

Brain Imaging Workshop - Spring 2001  
SPM Batch Processing  
Matt Budde

Outline:

- 1 SPM & Matlab
  - I Matlab Variables
  - II Cells
  - III Structures
- 2 SPM Batch Intro
  - I Matlab Path
  - II Batch Organization
  - III Analyses file
  - IV Variable Description files
  - V Running Batch
  - VI Debugging
- 3 Example Batch Processing
  - I Preprocessing
  - II Model specification
  - III Estimation and Contrasts
- 4 Summary

This presentation will introduce batch mode for SPM99 (<http://www.fil.ion.ucl.ac.uk/spm>) directed at those with little or no knowledge of batch mode but assumes a fairly good knowledge of SPM. (Batch mode is only usable in SPM fMRI, and not PET analysis) Although it may be repetitive for some, an intro on Matlab will provide the background necessary for effective use of batch processing capabilities.

## 1 SPM & Matlab

Matlab is a powerful mathematical program that is reasonably user friendly and easy to learn. The syntax of Matlab is much easier than other languages (C, C++, Java) and very well suited for the 'number crunching' of neuroimaging. SPM is written using Matlab's programming tools and is made up of (mainly) text files written in Matlab syntax called function files (.m). As the name indicates, each of these files performs a specific function, and together, these are known as the SPM package.

As well as being written in Matlab, SPM runs on what is known as the Matlab workspace (this is what you see when Matlab starts). In this workspace, variables can be stored for use in subsequent steps. One nice feature about Matlab is that no variable types need to be declared as

in other languages. The type is determined by the type of data being stored in the variable (char, int, double, etc...).

### Matlab Variables

For the rest of the guide, any input or output in Matlab is indicated by the Courier font.

Help in Matlab can be found by typing `help` or `help topic` (i.e. `help struct`).

The Matlab command prompt is `>>`.

Matlab uses matrix notation for many of its variables. Matrices are stored and created using square brackets `[]`. The size of a matrix is always `#rows x #columns`. A whitespace or comma separates elements into columns and a semicolon separates each row. Thus a 2 x 3 matrix of numbers could look like this:

```
>>numbers = [1 2 3;4 5 6] OR >>numbers = [1,2,3;4,5,6]
```

and produces:

```
numbers =  
    1    2    3  
    4    5    6
```

If three dots are placed at the end of the line, Matlab will treat the lines as a single command. It helps keep things organized in large variable assignments. Thus, this example is identical to those above:

```
>>numbers = [1,2,3;...  
4,5,6]
```

A semicolon at the end of the line executes a command as it would without it, but the output is not echoed. Thus:

```
>>numbers = [1 2 3;4 5 6]
```

```
numbers =  
    1    2    3  
    4    5    6
```

```
>>
```

but:

```
>>numbers = [1 2 3;4 5 6];  
>>
```

Once a variable is defined, it can be substituted into other variables provided the type and dimensions agree:

```
>>new = [numbers; 7 8 9]
new =

     1     2     3
     4     5     6
     7     8     9
```

For simple variables, each matrix can only contain a single data type. Batch mode typically uses numbers in a horizontal matrix, like this:

```
>>integers = [10 20 30 40]
```

An element of the matrix can be referenced by:

```
>>integers(3)

ans =

     30
```

or:

```
new(2,2)

ans =

     5
```

for 2D matrices.

Strings are also considered matrices, but each character is an element, not each word, and the brackets are not necessary:

```
>>story = 'one fish'
story =

one fish
```

where:

```
>>story(3)

ans =

     e
```

## Cells

A cell is a variable type that is much more flexible than a matrix. The advantage of cells is that they allow other variable types to be stored within them using curly braces {}, like this:

```
>>integers = {[1 2 3], [4 5 6], [7 8 9]}
```

producing:

```
integers =  
    [1x3 double]    [1x3 double]    [1x3 double]
```

