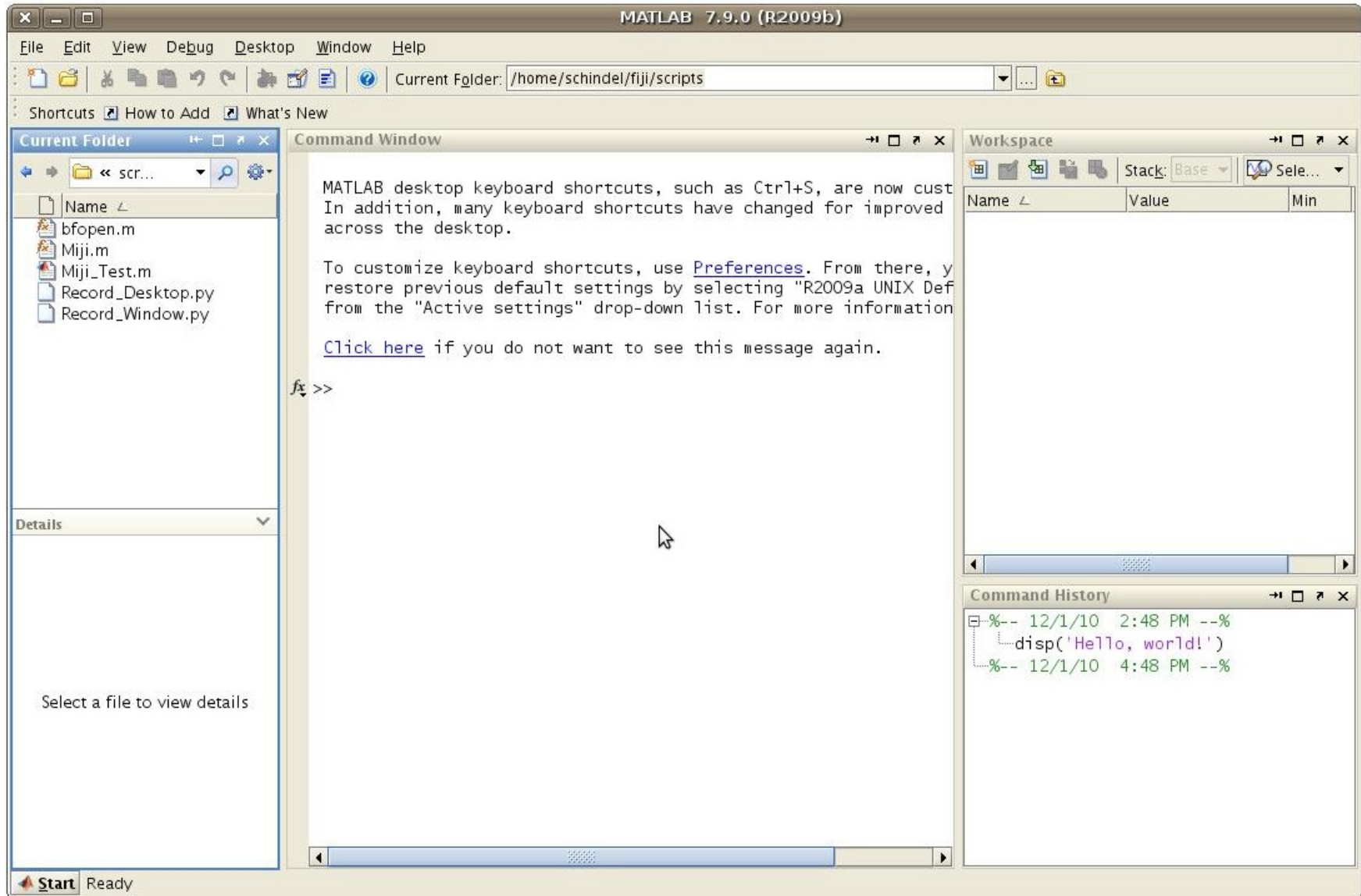# The workspace

Calling functions

A function is something you ask Matlab to execute. It has a name and a comma-separated parameter list.

