

# SCAI

## *NetWorkSpaces<sup>®</sup> for MATLAB*

### User Guide

**SCIENTIFIC**  
**COMPUTING ASSOCIATES**  
INC.

One Century Tower  
265 Church Street, New Haven, CT 06510-7010  
Tel: (203) 777-7442 • Fax (203) 776-4074

[www.LindaSpaces.com](http://www.LindaSpaces.com)

---

# Acknowledgements

**D. Atapattu, N. Carriero, J. Lai, S. Weston**  
Scientific Computing Associates, Inc.

© Copyright 2006 Scientific Computing Associates, Inc. All rights reserved. SCAI, NetWorkSpaces, and Sleigh are trademarks of Scientific Computing Associates, Inc. All other trademarks and registered trademarks are property of their respective holders.

Manual version 1.0 January 2006. Printed in the USA.

# Contents

<b>Introduction .....</b>	<b>1</b>
<b>Getting Started .....</b>	<b>5</b>
Prerequisites .....	5
NetWorkSpaces Server.....	5
Starting the Server.....	5
Stopping the Server .....	6
NetWorkSpaces Client .....	6
Installing the Client .....	6
Setting MATLABPATH.....	6
Starting the NetWorkSpaces Client .....	7
Starting Sleigh .....	8
Setting Up a Password-less SSH Login .....	8
Using an Alternative to SSH .....	9
<b>Tutorials .....</b>	<b>11</b>
NetWorkSpaces for MATLAB Tutorial.....	11
Sleigh for MATLAB Tutorial.....	14
Web Launch.....	17
<b>Appendix .....</b>	<b>19</b>
NetWorkSpaces for MATLAB Reference .....	19
netWorkspace Class .....	19
netWorkspace .....	19
fetch.....	20
fetchTry.....	21
find.....	22
findTry .....	22
store .....	23
declare .....	24
deleteVar.....	25
listVars.....	25
currentWs .....	26
close.....	26
server .....	27
nwsServer Class.....	28
nwsServer.....	28
openWs .....	28
useWs .....	29
listWss .....	30
mktempWs .....	31

socket.....	31
deleteNws.....	32
Sleigh Class.....	33
sleigh.....	33
eachWorker.....	34
eachElem.....	36
nws.....	39
stop.....	39
check.....	39
wait.....	40

# Introduction

**NetWorkSpaces (NWS)** provides a framework to coordinate programs written in scripting languages; NWS currently supports the languages Python, MATLAB, and R. This User Guide is for the MATLAB system, and it assumes that the reader is familiar with MATLAB.

A MATLAB program uses variables to communicate data from one part of the program to another. For example, `x = 123` assigns the value `123` to the variable named `x`. Later portions of the program can reference `x` to use the value `123`. This mechanism is generally known as *binding*. In this case, the binding associates the value `123` with the name `x`. The collection of name-value bindings in use by a program is often referred to as its "workspace."

Two or more MATLAB programs use **NWS** to communicate data by means of name-value bindings stored in a network-based workspace (a *NetWorkSpace*, which in MATLAB is an instance of a `NetWorkspace` object). One program creates a binding, while another program reads the value the binding associated with the name. This is clearly quite similar to a traditional workspace; however, a `NetWorkspace` differs from a traditional workspace in two important ways.

First, in a setting in which two or more MATLAB programs are interacting, it would not be unusual for one to attempt to "read" the value of a name before that name has been bound to a value. Rather than receiving an "unbound variable" error, the reading program (by default) simply waits (or "blocks") until the binding occurs. Second, a common usage paradigm involves processing a sequence of values for a given name. One MATLAB program carries out a computation based on the first value, while another might carry out a computation on the second, and so on. To facilitate this paradigm, more than one value may be bound to a name in a workspace and values may be "removed" (fetch) as opposed to read (find). By default, values bound to a name are consumed in first-in-first-out (FIFO) order, but other modes are supported: last-in-first-out (LIFO), multiset (no ordering implied) and single (only the last value bound is retained). Since all its values could be removed, a name can, in fact, have no values associated with it.

A `NetWorkspace` provides five basic operations: `store()`, `fetch()`, `fetchTry()`, `find()`, and `findTry()`.

`store()` introduces a new binding for a specific name in a given workspace.

`fetch()` fetches (removes) a value associated with a name.

`find()` reads a value without removing it.

---

Note that `fetch()` and `find()` block if no value is bound to the name. `fetchTry()` and `findTry()` are non-blocking; they return an empty value or user-supplied default if no value is available.

There are several additional `NetWorkspace` operations:

`close()` closes the connection to a workspace. Depending on the ownership, closing a connection to a workspace can result in removing the workspace.

`currentWs()` returns the name of the specified workspace.

`declare()` declares a variable name with a specific mode.

`deleteVar()` deletes a name from a workspace.

`listVars()` provides a list of variables (bindings) in a workspace.

`server()` returns the server a workspace connects to.

In addition to a `NetWorkspace`, a MATLAB client of NWS also uses an `NWSserver` object. This object is created automatically when a new `NetWorkspace` object is created, so you don't need to interact directly with it. However, if you need to access the server object, you can invoke the `server` method to get a handle to the server object.

A `NwsServer` object supports the following actions:

`openWs()` connects to a workspace or creates one if the specified workspace does not exist.

`useWs()` uses a `NetWorkspace` without claiming ownership.

`deleteWs()` explicitly deletes a workspace.

`listWss()` provides a list of workspaces in the server.

The operations above enable coordination of different programs using NWS. There is also a mechanism, built on top of **NWS**, called **Sleigh** (inspired by R's `SNOW` package) to enable parallel function evaluation. **Sleigh** is especially useful for running "embarrassingly parallel" programs on multiple networked computers. Once a sleigh is created by specifying the nodes that are participating in computations, you can use:

`eachElem()` to invoke a function on each element of a list, with the invocations being evaluated concurrently by the sleigh participants.

`eachWorker()` to have each participant in a sleigh invoke a function. You can use this operation, for example, to build up requisite state prior to an `eachElem()` invocation.