Cells are referenced one of two ways, and each way provides a different output. Using parenthesis () will return a cell element:

```
>>integers(2)  
  
ans =  
    [1x3 double]
```

and curly braces {} will return the actual element of the cell in its original format:

```
>>integers{2}  
  
ans =  
     4     5     6
```

and these can be used in combination:

```
>>integers{2}(2)  
  
ans =  
     5
```

Cells can also contain strings, and are slightly different than matrices stored in cells:

```
>>strings = {'red' 'green' 'blue'}
```

produces:

```
strings =  
    'red'    'green'    'blue'
```

Referencing a cell of strings is similar to a cell of matrices where parenthesis return a cell element:

```
>> strings(2)
ans =
    'green'
```

and curly braces return the string as strings are typically stored, as a matrix:

```
>>strings{3}
ans =
    blue
```

Cells are used frequently in batch processing because of their flexibility. They are used as variables within a different data type, structures.

### Structures

A structure is a very powerful variable type that stores a number of variables of different types (similar to an object in Object-Oriented programming). SPM commonly uses structures to store and pass variables between functions. Likewise, the structure is the basis of batch mode. A structure is defined by providing a field name and its corresponding value, like this:

```
>>garage = struct('owner','me','type','BMW','mpg',30.5,'topspeed',130)
```

which is identical to:

```
>>garage = struct(...
'owner',    'me',...
'type',     'BMW',...
'mpg',     30.5,...
'topspeed', 1...
);
```

and produces the following output:

```
garage =
    owner: 'me'
    type: 'BMW'
    mpg: 30.5000
    topspeed: 130
```

Referencing an element value is accomplished by indexing the variable:

```
>>garage.type
ans =
    BMW
```

Changing one value in the structure is done by referencing that field as well:

```
>>garage.mpg = 29.4

garage =

    owner: 'me'
    type: 'BMW'
    mpg: 29.4000
    topspeed: 130
```

In addition, structures with identical fields can be added as an array.

```
>>garage(2) = struct('owner','you','type','Harley','mpg',45.6,'topspeed',110)

garage =
1x2 struct array with fields:
    owner
    type
    mpg
    topspeed

>>garage(2)
ans =

    owner: 'you'
    type: 'Harley'
    mpg: 45.6000
    topspeed: 110
```

Additional structures can be copied by referencing a previous structure:

```
>>garage(3) = garage(2)
```

and changing the variables in `garage(3)` as needed.

```
>>garage(3).type = 'El Camino';
>>garage(3).mpg = 5;
```

This feature is especially useful in simplifying batch processing.

## 2 SPM Batch Intro

This section does not go into the theoretical and methodological issues of SPM, but it does discuss the underlying organization from a programming standpoint in order to provide a better understanding of batch mode.

As mentioned earlier, SPM is programmed into a number of function files. When SPM is initially started through Matlab, the function file to start the program is called (appropriately named `spm.m`). Once SPM is started, the rest of the processing is generally done with GUI

(graphical user interface) input. When a button is pressed, SPM is essentially calling a different function file. For example, if the coregistration button is pressed, the GUI calls the `spm_coreg_ui.m` function file, which is identical to typing `spm_coreg_ui` at the Matlab command line. Batch mode simply bypasses the entering of inputs, but performs the same functions that the GUI does. In this way, batch mode defines all of the necessary inputs before the actual processing is executed. Because the inputs are stored as text files, the inputs can easily be saved or changed for later use. Batch processing is also very flexible. It can be set up to run different steps, different subjects, different studies, or any combination of the three. It is possible to set up batch mode to process everything for a single study. However, because it is so flexible, it requires a good understanding to take full advantage of its capabilities.

