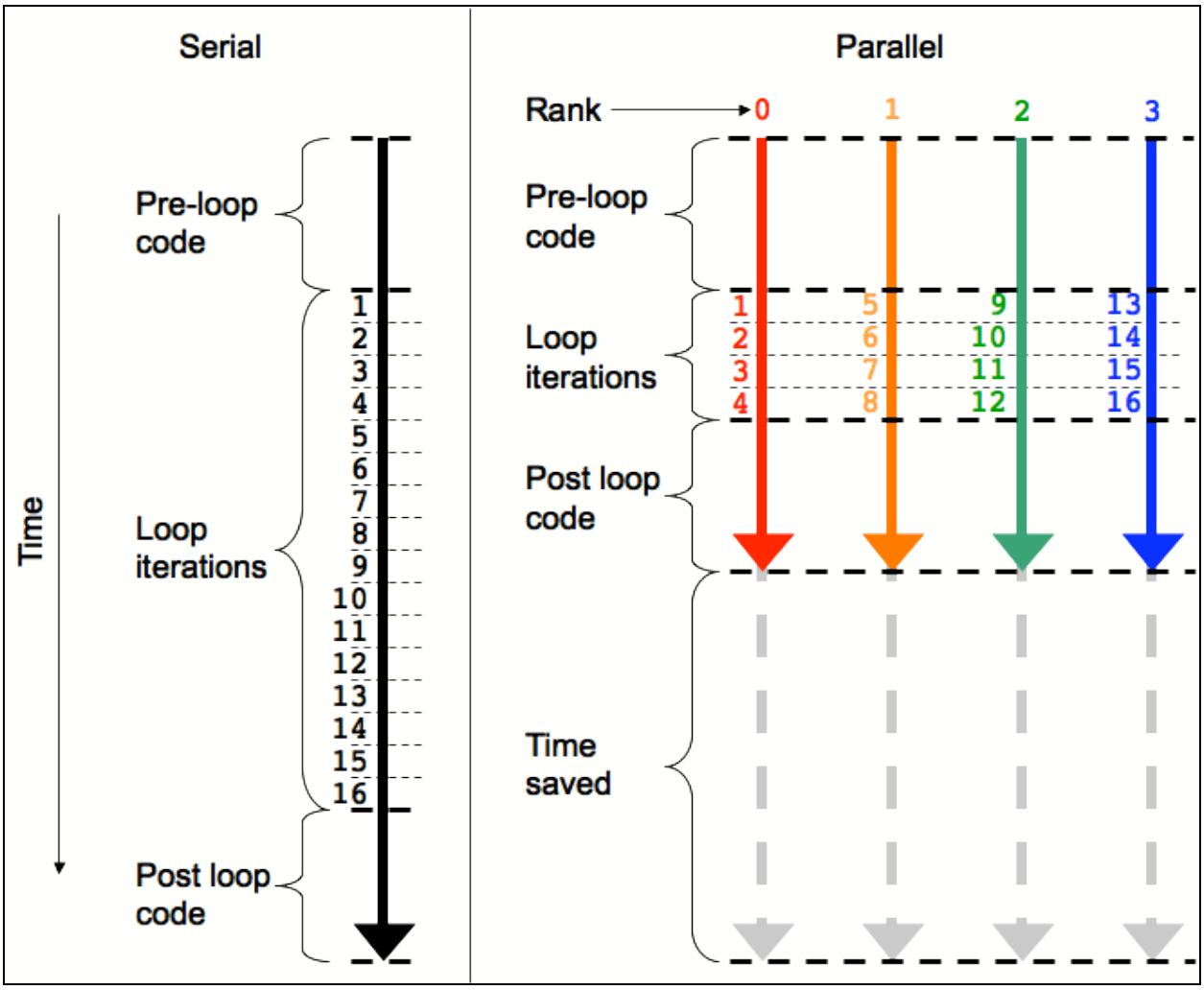

Figure 5 – Depiction of SPMD program flow for `param_sweep_parallel`.