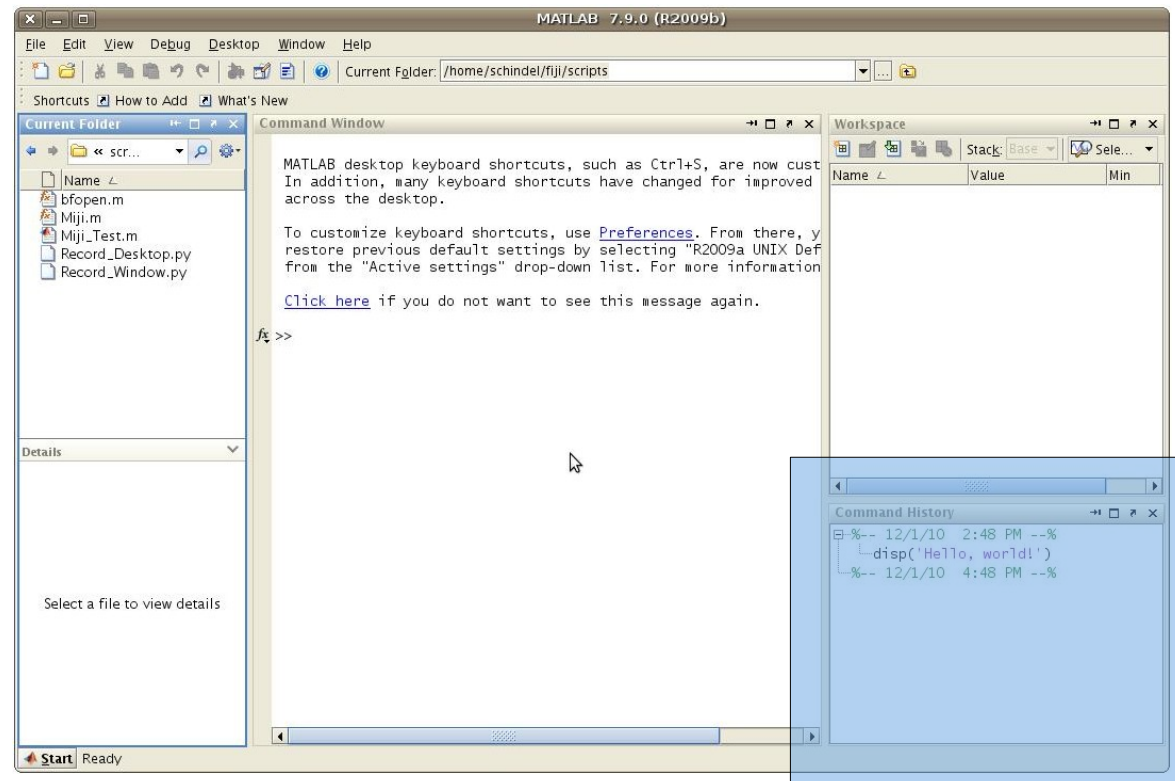**Example:** `disp('Hello, World!')`

A function can return a value.

Example: `rand()`

Note: to suppress output, you can add a semicolon (";")

The command history

Matlab records all commands in the history:

Double-click on an item to repeat it; alternatively, you can use Cursor Up/Down to go back/forth.

Tab completion

If you hit the *Tab* key (it is on the left of your keyboard, just above the Caps Lock key) after typing part of a function, Matlab will *complete* it, or show you alternatives if there are multiple completions.

Another convenient way to explore available functions is to click on the little *fx* symbol on the left border of the command window.
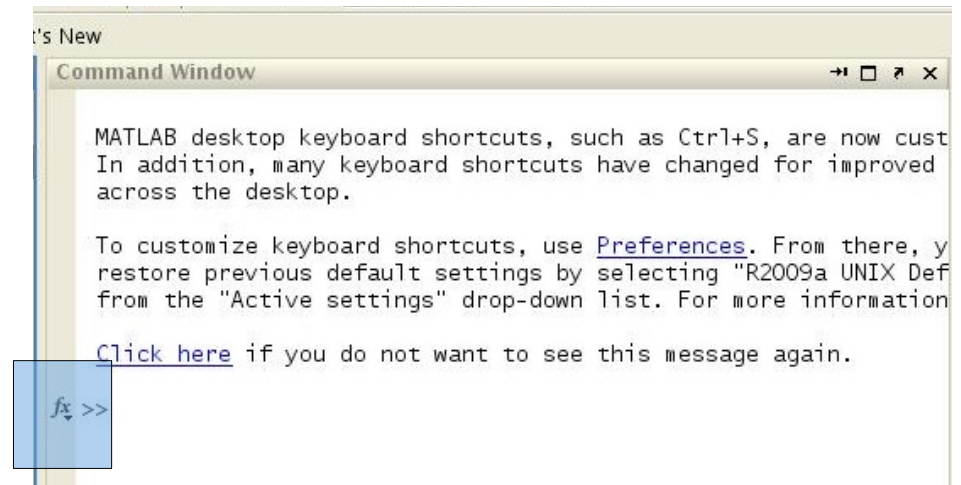
Variables

All values (numbers, text, matrices, etc) can be stored in *variables*. A variable has a name.