---

Various data structures (workspaces, name-value bindings, etc.) in a NWS server can be monitored and even modified using a Web interface. In distributed programming, even among cooperating programs, the state of shared data can be hard to understand. The Web interface presents a simple way of checking current state remotely. This tool can also be used for learning and debugging purposes, and for monitoring MATLAB programs as well.



# 2

## Getting Started

This chapter provides step-by-step procedures for setting up *NetWorkSpaces* and *Sleigh* for MATLAB.

### Prerequisites

*NetWorkSpaces* and *Sleigh* run on all Linux and Windows XP platforms. To use them, you must install the following software:

- MATLAB 7.0 or above.
- *NetWorkSpaces* server — <http://www.sourceforge.net/projects/nws-r>.

The *NetWorkSpaces* server also requires the following:

- Python 2.4 — <http://www.python.org>  
We recommend ActiveState Python — <http://www.activestate.com> for Windows.
- Twisted Framework — <http://twistedmatrix.com>.

### NetWorkSpaces Server

#### Starting the Server

There are two ways to start a *NetWorkSpaces* server:

- Use the `twistd` command:

```
% twistd -noy /etc/nws.tac
```

- Execute the `nws` script on Linux platform:

```
% nws start
```

This starts a *NetWorkSpaces* server on localhost at port 8765.

---

## Stopping the Server

There are also two ways to stop a *NetWorkSpaces* server:

- If you used the `twistd` command to start the server:

```
% kill `cat twistd.pid`
```

- If you used the `nws` script to start the server:

```
% nws stop
```

## NetWorkSpaces Client

### Installing the Client

To install *NetWorkSpaces* on Linux platforms:

1. Untar the `nws` package: `tar xzvf nws_matlab_1.0.tar.gz`
2. Run the install script `install.sh`

To install *NetWorkSpaces* on Windows XP

1. unzip the `nws` package.

### Setting MATLABPATH

Make sure `MATLABPATH` contains the location of installed `NetWorkSpaces` and `sleighCommon` directory. If `MATLABPATH` this is not setp correctly, then MATLAB will not know anything about *NetWorkSpaces* and *Sleigh*.

For example, for Linux platform, if *NetWorkSpaces* is installed in `/usr/local/nws`, then add the following statement to your shell startup script:

```
export MATLABPATH=/usr/local/nws:  
/usr/local/nws/sleighCommon:$MATLABPATH
```

For Windows XP, open up a MATLAB session, and follow these steps:

1. Click the **File** menu.
2. Click **Set Path**.
3. Click the **Add Folder** button.
4. Select the location of `nws`.

- 
5. Repeat step 1-3, and then select the location of `sleighCommon` folder.
  6. Click **Save**.

## Starting the NetWorkSpaces Client

Once you've got a NetWorkspace server up and running, you're ready to use *NetWorkSpaces*.

1. Start up a MATLAB session.
2. Type the following:

```
>> ws = netWorkspace('matlab space');  
>> store(ws, 'x', 1);
```

This step creates a workspace named 'matlab space' and stores a variable `x` with value 1 to the workspace. If you've encountered error messages while creating the workspace, it's likely that you didn't set up `MATLABPATH` correctly. Please review the [Setting MATLABPATH](#) section.

You can also view what's in the workspace using a web interface. To do this, you point your browser to [http://server\\_host\\_name:8766](http://server_host_name:8766), where `server_host_name` is the machine that a *NetWorkSpaces* server resides on.

To examine values that you've created in a workspace using the server's web interface, you also need a *babelfish*. The *babelfish* translates values into a human readable format so they can be displayed in a web browser. If a value is a string, then the web interface simply displays the contents of the string, without any help from the *babelfish*. But, if the value is any other type of MATLAB object, it needs help from the MATLAB *babelfish*. To start up *babelfish*, execute the following command in another MATLAB prompt:

```
>> babelfish
```

**Note:** This function will not return until you exit out of MATLAB.

For more information about *NetWorkSpaces*, see the [Tutorials](#) chapter.

---

## Starting Sleigh

*Sleigh* is a MATLAB class, built on top of the *NetWorkSpaces*, that makes it very easy to write simple parallel programs. *Sleigh* has concept of one master and multiple workers. The master sends jobs to workers that may or may not be on the same machine as the master. To enable the master to communicate with workers, *Sleigh* uses **SSH** mechanism to run jobs remotely. Therefore, before you can start *Sleigh*, you need to setup a password-less **SSH** login.

**Note:** If you use Windows, where SSH is not available, please refer to the section [Using an Alternative to SSH](#).

### Setting Up a Password-less SSH Login

To generate public and private keys, follow the steps below.

1. `ssh-keygen -t rsa`
2. `cd ~/.ssh` (The `.ssh` directory is located in your HOME directory.)
3. `cp id_rsa.pub authorized_keys` (This step allows password-less login to a local machine.)
4. For all the remote machines for which you want password-less login, append the content of `id_rsa.pub` to their `authorized_keys` file.

To test the password-less login, type the following command:

```
% ssh hostname date
```

If everything is setup correctly, you should not be asked for password and the current date on remote machine will be returned.

---

## Using an Alternative to SSH

If you're running *Sleigh* on a platform that does not support [SSH](#), *Sleigh* provides an alternative web launch option.

To use the web launch option, follow the steps below.

1. Start Sleigh using web launch:

```
>> opt.launch = 'web';  
>> s = sleigh({}, opt);
```

The *Sleigh* constructor does not return until it gets a signal that all workers have started and are ready to accept jobs.

2. Log in to a remote machine.
3. Start a MATLAB session.
4. Open a web browser and point to `http://server_host_name:8766`
5. Click on the newly created Sleigh workspace, and read the value from variable `runMe`. It usually has a value similar to:  
`webLaunch('sleigh_ride_0000000004_tmp1a6c0h', 'mercury', 8765);`
6. Copy the `runMe` value to the MATLAB session.
7. Repeat the previous steps for each worker that needs to be started.
8. Once all workers have started, delete the `DeleteMeWhenAllWorkersStarted` variable from the Sleigh workspace. This signals the *Sleigh* master that the workers have started and are ready to accept work.

Now you're ready to send jobs to remote workers. See the [Sleigh for MATLAB Tutorial](#) section in the [Tutorials](#) chapter for more information.