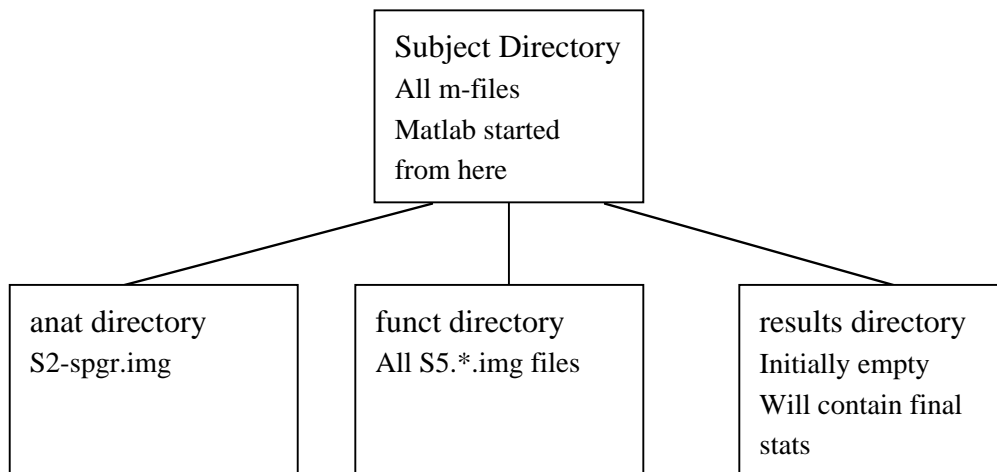
### Matlab Path

The Matlab path is important to understand. As with Unix/Linux, Matlab has a path where it searches for functions as they are called, from the beginning of the path to the end. Thus, `>>which functionname` will show the first instance of the file that occurs on the path. It is important to know which m-file is being called. Similarly, changes to the SPM code can be used in this way. If a directory that is listed before the SPM99 directory on the Matlab path contains a file named the same as a SPM file, the first file will be called.

### Batch Organization

Batch mode uses two types of files to run the processing. There is one top level file (analyses) and one or more variable description files. (examples will be shown later) These are text files written in Matlab syntax that define the necessary variables. Batch mode uses two function files (`spm_bch.m` and `spm_bch_bchmat.m`) to evaluate the files and run all the other steps of processing.

Initial file structure for examples:



## Analyses file

The analyses file is the top level batch file that controls which types of processing are done, in which directories, and by reading which m-files for variables. The working directory is relative to the pwd of Matlab, and the m-file is relative to the work\_dir. For a single process (realign), the file looks like this:

```
%-----  
analyses = struct(...  
    'type',          [2], ...  
    'work_dir',     [1], ...  
    'mfile',        [1], ...  
    'index',        [1] ...  
);  
  
%-----  
type = {'slicetime','realign','coreg','normalize','smooth',...  
        'model','contrasts','defaults_edit','headers','means'};  
  
%-----  
work_dir = { ...  
            './funct',...  
            };  
  
%-----  
mfile = { ...  
         './realign.m',...  
         };  
  
%-----
```

To complete all of the preprocessing steps for a single subject, the analyses description looks like this:

```
%-----  
analyses = struct(...  
    'type',          [ 2 3 4 4 5], ...  
    'work_dir',     [ 1 2 2 1 1], ...  
    'mfile',        [ 1 2 3 4 5], ...  
    'index',        [ 1 1 1 1 1] ...  
);  
  
%-----  
type = {'slicetime','realign','coreg','normalize','smooth',...  
        'model','contrasts','defaults_edit','headers','means'};  
  
%-----  
work_dir = { ...  
            './funct',...  
            './anat',...  
            './results',...  
            };  
  
%-----  
mfile = { ...  
         './realign.m',...  
         './coreg.m',...  
         './normanat.m',...  
         './normfunct.m',...  
         './smooth.m',...  
         './estimatel.m',...  
         './contrastsl.m',...  
         };  
  
%-----
```

The mfile refers to the variable description files that are specific to each type of processing. The index variable refers to which structure declared in the variable description file is to be used. Thus, if both `realign(1)` and `realign(2)` were defined in `realign.m`, an index of 2 would use the variables defined in `realign(2)`.