4.1. param_sweep_parallel.m.

This section describes `param_sweep_parallel.m`. See Figure 4.

Line 9 enables or disables the pMatlab library. If `PARALLEL` is set to 0, then the script will not initialize the pMatlab library, call any pMatlab functions, or create any pMatlab data structures. It will simply run serial MATLAB code on the local processor. If `PARALLEL` is set to 1, then the pMatlab library will be initialized, dmats will be created instead of regular MATLAB matrices, and any functions that have dmat input arguments will call the associated pMatlab function.

Lines 12-13 set the number of iterations to perform, `n`, and number of output arguments of `sample_function`, `m`. These values will be used to set the dimensions of the matrix that will store the output of all loop iterations. `n` specifies the number of rows in the matrix and `m` specifies the number of columns. Each row will store the outputs of a single iteration and each column will store one output. In the example, `sample_function` computes three outputs and is called 16 times.

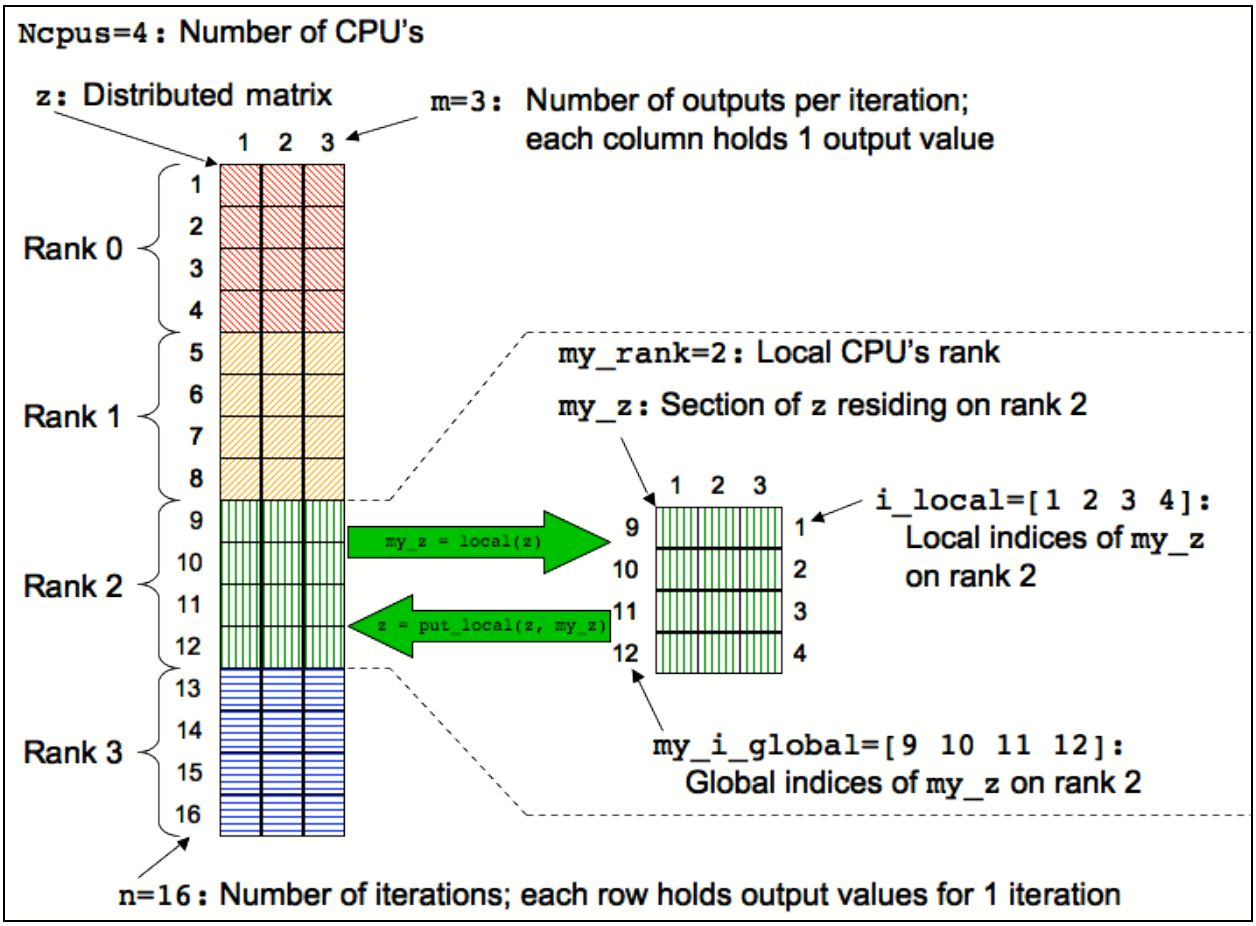

Figure 6 - Graphical depiction of data structures created in `param_sweep_parallel.m`.

Lines 16 through 20 specify the map, `map1`, for the output matrix. By default, `map1` is set to 1. If `PARALLEL` is set to 1, then a map object is constructed that distributes the output matrix's rows among processors, essentially distributing loops iterations among processors. See Figure 5.

Line 23 creates the output matrix, `z`, and initializes it to all zeros. Note that if `PARALLEL` is set to 0, then an `n×m×1` matrix is created, which is equivalent to a `n×m` matrix. If `PARALLEL` is set to 1, then `map1` is a map object and the pMatlab `zeros` function is called and creates a dmat object that distributes the matrix across all processors, as depicted in Figure 6.

Clearly, each processor owns only a portion of the global output matrix. Line 26 obtains the global indices in dimension 1 (i.e. rows) owned by just the local processor. In this example, the output matrix contains 16 rows distributed across four processors. Processor 0 owns rows 1 through 4, processor 1 owns 5 through 8, processor 2 owns 9 through 12 and processor 3 owns 13 through 16. Thus, `global_ind` will return the vector `[1 2 3 4]` on rank 0, `[5 6 7 8]` on rank 1, `[9 10 11 12]` on rank 2, and `[13 14 15 16]` on rank 3. For more details on global vs. local indices, see [2].