## NetWorkSpaces for MATLAB Tutorial

*NetWorkSpaces* (*NWS*) is an MATLAB package that makes it very easy to communicate and coordinate multiple MATLAB programs running on different machines. In this chapter, we will give a small walk-through of writing a simple program using NetWorkSpaces. The appendix, [NetWorkSpaces for MATLAB Reference](#), contains a detailed reference of the `NetWorkspace` class.

First, start an interactive MATLAB session.

Next, create a workspace called 'bayport':

```
>> ws = netWorkspace('bayport');
```

This workspace is created on a NWS server located on `localhost`, port `8765`. Additional arguments can be passed to the `netWorkspace` constructor to specify a NWS server located on different hostnames and ports. Please refer to the appendix, [NetWorkSpaces for MATLAB Reference](#), for more examples.

Once we have a workspace, we can write data into a variable using the `store` method:

```
>> store(ws, 'joe', 17);
```

The variable 'joe' now has a value of `17` in the workspace named 'bayport'.

To find out the value associates with the variable 'joe', we use the `find` method:

```
>> age = find(ws, 'joe');
```

which sets 'age' to `17`.

Note that the `find` method will block until the variable 'joe' has a value. This is important when you're trying to read that variable from a different machine. If it didn't block, you might have to repeatedly try to read the variable until it succeeded. Of course, there are times when you don't want to block, but just want to see if some variable has a value. You do that using the `findTry` method.

```
>> age = findTry(ws, 'chet');
```

---

This assigns '0' to `age`, since we didn't store any value to variable `'chet'`. If you'd rather have `findTry` return some other value when the variable doesn't exist, you can pass in an extra argument to the `findTry` method:

```
>> age = findTry(ws, 'chet', 1);
```

which assigns `1` to `age` if the variable `'chet'` doesn't exist.

So far, we've carefully executed the `store` method once for each variable created. What if we execute the `store` method more than once? Then, it depends on the mode of the variable. By default, a variable created by the `store` method uses first-in-first-out (FIFO), which means the first stored value in the workspace will be the first one to be read. If you want to use a different mode for the variable, you have to use the `declare` method to create the variable first. Please refer to the [NetWorkSpaces for MATLAB Reference](#) for descriptions on different modes.

Since the `find` method does not remove value from the workspace, we cannot use `find` to read multiple values associated with a variable. To read through all the values, we have to use the `fetch` method. `fetch` works the same as `find`, but in addition, it removes the value. This can be useful in sending a sequence of messages from one program to another.

Let's try writing multiple values to a variable:

```
>> n = [16 19 25 22]
>> for i = 1:length(n)
    store(ws, 'biff', n(i))
end
```

To read the values, we just call `fetch` repeatedly:

```
>> n = []
>> for i = 1:4
    n = [n fetch(ws, 'biff')]
end
```

If we didn't know how many values were stored in a variable, we could have done the following:

```
>> n = []
>> while 1
    t = fetchTry(ws, 'biff');
    if t==0 break; end;
    n = [n t];
end
```

This uses `fetchTry`, which works like `fetch`, except that it is non-blocking.

---

These are the basic operations provided by NWS. It's a good idea to play around with these operations using two MATLAB sessions. That way, you can really transfer data between two different programs. Also, you can see how the blocking operations work. In the examples above, we were careful never to block because we were only using one MATLAB session.

To use two MATLAB sessions, just execute MATLAB in another window, export `MATLABPATH` or have it saved and load up automatically when a new window is opened. Then, open the 'bayport' workspace. This can be done with the same command that we used previously, but this time, the command:

```
>> ws = netWorkspace('bayport')
```

won't create the 'bayport' workspace, since it already exists. Now you can execute a operation such as:

```
>> x = fetch(ws, 'frank')
```

in one session, watch it block for a minute, and then execute `store` in the other session:

```
>> store(ws, 'frank', 18)
```

and see that the fetch in the first session completes.

While you're experimenting with these operations, it can be helpful to use the NWS server's web interface to see what's going on in your workspaces. Just point your web browser to the URL:

```
http://localhost:8766
```

If you're using a browser on a machine other than the NWS server, you have to use the appropriate hostname, rather than `localhost`.

---

## Sleigh for MATLAB Tutorial

*Sleigh* is a MATLAB class, built on top of *NWS*, that makes it easy to write simple parallel programs. It provides two basic functions for executing tasks in parallel: `eachElem` and `eachWorker`.

`eachElem` is used to execute a specified function multiple times in parallel with a varying set of arguments.

`eachWorker` is used to execute a function exactly once on every worker in the sleigh with a fixed set of arguments.

`eachElem` is all that is needed for most basic programs, so that is what we'll start with.

First, we need to start up a sleigh:

```
>> s = sleigh
```

This starts two sleigh workers on the local machine using ssh, but a cell array argument can be used to specify different machines to start the workers. If this fails, it's probably because SSH isn't properly configured or supported. If SSH is not supported, you can use web launch, which described below, as an alternative solution.

Let's shut down sleigh so we can start workers on different machines.

```
>> stop(s)
```

This deletes the sleigh's NWS workspace, and shuts down all of the sleigh worker processes.

Now we'll make a new sleigh, with workers on `node1` and `node2`, and we'll use an NWS server that's running on `node10`:

```
>> opt.nwsHost = 'node10'  
>> s = sleigh({'node1', 'node2'}, opt);
```

Now we're ready to run a parallel program. Here it is:

```
>> result = eachElem(s, 'add1', 1:10);
```

In this simple command, we have defined a set of data that is processed by multiple workers in parallel, and each of the results is returned in a cell array. Of course, you would never really bother to do such a trivial amount of work with a parallel program, but you get the idea.

---

This `eachElem` function puts 10 tasks into the sleigh workspace. Each task contains one value from 1 to 10. This value is passed as the argument to the `add1` function. The return value of the function is put into the sleigh workspace. The `eachElem` function waits for all of the results to be put into the workspace, and returns them as a cell array, which are numbers from 2 to 11.

Note that `add1` is an M function that looks like this:

```
function y = add1(x)
y = x+1;
```

As a second example, let's add two lists together.