### Variable Description files

Each variable description file contains the structure that is specific to that type of process. The variables are identical to what would be entered if the processing were done through the GUI. The best method to create these files is to go through the GUI once for each step, and write down the inputs. Then, using the batch manual (`spm_bch.man`, found with the SPM distribution), create a file that is specific to each of the steps, or modify an existing file. In most cases, only the variables used in the GUI inputs need to be defined in the mfile. The suggested way of creating the file is to use an existing file that has worked and modify the necessary inputs. This ensures that the appropriate variables are defined and that they are in the appropriate data type.

Example variable description file for smooth:

```
%-----
% user variables defined here
%-----

smfiles = spm_get('files', './funct', 'nrS5.*.img');

%-----
% batch variables defined here for analysis 'smooth'
%-----

smooth=struct( ...
    'FWHMmm', [8], ...
    'files', {smfiles} ...
);
```

Since the file is written in Matlab syntax, any variable substitutions or other Matlab and SPM functions can also be used. A few useful examples are obtaining file names using `spm_get`, as shown above (see `>>help spm_get`), or declaring variables at the beginning of the file and substituting the variable name later, as in this contrasts example:

```
%-----
% user variables defined here
%-----

ronly = [1 0 0];
lonly = [0 1 0];

%-----
% batch variables defined here for analysis 'contrasts'
%-----
```

```

contrasts(1) = struct( ...
    'names',    {'Ronly', 'Lonly'}, ...
    'types',    {'T', 'T'}, ...
    'values',   {'ronly', 'lonly'} ...
);

```

An outside file of variable assignments can be used as well. If the file `soa.m` contains the lines:

```

ronly = [1 0 0];
lonly = [0 1 0];

```

putting:

```
run soa
```

in the user defined section will allow substitution as in the previous example.

### Running Batch

Once all of the files are specified correctly, `>>spm_bch('analysesfile.m')` will execute batch mode. First, the two (or more) mfiles are written in binary format (.mat). These are loaded into the workspace when a function is called. If there are any errors in syntax or batch execution, batch mode will stop and give some type of error message. Since batch mode tends to give very generalized error messages, several things can be done to identify the problem.

### Debugging

Since batch mode is affectionately known as SPM 'crash' mode, problems are inevitable both to the novice and advanced user. Batch mode only yields several error messages, so problems can be difficult to find. In addition, batch mode doesn't catch the errors that would arise during processing. However, there are several different things that can be done to pinpoint the problem.

```
>>which filename
```

Ensures that the correct analyses and variable description files are being used.

```
>>run filename
```

```
>>who
```

Run executes the file and helps in finding syntax errors like missing commas or quotation marks. Also, it will send an error message if a variable substitution occurs without having the variable specified elsewhere. The error messages are detailed enough to know what the problem is and where it occurs. Once the file runs without errors, 'who' will show the variables that exist in the workspace. By typing the variable name, one can see if the variable contains the values that were intended. One example of this is where `spm_get` is used. If no files are found, the variable will contain an empty array and cause an error only during processing, whereas if it was specified correctly, the variable should contain a list of filenames.

### **3 Example Batch Processing**

The example data set is a simple block-design motor task. The subject was instructed to alternate hand tapping for a period of 20 seconds with 20 seconds of rest in between. Thus, in intervals of 20 seconds, the task went:

Rest - right - left - rest - right - left - rest - right - left - rest - right - left - rest

For a total scan time of 2:60. The TR was 2 sec, which resulted in 130 EPI functional scans (first 4 were removed), and an accompanying 3D SPGR anatomical volume. In this example, batch mode will run everything from pre-processing through contrasts. Three batch steps were executed: preprocessing, specifying model, and stats. Thus, three analyses files were used: runallpre.m, runmodel.m, and runallstats.m, respectively. A variable description file for each step was created. The functional images were first realigned. The anatomical image was then coregistered to the mean produced by realignment. Then anatomical image was normalized to the T1 template and the parameters were applied to the functional images as well. The functional images were then smoothed. All of this was executed in a single batch step. The model was specified in another step and created in a single batch execution. Finally, the model was estimated and contrasts were done in the third and final step. Slice timing was not done because the images were acquired coronally, which complicates that step, and it is less important for block designs.