Assigning a value to a variable: `height = 10`

Using a variable: `disp(a)`

Non-trivial assignment: `second = first`

Note: the name on the left side refers to the variable *itself*, while the name on the right side is interpreted as the variable's *current value*.

Variables *(continued)*

If you call a function returning a value without assigning it to a variable, the value is stored in the special variable `ans`:

Example: `cos(pi)`

Variable names are case-sensitive.

Example: `PI`

Variables *(continued)*

Variable names consist of a letter or underscore, followed by an arbitrary number of letters, underscores or numbers.
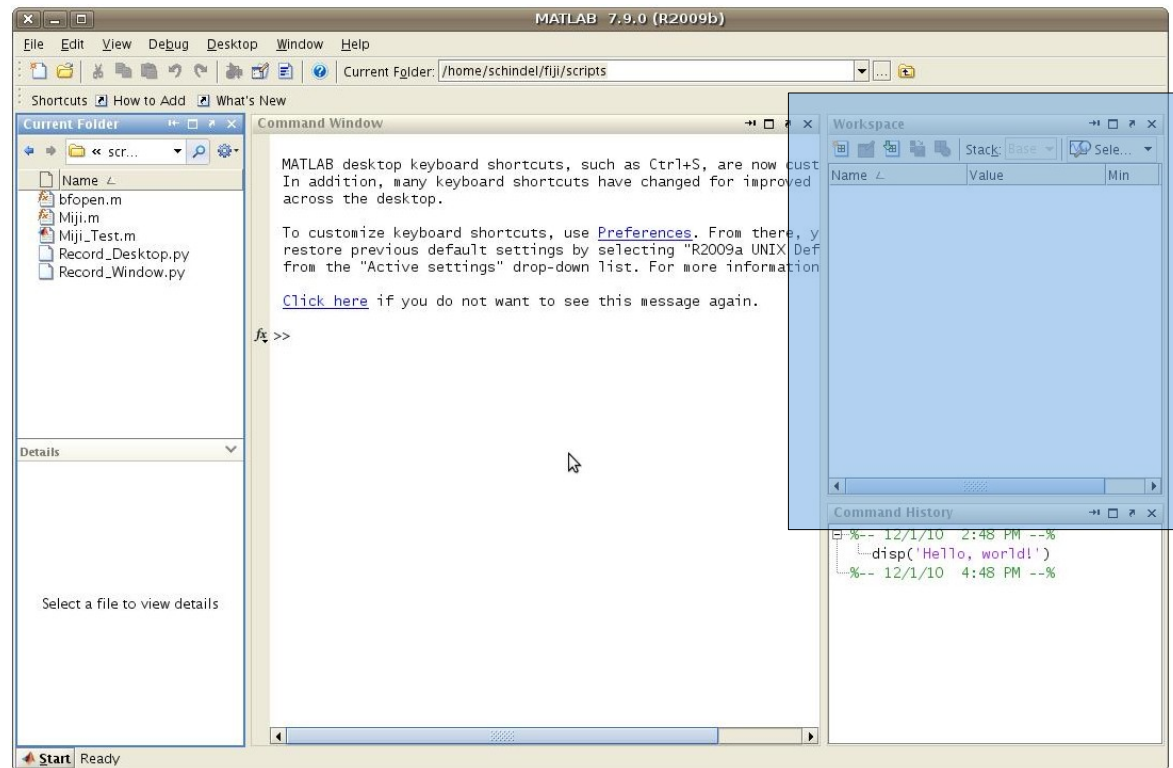
Example: `a; hello123`

Invalid: `1first`

Note: Make sure you choose sensible names! Six months from now, you do not remember what "`c`" was supposed to mean...

Variables *(continued)*

All variables are listed in the *workspace*:

Double-click in the workspace to edit the variable.

Operators

Arithmetic expressions consist of operators such as `+`, `-`, `*`, `/`.

Example: `(2^3 - 5) * 7 + tanh(1)`

Operators always operate on two entities, one to the left of the operator, and one to the right.

Types

Numbers[*], text (so-called *strings*), vectors and matrices are *types*.