We first defined an `add2` function:

```
function z = add2(x, y)
z = x + y;
```

Then, we invoke the function with two variable arguments. Since we have two variable arguments, we have to enclose two vectors in a cell array.

```
>> result = eachElem(s, 'add2', {1:5, 6:10});
```

This is the parallel equivalent to the MATLAB command:

```
>> result = (1:5) + (6:10);
```

We can keep adding more list arguments this way, but there's also a way to add arguments that are the same for every task, which we call fixed arguments:

```
>> result = eachElem(s, 'add2', 1:10, 1);
```

This is equivalent to the MATLAB command:

```
>> result = (1:10) + 1;
```

It is also equivalent to invoking `add1` function we defined previously.

```
>> result = eachElem(s, 'add1', 1:10)
```

The order of the arguments passed to the function are normally in the specified order, which means that the fixed arguments always come after the varying arguments. To change this order, a permutation vector can be specified. The permutation vector is specified using the `argPermute` field in the `execOptions` parameter.

---

For example, to perform the parallel equivalent of the MATLAB operation `100 - (1:10)`, we do the following:

```
>> opt.argPermute = [2 1];
>> result = eachElem(s, 'sub2', 1:10, 100, opt);
```

This permutation vector says to use the second argument first, and then use the first argument second, reversing the order of two arguments. The `sub2` function used in this example looks like this:

```
function z = sub2(x, y)
z = x - y;
```

There is another keyword argument, called `blocking`, which, if set to `0`, will make `eachElem` return immediately after submitting the tasks, thus making it non-blocking. A `pending` object is returned, which can be used periodically to check how many of the tasks are completed, and also to wait until all tasks are finished. Here's a quick example:

```
>> opt.blocking = 0;
>> sp = eachElem(s, 'add2', {1:1000, 1001:2000}, opt)
>> while check(sp) > 0
% Do something useful for a little while
% pass
>> result = wait(sp)
```

There is also a keyword argument called `loadFactor` that enables watermarking. This limits the number of tasks that are put into the workspace at the same time. This could be important if you're executing a lot of tasks. For example, setting the `loadFactor` to `3` limits the number of tasks in the workspace to 3 times the number of workers in the sleigh workspace. Here's how to do it:

```
>> opt.loadFactor = 3;
>> result = eachElem(s, 'add2', {1:100, 101:200}, opt);
```

The results are exactly the same as not using a load factor. Setting this option only changes the way that tasks are submitted by the `eachElem` function.

As you can see, Sleigh makes it very easy to write simple parallel programs. But you're not limited to simple programs. You can use NWS operations to communicate between the worker processes, allowing you to write message passing parallel programs much more easily than using MPI or PVM, for example.

---

## Web Launch

Sleigh supports web launch, in case SSH is not supported on the platform on which you're running. To use web launch, you set the `launch` field in the `opts` argument to `'web'`.

```
>> opt.launch = 'web';  
>> s = sleigh({}, opt);
```

Sleigh constructor on the master node is blocked until you tell it that all workers are ready to receive work.

Next, go to a remote worker machine and start an MATLAB session there. Then, open up a browser and point it to the server host at port 8766. Click on the sleigh workspace that's created. It always has a name similar to `sleigh_ride_000000004_tmp1a6c0h`. The sleigh workspace should have a variable called `runMe` that contain a value similar to this: `webLaunch('sleigh_ride_000000004_tmp1a6c0h', 'mercury', 8765);`.

Copy this value to the MATLAB session on remote machine. This essentially starts up a Sleigh worker on the remote machine. Do the same thing (copy and paste the 'runMe' value) to as many workers (remote machines) as you would like to start. Once you have started up all workers, you can then delete the variable called 'DeleteMeWhenAllWorkersStarted' in the sleigh workspace. This signals sleigh master to unblock and can start sending tasks to workers.



## NetWorkSpaces for MATLAB Reference

### netWorkspace Class

#### netWorkspace

##### Description

netWorkspace class constructor.

##### Usage

```
ws = netWorkspace(wsName, h, p); or  
ws = netWorkspace(wsName, opts);
```

##### Arguments

<code>wsName</code>	name of the workspace.
<code>h</code>	server host name.
<code>p</code>	server port.
<code>opts</code>	an optional structure of fields. see details.

##### Details

`netWorkspace(ws, h, p)` creates an object for the shared netWorkspace `wsName` residing in the netWorkspace server running on host `h`, port `p`. If host name and port number are not provided, then `localhost` is used as default value for host name, and `8765` is used as default value for port.

`opts` is an optional structure of fields. If `opts` is presented, then it is assumed that an `nws` server object or host name and port number is provided as part of structure field. Therefore, `h` and `p` CANNOT be provided as part of arguments. It can ONLY provided as part of `opts` fields.

`opts` may contain these fields:

<code>server</code>	<code>nwsServer</code> class object. This field is not needed, if host name and port number are provided.
---------------------	---

---

<code>host</code>	server host name. Default value is <code>localhost</code> .
<code>port</code>	server port. Default value is <code>8765</code> .
<code>persistent</code>	If true, make the workspace persistent. A persistent workspace won't be purged when the owner disconnects from an nws server. Note that only the client who actually takes ownership of the workspace can make the workspace persistent. The <code>persistent</code> argument is effectively ignored if <code>useUse</code> is true.
<code>useUse</code>	By default 'open' semantic is used when creating a workspace (i.e., trying to claim ownership). If <code>useUse</code> is set to 1, 'use' semantic applies, which means no ownership is claimed.

### *Examples*

Example 1: create a workspace with default options

```
>> nws = netWorkSpace('nws example')
```

Example 2: connect to a workspace using `useUse` semantic

```
>> opt.useUse = 1
>> nws = netWorkSpace('example2', opt)
```

Example 3: create a workspace on an nws server that resides on `node1`, port `5555`.

```
>> nws = netWorkSpace('example3', 'node1', 5555)
```

## **fetch**

### *Description*

Fetch something from the shared `netWorkSpace` `nws`.

### *Usage*

```
X = fetch(nws, varName);
```

### *Arguments*

`nws` a `netWorkSpace` class object.

`varName` name of the variable to fetch.