## Preprocessing

---

**runallpre.m** (analyses file for realignment, coregistration, normalization, and smoothing) (each step defined in it own mfile)

---

```
%-----
analyses = struct(...
    'type',          [2 3 4 4 5], ...
    'work_dir',     [1 2 2 1 1], ...
    'mfile',        [1 2 3 4 5], ...
    'index',        [1 1 1 1 1] ...
);
%-----
type = {'slicetime','realign','coreg','normalize','smooth',...
        'model','contrasts','defaults_edit','headers','means'};
%-----
work_dir = { ...
    './funct',...
    './anat',...
    './results',...
    };
%-----
mfile = { ...
    './realign.m', ...
    './coreg.m', ...
    './normanat.m', ...
    './normfunct.m', ...
    './smooth.m', ...
    };
%-----
```

---

**realign.m (variable definition file for realignment)**

---

```
%-----  
% user variables defined here for analysis 'realign'  
%-----  
  
refiles = spm_get('files', './funct', 'S5*.img');  
  
%-----  
% batch variables defined here for analysis 'realign'  
%-----  
realign = struct( ...  
    'subject_nb',    1, ...  
    'num_sessions',  1, ...  
    'sessions',      [1], ...  
    'option',        3, ...  
    'modality',      1, ...  
    'reslice_method', -9, ...  
    'create',        3, ...  
    'mask',          0, ...  
    'adjust_sampling_errors', 0 ...  
);  
%-----  
sessions= struct( ...  
    'images', {{refiles}}...  
);
```

---

**coreg.m (variable definition file for coregistration)**

---

```
%-----  
% user variables defined here for analysis 'coreg'  
%-----  
  
anat = spm_get('files', './anat', 'S2-spgr.img');  
funct = spm_get('files', './funct', 'meanS5.001.img');  
  
%-----  
% batch variables defined here for analysis 'coreg'  
%-----  
coreg = struct( ...  
    'subject_nb',    1, ...  
    'opt',           1, ...  
    'target_mod',    5, ...  
    'object_mod',    2, ...  
    'target_image',  [funct], ...  
    'object_image',  [anat], ...  
    'other_image',   [] ...  
);  
%-----
```

---

**normanat.m (variable definition file for anatomical image normalization)**

---

```
%-----  
% user variables defined here for analysis 'normalize'  
%-----  
  
anatom = spm_get('files', './anat', 'S2-spgr.img');  
tltemplate = spm_get('files', '.', 'T1.img');  
  
%-----  
% batch variables defined here for analysis 'normalize'  
%-----  
normalize = struct( ...  
    'option',      3,...  
    'nsubjects',  1, ...  
    'object_masking', 0, ...  
    'template',    {{tltemplate}}, ...  
    'type',        0, ...  
    'mask_brain',  0, ...  
    'interp',      -9, ...  
    'image',       {{anatom}}, ...  
    'objmask',     [''], ...  
    'matname',     [''], ...  
    'images',      {{anatom}} ...  
);
```

---

**normfunct.m (variable definition file for functional image normalization)**

---

```
%-----  
% user variables defined here for analysis 'normalize'  
%-----  
  
nfunct = spm_get('files', './funct', 'rS5*.img');  
params = spm_get('files', './anat', 'S2-spgr_sn3d.mat');  
  
%-----  
% batch variables defined here for analysis 'normalize'  
%-----  
normalize(2) = struct( ...  
    'option',      2,...  
    'nsubjects',  1, ...  
    'object_masking', 0, ...  
    'template',    {{}}, ...  
    'type',        0, ...  
    'mask_brain',  0, ...  
    'interp',      -9, ...  
    'image',       {{}}, ...  
    'objmask',     [''], ...  
    'matname',     {{params}}, ...  
    'images',      {{nfunct}} ...  
);  
%-----
```