Example: `[1 0 1]` is a row vector

Example: `[1 0 0; 0 1 0; 0 0 1]` is the 3x3 identity matrix

Example: `'Hello, MPI'` is a string

---

[*] Numbers can be 8-bit, 16-bit, 32-bit signed/unsigned integers, or 32-bit, 64-bit floating point numbers.

Vectors and Matrices

Originally, Matlab ("Matrix laboratory") was designed as an easy user interface to a large library of linear algebra functions implemented in Fortran.

To this date, vector and matrix operations are the strongest points of Matlab.

For example, typing a single number is equivalent to typing a 1x1 matrix.

Vector and Matrix operators

Matrix multiplication: `2 * [3 -1]`

Transposed matrix: `[5 4]'`

Dot product: `[1 2] * [3 1]'`
    also: `dot([1 2], [3 1])`

Point-wise multiplication: `[1 2] .* [3 1]`

Convoluted dot product:
    `sum([1 2] .* [3 1])`

Accessing Vectors and Matrices

Single value: `matrix(row, column)`

Submatrix: `matrix(2:3, 1:3)`

Column vector: `matrix(row, :)`

Matrix as vector: `matrix(:)`

Identity matrix: `eye(3)`

Linear and logical indexing

You can access a matrix as if it was a single column vector: `matrix(2:5)`

Make a logical matrix: `matrix > 1`

The logical matrix consists of boolean entries, i.e. `true` or `false`. Boolean values can be used as if they were `1 (= true)` or `0 (= false)`.

Logical indexing (extract certain elements from a vector (or matrix): `matrix(matrix > 1)`

Cell arrays

Cell arrays are lists that can contain non-numerical values.

Example:

```
cell{1} = 'Hello'; cell{2} = 2;
```

Structs

Structs are like cell arrays, but the items are references by a name rather than an index.

Example:
```
bag.label = 'Hello'; bag.x = 2;
```

Note: If you know object-oriented script languages like Python or Perl, you will recognize this "*bag of things*" approach of representing objects.

Types of variables

Variables have implicit types: when assigning a value to a variable, the type is inferred.

Example: `bag = 1; bag.name = 'Hello';`

You cannot reuse a numerical array as a cell array directly.

Example: `bag = 1; bag{1} = 2;`

Loading and Saving data

To remember the variables in your workspace, just save them into a file:

```
save('/path/to/file.mat')
```

To save just a part, append the names of the variable(s) to the parameter list.

```
save('/path/to/file.mat', 'bag')
```

You can load the variables from the file later:

```
load('/path/to/file.mat')
```

Plots

Given a list of values or coordinates, it is very easy to plot the values:

List of values: `plot(values)`

List of coordinates: `plot(x, y)`

Double log plot of a list of coordinates:
`loglog(x, y)`

Help!

If you forgot how to call a certain function (e.g. what parameters it takes), ask for help:

```
help save
```