---

### Details

`fetch` blocks until a value for `varName` is found in the shared `netWorkspace nws`. Once found, remove a value associated with `varName` from the shared `netWorkspace` and return it in `X`. This operation is atomic. If multiple Distributed Computing Toolbox (DCT) sessions `fetch` or `fetchTry` a given `varName`, any given value from the set of values associated with `varName` will be return to just one DCT session. If there is more than one value associated with `varName`, the particular value removed depends on `varName`'s behavior. See `declare(nws,...)` for details.

### Examples

```
>> nws = netWorkspace('nws example');
>> store(nws, 'x', 1); % store variable x with value 1 to nws
>> X = fetch(nws, 'x'); % X = 1
>> X = fetch(nws, 'x'); % no value for x; block on fetch
```

## fetchTry

### Description

Attempt to fetch something from the shared `netWorkspace nws`; anon-blocking version of `fetch(nws, ...)`.

### Usage

```
[X, F] = fetchTry(nws, varName, ifMissing);
```

### Arguments

`nws` a `netWorkspace` class object.

`varName` name of variable to fetch.

`ifMissing` value to return, if variable by `varName` is not found in `netWorkspace nws`. The default value is `0`.

### Details

Look in the shared `netWorkspace nws` for a value bound to `varName`. If found, remove a value associated with `varName` from `nws` and return it in `X`. Set `F` to true. This operation is atomic. If multiple DCT sessions `fetch` or `fetchTry` a given `varName`, any given value from the set of values associated with `varName` will be return to just one DCT session. If there is more than one value associated with `varName`, the particular value removed depends on `varName`'s behavior. See `declare(nws,...)` for details.

If not found, return the value of `ifMissing` in `X` and set `F` to false.

If `ifMissing` is not given, `0` is used.

---

## Examples

```
>> nws = netWorkspace('nws example');
>> store(nws, 'x', 1);
>>
>> [X, F] = fetchTry(nws, 'x'); % X = 1, F = 1
>> [X, F] = fetchTry(nws, 'x'); % X = 0, F = 0
```

## find

### Description

Find something in the shared netWorkspace nws.

### Usage

```
X = find(nws, varName);
```

### Arguments

**nws** a netWorkspace class object.

**varName** name of variable to find.

### Details

`find(nws, varName)` blocks until a value for `varName` is found in the shared netWorkspace `nws`. Once found, return in `X` a value associated with `varName`, but the value is not removed from netWorkSpce `nws`. If there is more than one value associated with `varName`, the particular value returned depends on `varName`'s behavior. See [declare\(nws, ...\)](#) for details.

### Examples

```
>> nws = netWorkspace('nws example');
>> store(nws, 'x', 1);
>> X = find(nws, 'x'); % X = 1
>> listVars(nws); % X value is not removed from nws
```

## findTry

### Description

Attempt to find `varName` in the shared netWorkspace `nws`; a non-blocking version of `find(nws, ...)`.

### Usage

```
[X, F] = findTry(nws, varName, isMissing);
```

---

### Arguments

<code>nws</code>	a <code>netWorkspace</code> class object.
<code>varName</code>	name of variable to find.
<code>isMissing</code>	value to return if no variable by <code>varName</code> is found in <code>netWorkspace nws</code> . The default value is <code>0</code> .

### Details

`[X, F] = findTry(nws, varName, isMissing)` looks in the `nws` for a value bound to `varName`. If found, returns a value associated with `varName` in `X` and sets `F` to true. No value is removed from `netWorkspace nws`. If there is more than one value associated with `varName`, the particular value returned depends on `varName`'s behavior. See [declare\(nws, ...\)](#) for details.

If not found, return the value of `isMissing` in `X` and set `F` to false.

If `isMissing` is not given, `0` is used.

### Examples

```
>> nws = netWorkspace('nws example');  
>> [X, F] = findTry(nws, 'x'); % check if variable x exists
```

## store

### Description

Store something in a `netWorkspace`.

### Usage

```
store(nws, varName, value);
```

### Arguments

<code>nws</code>	a shared <code>netWorkspace</code> class object.
<code>varName</code>	variable name.
<code>value</code>	value to store in the variable.

### Details

`store(nws, varName, v)` associates the value `v` with the variable `varName` in the shared `netWorkspace nws`, thereby making the value available to all the instances of MATLAB running in the current Distributed Computing Toolbox (DCT) session. `value` may be any MATLAB value—matrix, structure, function handle, etc.

---

If a mode has not already been set for `varName`, `fifo` will be used (see `declare(nws, ...)`). If no variable name is given, then value `v` must itself be a variable, and the value `v` will be associated with the variable's name in the netWorkspace `nws`. See the examples below for details.

Note that, by default ('fifo' mode), `store` is not idempotent: repeating `store(nws, 'X', v)` will add additional values to the set of values associated with the name 'X'. See the examples below for details.

### *Examples*

```
>> M = magic(5);  
>> store(nws, M);
```

and

```
>> store(nws, 'M', magic(5));
```

do the same thing.

```
>> for x = 1:10  
store(nws, 'numList', x);  
end
```

will associated ten distinct values with the name `numList`.

```
>> for i = 1:10  
fetch(nws, 'numList');  
end
```

will retrieve these distinct values, one at a time from the first to last stored.

## **declare**

### *Description*

Declare a variable with a particular mode in a shared netWorkspace.

### *Usage*

```
X = declare(nws, varName, mode);
```

### *Arguments*

`nws` a shared netWorkspace class object.

`varName` variable name.

`mode` variable mode.

---

### *Details*

If `varName` has not already been declared in `nws`, the behavior of `varName` will be determined by `mode`, which can be `'fifo'`, `'lifo'`, `'multi'`, or `'single'`. In the first three cases, multiple values can be associated with `varName`. When a value is retrieved for `varName`, the oldest value stored will be used in `'fifo'` mode, the youngest in `'lifo'` mode, and a nondeterministic choice will be made in `'multi'` mode. In `'single'` mode, only the most recent value is retained.

### *Examples*

```
>> nws = netWorkspace('nws example');
>> declare(nws, 'pi', 'single');
>> store(nws, 'pi', 2.171828182);
>> store(nws, 'pi', 3.141592654);
```

`listVars(nws)` will show that only the most recent value of `pi` is retained.

## **deleteVar**

### *Description*

Delete a variable from a shared `netWorkspace`.