Line 29 copies the processor's local portion of the output dmat to a regular MATLAB matrix, `my_z`.

Lines 32 through 42 comprise the core of the parameter sweep application. These lines define the for loop that will compute all `n` iterations of `sample_function` on distributed over multiple processors. The key difference in programming serial and parallel parameter sweep applications is the concept of global and local indices iterations and indices in parallel programming.

At line 45, each processor has computed the results for its iterations. The `put_local` function copies the contents of `my_z` on each processor into the dmat `z`.

At line 48, the dmat `z` contains the results of all iterations of `sample_function`, but distributed across all processors. The `agg` function aggregates the contents of `z` into a regular MATLAB matrix, `z_final`, located on the leader process, rank 0. Note that on all other processors, `z_final` contains just the processor's local section of `z` instead of the entire contents of `z`.

Line 51 indicates that the program completed successfully.

Line 54 displays on Pid=0 the results of all `n` calls to `sample_function`. Note that since `z_final` is empty on all other processors, just the local portion of `z` will be displayed.

```
          param_sweep_serial

SUCCESS
    1.0000         0    2.5000
    2.0000         0    5.0000
    3.0000         0    7.5000
    4.0000         0   10.0000
    5.0000         0   12.5000
    6.0000         0   15.0000
    7.0000         0   17.5000
    8.0000         0   20.0000
    9.0000         0   22.5000
   10.0000         0   25.0000
   11.0000         0   27.5000
   12.0000         0   30.0000
   13.0000         0   32.5000
   14.0000         0   35.0000
   15.0000         0   37.5000
   16.0000         0   40.0000
```

```
          param_sweep_parallel

SUCCESS
    1.0000         0    2.5000
    2.0000         0    5.0000
    3.0000         0    7.5000
    4.0000         0   10.0000
    5.0000    1.0000   12.5000
    6.0000    1.0000   15.0000
    7.0000    1.0000   17.5000
    8.0000    1.0000   20.0000
    9.0000    2.0000   22.5000
   10.0000    2.0000   25.0000
   11.0000    2.0000   27.5000
   12.0000    2.0000   30.0000
   13.0000    3.0000   32.5000
   14.0000    3.0000   35.0000
   15.0000    3.0000   37.5000
   16.0000    3.0000   40.0000
```