Or ask for more verbose help in an extra window:

```
doc save
```

You can search the help for a keyword, too:

```
docsearch save
```

Scripts

For recurring tasks, you can save a sequence of commands into a file (file extension: *.m*).
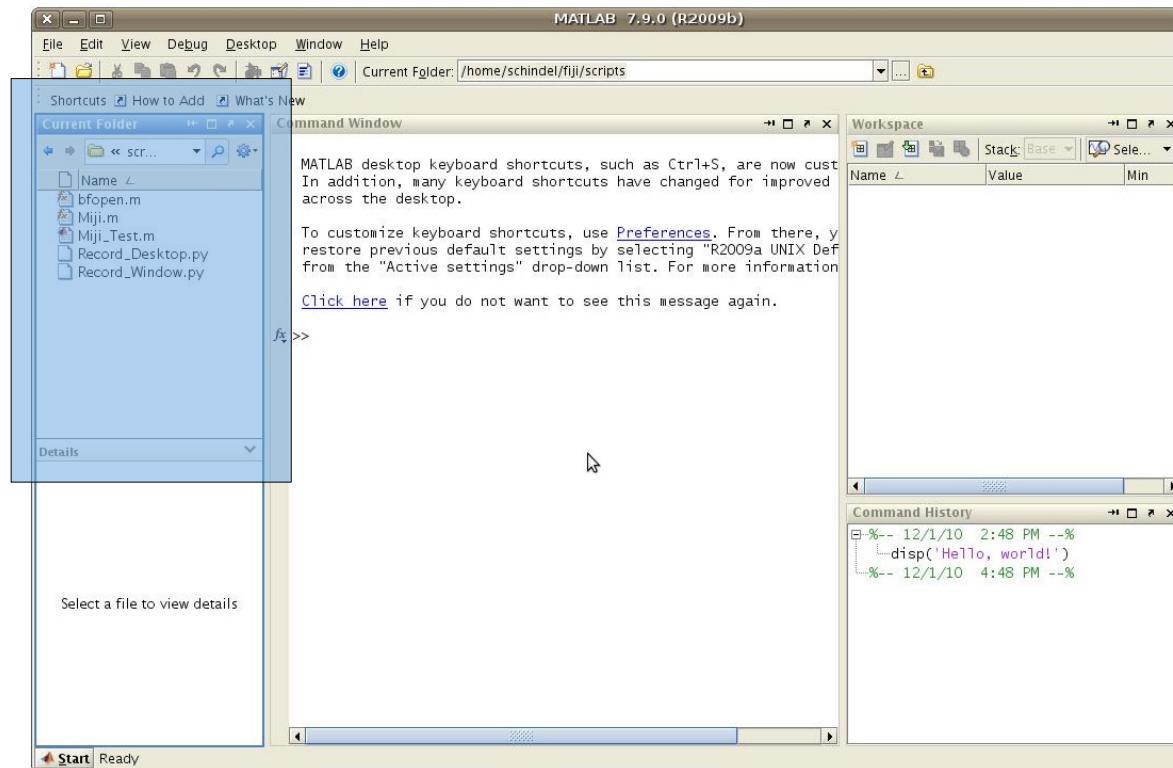
Example: `edit hello.m`

This script can be called by its name: `hello`

Note: on Linux, the keyboard bindings of Matlab are imitating Emacs (Ctrl+X Ctrl+S instead of Ctrl+S for saving).

Note[2]: you can edit the file with any editor you like!

Paths

Matlab looks for scripts in the current folder:

There is also a search path: File>Set Path...

User functions

Instead of scripts, you can write functions into the file. Example:

```
function [mean,stddev] = stat(x)
n = length(x);
mean = sum(x)/n;
stddev = sqrt(sum((x-mean).^2)/n);
```

Note: the file and function name must match.

Note[2]: A function can return a list; the specified variables must be assigned a value in the function. Use a statement like `[mean s] = stat(x)` to assign the return values to variables.

Scripts vs functions

There are a few differences between scripts and user-specified functions.

Most notably, functions can take and return values.

And while the variables used in scripts are living in the workspace, the variables used in the functions do not.

Natural progression:
   interactive fooling around $\rightarrow$ script* $\rightarrow$ function

* You can select part of the command history and *Create Script* in the context menu

Comments

When programming a cool function, it is often necessary to add information for yourself, so you do not get confused by your own code 6 months later!

Example:

```
values = rand(10)
% Cut off at 0.5 (disallow smaller)
values = max(0.5, values)
```

Note: often, code is *commented out*, i.e. disabled

Conditionals

Sometimes, functions need to behave differently in certain conditions.

Example:

```
if x < 5
  disp('x is too small!')
end
```

Note: the part between the condition and the *end* keyword is called *code block* and should be indented (you'll thank yourself 6 months from now).

Conditionals (continued)

Conditionals can have alternative clauses

Example:

```
if x < 5
  disp('x is too small!')
elseif x > 10
  disp('x is too large!')
else
  disp('x is just perfect!')
end
```

Loops

To execute a code block multiple times, make a *for* or *while* loop:

```
sum = 0
for k = 1:10
   sum = sum + k
end


sum2 = 1
while sum2 < 100
   sum2 = sum2 * 2
end
```

Note: `i` denotes the complex number (`i^2 = -1`), and likewise `j`, so using those letters as variables works, but is quite slow.

Java

Java libraries can be used easily inside Matlab:

```
% Tell Matlab where the library is
javaaddpath /Users/me/library.jar
% Tell Matlab about the class
import my.Clazz
% create an instance of the class
instance = Clazz();
```

Tip: you can use all of Fiji by adding Fiji's *scripts/* directory to the search path and calling

```
Miji(false);
```

Input

Often, it is useful to have a function ask for interactive input.

Example:

```
count = input('How often? ');
disp(strcat(['You said ',...
  num2str(count),...
  ' times?']));
```

Anonymous functions

In some cases, you might want to assign a code block rather than a value to a variable.