### *Usage*

```
X = deleteVar(nws, varName);
```

### *Arguments*

`nws` a shared `netWorkspace` class object.

`varName` variable name of the object to delete.

### *Examples*

```
>> nws = netWorkspace('nws example');
>> store(nws, 'x', 1);
>> deleteVar(nws, 'x');
>> listVars(nws); % no variables in nws example
```

## **listVars**

### *Description*

List variables in a `netWorkspace` `nws`.

### *Usage*

```
X = listVars(nws, netWorkspaceName);
```

---

### *Arguments*

`nws` a shared `netWorkspace` class object.

`netWorkspaceName` name of `netWorkspace`.

### *Details*

`X = listVars(netWorkspaceName)` returns a struct array listing variables in the named `netWorkspace`, with the number of values, fetchers, finders and the mode for each. If no `netWorkspaceName` is given, then current `netWorkspace` name is used.

If no left hand side, print a tabular listing.

### *Examples*

```
>> nws = netWorkspace('nws example');
>> store(nws, 'x', 1);
>> listVars(nws);
```

## **currentWs**

### *Description*

Report the name of the shared `netWorkspace` used by this shared `netWorkspace` object.

### *Usage*

```
X = currentWs(nws);
```

### *Arguments*

`nws` a shared `netWorkspace` class object.

### *Examples*

```
>> nws = netWorkspace('nws example');
>> wsName = currentWs(nws);
>> display(wsName); % wsName = nws example
```

## **close**

### *Description*

Close connection to a shared `netWorkspace` `nws` to a `nws` server.

### *Usage*

```
X = close(nws);
```

---

### *Arguments*

`nws` a shared `netWorkspace` class object.

### *Examples*

```
>> nws = netWorkspace('nws example');  
>> close(nws);
```

## **server**

### *Description*

Return `nwsServer` class object that a `netWorkspace` `nws` connects to.

### *Usage*

```
s = server(nws);
```

### *Arguments*

`nws` a shared `netWorkspace` object.

### *Examples*

```
>> s = server(nws);
```

---

## nwsServer Class

### nwsServer

#### Description

`nwsServer` class constructor.

#### Usage

```
nwss = nwsServer(h, p)
```

#### Arguments

`h` server host name. Default is `'localhost'`.

`p` server port number. Default is `8765`.

#### Examples

Example 1: create a nws server with default options:

```
>> nwss = nwsServer; % create nws server with default options
```

Example 2: create a nws server on node1, port 5000:

```
>> nwss = nwsServer('node1', 5000);
```

### openWs

#### Description

Create a `netWorkspace` class object.

#### Usage

```
nws = openWs(nwss, netWorkspaceName, opts);
```

#### Arguments

`nwss` `nwsServer` class object.

`netWorkspaceName` name of the `netWorkspace` to open.

`opts` a structure of optional fields.

---

## Details

`openWs` (`nwss`, `netWorkspaceName`) creates a shared `netWorkspace` object for the `netWorkspace` named `netWorkspaceName` (which must be a string). If the shared `netWorkspace` does not exist, it will be created (upon first use) and claim the ownership of the workspace. This can be useful as an arbitration mechanism. The output of `listWss` indicates which `netWorkspace` this MATLAB process created. These will be destroyed when this MATLAB process closes the connection to the `netWorkspace` server that was used to create these spaces.

The `opts` argument has two optional fields:

- `persistent`: a boolean value indicating whether the workspace is persistent or not. If the workspace is persistent (value of 1), then the workspace is not purged when it is removed from the server `nwss`. Default value is 0.
- `space`: a `netWorkspace` class object to use open operation. If this field is not specified, then `openWs` will construct a `netWorkspace` class object on this `nws` server `nwss`.

## Examples

Example 1: open a workspace with default options:

```
>> nwss = nwsServer;  
>> nws = openWs(nwss, 'my space');
```

Example 2: open a persistent workspace:

```
>> nws2 = openWs(nwss, 'my space2', opt);
```

## useWs

### Description

Create a `netWorkspace` object, but not own it.

### Usage

```
nws = useWs(nwss, netWorkspaceName, opts);
```

### Arguments

<code>nwss</code>	<code>nwsServer</code> class object.
<code>netWorkspaceName</code>	name of the <code>netWorkspace</code> to open.
<code>opts</code>	a structure of optional fields.

---

### *Details*

`useWs(nwss, netWorkspaceName, opts)` creates a shared `netWorkspace` object for the `netWorkspace` named `netWorkspaceName` (which must be a string). If the shared `netWorkspace` does not exist, it will be created but no owner associates with it.

`opts` argument contains the following field:

- `space`: a `netWorkspace` object that represents the workspace that client wants to 'use' but not own. If this field is not specified, then `useWs` will construct a `netWorkspace` class object on server `nwss`.

### *Examples*

```
>> nwss = nwsServer;  
>> nws = useWs(nwss, 'my space');
```

## **listWss**

### *Description*

List all `netWorkSpaces` in `nwsServer` `nwss`.

### *Usage*

```
X = listWss(nwss);
```

### *Arguments*

`nwss`            `nwsServer` class object.

### *Details*

`X = listWss(nwss)` returns a struct array listing information about all the `netWorkSpaces` currently available on the server reached by connection `nwss`.

### *Examples*

```
>> nws = netWorkspace('my space');  
>> store(nws, 'a', 1);  
>> store(nws, 'b', 2);  
>> nwss = server(nws);  
>> listWss(nwss);
```

---

## mktempWs

### Description

Create a unique temporary workspace using the template `wsNameTemplate`.

### Usage

```
name = mktempWs(nwss, wsNameTemplate);
```

### Arguments

`nwss` `nwsServer` class object.

`wsNameTemplate` template for the netWorkspace name.

### Details

`name = mktempWs(nwss, wsNameTemplate)` returns the name of a temporary space created on the server reached by `nwss`. The template should contain a `%d`-like that will be replaced by a serial counter maintained by the server to generate a unique new workspace name. The user must then invoke `openWs()` with this name to create an object to access this workspace. `wsNameTemplate` defaults to `'matlab_ws_%010d'`.

### Examples

```
>> nwss = nwsServer;  
>> nws = mktempWs(nwss, 'template_%010d');
```

## socket

### Description

Return the socket connection of a `nwsServer` `nwss`.