Figure 7 – Comparison of outputs for the serial and parallel parameter sweep applications.

5. Serial vs. Parallel

Figure 7 compares the results of `param_sweep_serial` and `param_sweep_parallel`. To reiterate, the first, second and third columns contain the iteration number, the rank of the processor that computed that iteration, and the value computed based on the iteration, respectively. Note that the values for the first and second columns are the same. Only the second column, the processor ranks, differs. This shows that different iterations are indeed

computed on different processors, but that as long as the results are not dependent which processor they were computed on, the results are the same.

6. Adding Parallelism

One tenet of good software engineering is that programs should not be run on full scale inputs immediately. Rather, programs should initially be run on a small test problem to verify functionality and to validate against known results. The program should be scaled to larger inputs until the program is fully validated and ready to be run at full scale. The same is true for parallel programming. Parallel programs should not run at full scale on 32 processors as soon as the programmer has finished taking a first stab at writing the application. Both the test input and number of processors should be gradually scaled up.

The following is the recommended procedure for adding parallelism to a pMatlab application. This procedure gradually adds complexity to running the application.

1. **Run with 1 processor on the user's local machine with the pMatlab library disabled.** This tests the basic serial functionality of the code.
2. **Run with 1 processor on the local machine with pMatlab enabled.** Tests that the pMatlab library has not broken the basic functionality of the code.
3. **Run with 2 processors on the local machine.** Tests the program's functionality works with more than one processor without network communication.
4. **Run with 2 processors on multiple machines.** Test that the program works with network communication.
5. **Run with 4 processors on multiple machines.**
6. **Increase the number of processors, as desired.**

Figure 8 shows the sequence of parameters that should be used to scaling pMatlab applications. See [2] for examples on how to set these parameters and for more details on how to apply good software engineering practices to pMatlab.

	In pMatlab code	In `RUN.m`	
1.	`PARALLEL = 0;`	`Ncpus = 1;`	`cpus = {};`
2.	`PARALLEL = 1;`	`Ncpus = 1;`	`cpus = {};`
3.	`PARALLEL = 1;`	`Ncpus = 2;`	`cpus = {};`
4.	`PARALLEL = 1;`	`Ncpus = 2;`	`cpus = {'node1','node2'};`
5.	`PARALLEL = 1;`	`Ncpus = 4;`	`cpus = {'node1','node2'};`
6.	`PARALLEL = 1;`	`...`	`cpus = {'node1','node2'};`

Figure 8 – Example sequence of parameters for scaling parallel programs.

7. Conclusion

In this is paper we accomplished the following:

- Defined parameter sweep applications
- Presented an example serial parameter sweep application implemented in MATLAB
- Parallelized the serial example using pMatlab
- Compared the results of the serial and parallel parameter sweep applications

The applications discussed in this paper are only simple examples of how pMatlab can be used to parallelize one type of embarrassingly parallel application. However, the basic concepts presented can be easily expanded to encompass more complex applications. For more information on specific pMatlab functions discussed here, see [3].

8. References

[1] http://setiathome.ssl.berkeley.edu
[2] H. Kim. "Introduction to Parallel Programming and pMatlab." MIT Lincoln Laboratory.
[3] H. Kim, N. Travinin. "pMatlab Function Reference." MIT Lincoln Laboratory.