Example:

```
myfunc = @(x)(x + 2);
myfunc(1)
myfunc = @(x)(x ^ 3);
myfunc(1)
```

This is important e.g. when using the optimization toolbox, to specify the target function.

Debugging

The most tedious part of developing code is always the debugging. (Computers do what we tell them to do instead of what we want them to do.)
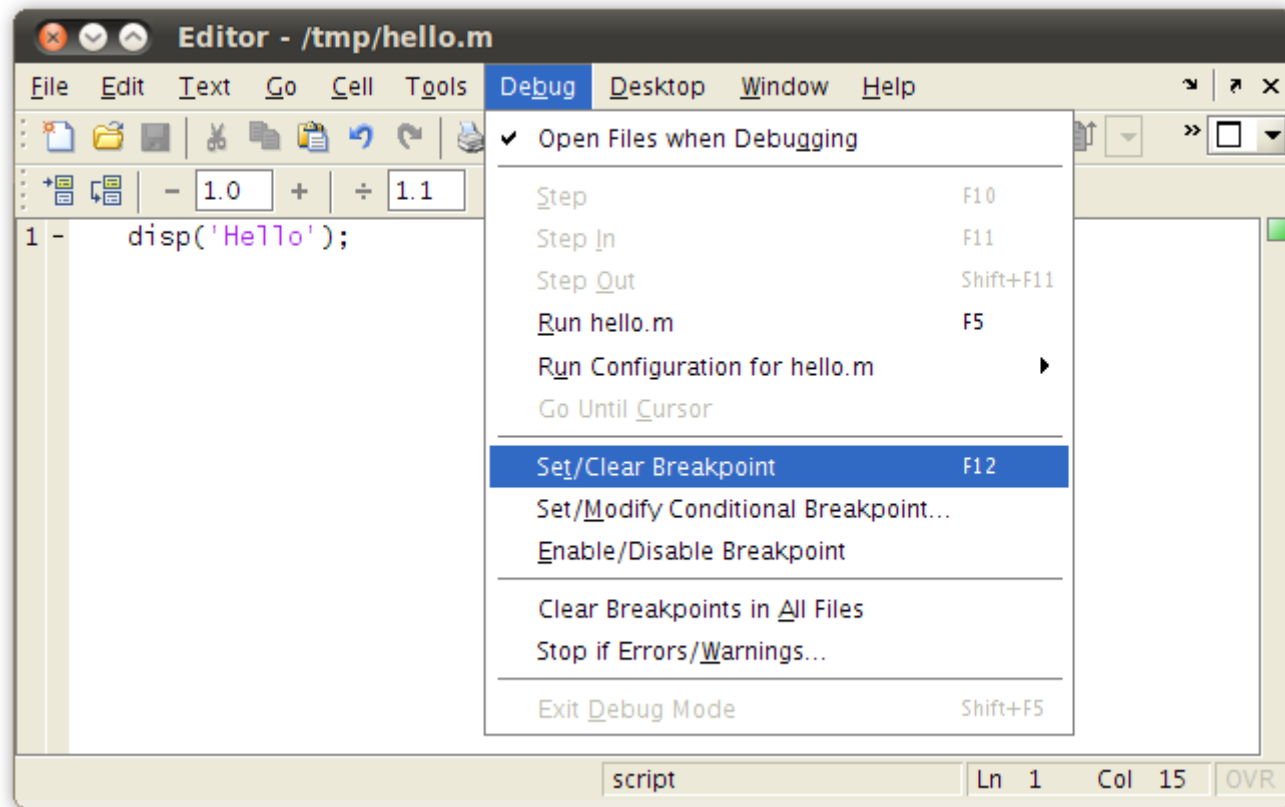
The most important tool to find out what is happening is to simply remove the semicolons from certain lines, to see the intermediate results.

Another tool is to use the `disp()` function to output so-called *debug messages*.

Debugging (continued)

Alternatively, you can use the builtin debugger.

Simple profiling

If your code runs slow, you need to find out what parts are responsible for the slowness. This is called *code profiling*.

The easiest form of profiling is provided by the `tic` and `toc` functions:

```
tic
for k = 1:10000
  disp(k);
end
toc
```

Further reading

The Mathworks has a metric ton of documentation:

http://www.mathworks.com

We also try to have useful information on the Wiki:

http://wiki/wiki/imagepro/index.php/MATLAB

The best source for additional Matlab code:

http://www.mathworks.com/matlabcentral/