### Usage

```
s = socket(nwss);
```

### Arguments

`nwss` `nwsServer` class object.

### Examples

```
>> nwss = nwsServer;  
>> socket(nwss);
```

---

## **deleteNws**

### *Description*

Delete the shared netWorkSpace given by wsName on a nwsServer `nwss`.

### *Usage*

```
X = deleteNws(nwss, wsName);
```

### *Arguments*

`nwss`            nwsServer class object.

`wsName`        nameof netWorkSpace.

### *Examples*

```
>> nwss = nwsServer;  
>> deleteNws(nwss, 'my space');
```

---

## Sleigh Class

### sleigh

#### Description

Sleigh class constructor.

#### Usage

```
s = sleigh(nodeNames, sOpts);
```

#### Arguments

**nodeNames** a cell array of host names.

**sOpts** an optional structure of fields, see Details.

#### Details

`s = sleigh` brings up two workers on `'localhost'` with default options for `sOpts` argument.

The `nodeNames` argument is a cell array of host names. One worker process is started for each of the host names in the cell array. The default value is `{'localhost', 'localhost'}`, which causes two worker processes to be created and executed on local machine. Note that this argument is completely ignored if you use web launch (see description below of the `launch` field in `sOpts` argument).

The `sOpts` argument contains several fields that give you ability to customize sleigh objects.

- `nwsHost`: netWorkspace server host name. Default is local machine.
- `nwsPort`: netWorkspace server port. Default port is `8765`.
- `nwsPath`: directory path of where nws package is installed on worker nodes. Default is set to where NWS MATLAB is installed on the master node. This field is irrelevant to web launch.
- `outfile`: all remote workers' standard errors will be redirected to this file. Default is `/dev/tmp`. This field is irrelevant to web launch.
- `launch`: different remote login methods to launch workers. You can choose between `web` or `ssh`. Default uses `ssh` to launch workers.
- `scriptDir`: directory where matlab worker script is stored. Default is set to the `bin` directory under the `NWS MATLAB` directory. This field is irrelevant to web launch.
- `user`: user name for remote login to execute workers. Default is the value set by environment variable `USER` on the sleigh master.

- 
- `workingDir`: define path to use as the current working directory for remote workers. Default is `/tmp`. This field is irrelevant to web launch.
  - `wsNameTemplate`: template for the sleigh workspace name. This must be a legal `format` string, containing only an integer format specifier. Default prefix is `'sleigh_ride_%010d'`.

### Examples

Example 1: start sleigh workers with default options:

```
>> s = sleigh
```

Example 2: create two sleigh workers on `node0` and `node1`, and server running on `node0`:

```
>> opt.nwsHost = 'node0'  
>> s = sleigh({'node0', 'node1'}, opt)
```

Example 3: start sleigh with web launch:

```
>> opt.launch = 'web'  
>> s = sleigh({}, opt)
```

## eachWorker

### Description

Evaluate function exactly once for each worker in sleigh `s`.

### Usage

```
x = eachWorker(s, todo, fixedArgs, execOptions);
```

### Arguments

- |                          |   |
|--------------------------|---|
| <code>s</code>           | a sleigh class object.                                    |
| <code>todo</code>        | function or expression to be evaluated by sleigh workers. |
| <code>fixedArgs</code>   | a scalar or a cell array of fixed arguments/constants.    |
| <code>execOptions</code> | an optional structure of fields.                          |

---

## Details

The only required arguments for `eachWorker` are `s` and `todo`. The rest are optional arguments. The results from `eachWorker` are normally returned as a cell array for `define` and `eval` type (see the description below of the `type` field in `execOptions` argument), and returned as a cell array of cell array for `invoke` type. However, if the `blocking` field of `execOptions` is set to `0`, then a `sleighPending` object is returned.

`fixedArgs` is a cell array of fixed arguments/constants. If the function to be executed requires only one fixed argument, then `fixedArgs` can be expressed as a vector instead of a cell array. `fixedArgs` can be accessed through a global variable named `sleighTask.argCA`, which is a cell array of values mapped from `fixedArgs`.

`execOptions` argument contains the following fields:

- `type`: determines the type of function invocation to perform. This can be `invoke`, `define`, or `eval`. Default type is `invoke`. If type is `invoke` or `define`, then the `todo` argument must be a valid `M` function name. If type is `eval`, then the `todo` argument is evaluated as it is.
- `blocking`: determines whether to wait for the results or return as soon as the tasks have been submitted. Default value is set to `1`, which means `eachWorker` is blocked until all tasks are completed. If value is set to `0`, then `eachWorker` returns immediately with a `sleighPending` object that is used to monitor the status of the tasks, and eventually used to retrieve results. You must wait for the results to be completed before submitted more tasks to the sleigh workspace.

## Examples

Example 1: execute function with no arguments:

```
>> s = sleigh;  
>> status = eachWorker(s, 'init');
```

Here is how `init.m` may look like:

```
function x = init()  
x = 1;
```

Example 2: execute `eachWorker` with `eval` execution type:

```
>> s = sleigh;  
>> opt.type='eval'  
>> eachWorker(s, '2*sleighTask.argCA{1}', 3, opt)
```

---

## eachElem

### Description

Evaluate multiple argument sets using workers in sleigh `s`.

### Usage

```
x = eachElem(s, todo, elementArgs, fixedArgs, execOptions);
```

### Arguments

`s` a sleigh class object.

`todo` function or expression to be evaluated by sleigh workers.

`elementArgs` a vector or a cell array of vectors (x, y, ... z).

`fixedArgs` a scalar or a cell array of fixed arguments/constants

`execOptions` an optional structure of fields.

### Details

`eachElem(s, todo, elementArgs, fixedArgs, execOptions)` forms argument sets from vectors passed in `elementArgs` and `fixedArgs`. `elementArgs` is a cell array of vectors (x, y, ... z), where all vectors must be of the same length. Argument set `i` consists of the `i`th element of x, y, ..., z. If `fixedArgs` is provided, then these additional arguments are appended to the argument set. The ordering of arguments can be changed by using `argPermute` vector, which defined as part of `execOptions` (see below).

The results from `eachElem` are normally returned as a cell array for `define` and `eval` type (see the description below of the `type` field in `execOptions` argument), and returned as a cell array of cell array for `invoke` type. However, if blocking field of `execOptions` is set to 0, then a `sleighPending` object is returned.

`todo` argument is a function or an expressions to be evaluated by sleigh workers. `todo` must be a function name (for `invoke` and `define` type) or expressions (for `eval` type) in quotes.

Arguments passed to the evaluated function or expression can be accessed through a global variable namd `sleighTask.argCA`, which is a cell array of values concatenate together from `elementArgs` and `fixedArgs`.

`execOptions` argument is a structure that contains the following fields:

- `type`: determines the type of function invocation to perform. This can be `invoke`, `define`, or `eval`; the default is `invoke`. If `type` is `invoke` or `define`, then the `todo` argument must be a valid M function name. If the type is `eval`, then the `todo` argument is evaluated as is.

- 
- **blocking**: determines whether to wait for the results, or to return as soon as the tasks have submitted. The default value is set to `1`, which means `eachElem` is blocked until all tasks are completed. If value is set to `0`, then `eachElem` returns immediately with a `sleighPending` object that is used to monitor the status of the tasks, and eventually used to retrieve results. You must wait for the results to be completed before submitting more tasks to the sleigh workspace.
  - **argPermute**: a vector of integers used to permute each (`elementArgs` + `fixedArgs`) argument list to match the calling convention of `todo`.
  - **loadFactor**: to restrict the number of tasks put out to the workspace. If set, no more than `loadFactor*#` of worker tasks will be posted at once. This helps to control resource demands on the nws server. It is mutually exclusive with `blocking` set to `0`.

### Examples

Here is how `addone.m` can look:

```
function result = addone(x)
result = x + 1;

>> s = sleigh;
>> result = eachElem(s, 'addone', {[1 2.5 3.1]})

result =

{1 x 1 cell}
{1 x 1 cell}
{1 x 1 cell}
```

To extract result:

```
>> result{1}{1}
result =
2

eachElem(s, 'foo', {1:3, 4:6, 7:9})
```

will generate three invocations of 'foo':

```
foo(1, 4, 7)
foo(2, 5, 8)
foo(3, 6, 9)
```

---

while

```
elementArgs = {1:3, 4:6, 7:9}
fixedArgs = {100, 200, 300}
eachElem(s, 'foo', elementArgs, fixedArgs)
```

will generate:

```
foo(1, 4, 7, 100, 200, 300)
foo(2, 5, 8, 100, 200, 300)
foo(3, 6, 9, 100, 200, 300)
```

To change the invocation type:

```
>> opt.type = 'eval'
>> eachElem(s, 'sleighTask.argCA{1}*2', 1:5, {}, opt);
ans =
[2]
[4]
[6]
[8]
[10]
```

To change the order of passing arguments:

```
>> opt.argPermute = [1 3 2 6 5 4]
>> opt.type = 'invoke'
>> eachElem(s, 'foo', elementArgs, fixedArgs)
```

will generate:

```
foo(1, 7, 4, 300, 200, 100)
foo(2, 8, 5, 300, 200, 100)
foo(3, 9, 6, 300, 200, 100)
```

---

## **nws**

### *Description*

Return the sleigh netWorkspace class object.

### *Usage*

```
nws(s);
```

### *Arguments*

**s** a sleigh class object.

### *Examples*

```
>> s = sleigh
>> nws(s)
```

## **stop**

### *Description*

Remove worker processes and deletes the sleigh workspace.

### *Usage*

```
stop(s)
```

### *Arguments*

**s** a sleigh class object.

### *Examples*

```
>> s = sleigh
>> stop(s)
```

## **check**

### *Description*

Return the number of results yet to be generated for the pending sleigh job **sp**.

### *Usage*

```
check(sp);
```

### *Arguments*

**sp** a `sleighPending` class object.

---

### *Details*

The pending sleigh job `sp` is obtained through non-blocking `eachElem` or non-blocking `eachWorker`. If the job has already completed and the results have been gathered, `0` is returned.

### *Examples*

```
>> s = sleigh
>> opt.blocking = 0
>> sp = eachElem(s, 'addone', 1:1000, {}, opt)
>> check(sp)
```

## **wait**

### *Description*

Wait and block for the results to be completed for the pending sleigh job `sp`. If the job has already completed, then `wait` returns immediately with generated results.

### *Usage*

```
wait(sp);
```

### *Arguments*

`sp` a `sleighPending` class object.

### *Details*

The pending sleigh job `sp` is usually obtained from the return value of a non-blocking `eachElem` or non-blocking `eachWorker`.

### *Examples*

```
>> s = sleigh
>> opt.blocking = 0
>> sp = eachElem(s, 'addone', 1:1000, {}, opt)
>> wait(sp)
```

## A

Appendix 19

## G

Getting Started 5

## I

Installing the Client 6

Introduction 1

## N

NetWorkSpaces Client 6

NetWorkSpaces for MATLAB Reference 19

check 39

close 26

currentWs 26

declare 24

deleteNws 32

deleteVAr 25

eachElem 36

eachWorker 34

fetch 20

fetchTry 21

find 22

findTry 22

listVars 25

listWss 30

mktempWs 31

netWorkSpace 19

netWorkSpaces Class 19

nws 39

nwsServer 28

nwsServer Class 28

openWs 28

server 27

sleigh 33

Sleigh Class 33

socket 31

stop 39

store 23

useWs 29

wait 40

NetWorkSpaces for MATLAB Tutorial 11

NetWorkSpaces Server 5

## P

Prerequisites 5

## S

Setting Up a Password-less SSH Login 8

Setting Up MATLABPATH 6

Starting Sleigh 8

Starting the NetWorkSpaces Client 7

Starting the Server 5

Stopping the Server 6

## T

Tutorials 11

## U

Using an Alternative to SSH for Windows 9





**SCIENTIFIC**  
**COMPUTING ASSOCIATES**  
INC.

One Century Tower  
265 Church Street, New Haven, CT 06510-7010  
Tel: (203) 777-7442 • Fax (203) 776-4074

[www.LindaSpaces.com](http://www.LindaSpaces.com)