---

**smooth.m (variable definition file for smoothing)**

---

```
%-----  
% user variables defined here for analysis 'smooth'  
%-----  
  
smfiles = spm_get('files', './funct', 'nrS5.*.img');  
  
%-----  
% batch variables defined here for analysis 'smooth'  
%-----  
  
smooth=struct( ...  
    'FWHMmm', [8], ...  
    'files', {smfiles} ...  
);  
%-----
```

---

**Model specification**

---

---

**runmodel.m (analyses file creation of fMRI model)**

---

```
%-----  
analyses = struct(...  
    'type', [6], ...  
    'work_dir', [1], ...  
    'mfile', [1], ...  
    'index', [1] ...  
);  
%-----  
type = {'slicetime','realign','coreg','normalize','smooth',...  
        'model','contrasts','defaults_edit','headers','means'};  
%-----  
work_dir = { ...  
    './results',...  
};  
%-----  
mfile = { ...  
    './modell.m'...  
};  
%-----
```

---

**modell1.m (variable definition file for creation of model)**

---

```
%-----  
% user variables defined here for analysis 'model'  
%-----  
  
rblock = [10 40 70 100];  
lblock = [20 50 80 110];  
  
%-----  
% batch variables defined here for analysis 'model'  
%-----  
model(1) = struct( ...  
    'types',          1, ...  
    'stop_writing',  1, ...  
    'RT',            2, ...  
    'replicated',    0, ...  
    'same_time_param', 0, ...  
    'nsess',         1, ...  
    'nscans',        [130], ...  
    'conditions_nb', [2], ...  
    'conditions',    [1], ...  
    'regressors_nb', [0], ...  
    'regressors',    [], ...  
    'parametrics_type', {'none'}, ...  
    'parametrics',   [], ...  
    'stochastics_flag', [0], ...  
    'stochastics',   [] ...  
);  
  
%-----  
conditions(1) = struct( ...  
    'names',          {'Rhand','Lhand'}, ...  
    'onsets',         {rblock, lblock}, ...  
    'types',          {'epochs','epochs'}, ...  
    'durations',      [], ...  
    'bf_ev',          [0 0], ...  
    'bf_ep',          [1 1], ...  
    'volterra',       0, ...  
    'variable_dur',  0 ...  
);  
  
%-----  
bf_ev(1) = struct( ...  
    'ev_type', 1, ...  
    'win_len', [], ...  
);  
  
bf_ep(1) = struct( ...  
    'ep_type', 4, ...  
    'length', 10, ...  
    'conv',    1, ...  
    'deriv',   0 ...  
);  
%-----
```

## Estimation and Contrasts

---

**runallstats.m** (analyses file for estimation and contrasts)

---

```
%-----
analyses = struct(...
    'type',          [6 7], ...
    'work_dir',     [3 3], ...
    'mfile',        [1 2], ...
    'index',        [1 1] ...
);
%-----
type = {'slicetime','realign','coreg','normalize','smooth',...
        'model','contrasts','defaults_edit','headers','means'};
%-----
work_dir = { ...
    './funct',...
    './anat',...
    './results',...
    };
%-----
mfile = { ...
    './estimate1.m',...
    './contrastsl.m',...
    };
%-----
```

---

**estimatel.m (variable definition file for estimation)**

---

```
%-----  
% user variables defined here for analysis 'model'  
%-----  
  
estimgs = spm_get('files', './funct', 'snrS5.*.img');  
  
%-----  
% batch variables defined here for analysis 'model'  
%-----  
  
model(1) = struct( ...  
    'types',          3, ...  
    'files',          {{estimgs}}, ...  
    'global_effects', {'Scaling'}, ...  
    'burst_mode',    0, ...  
    'HF_fil',         'none', ...  
    'HF_cut',         [0], ...  
    'LF_fil',         'hrf', ...  
    'LF_cut',         16, ...  
    'int_corr',       'none', ...  
    'now_later',      1, ...  
    'stop_writing',   1, ...  
    'trial_fcon',     0, ...  
);
```

---

**contrastsl.m (variable definition file for contrasts)**

---

```
%-----  
% user variables defined here for analysis 'contrasts'  
%-----  
  
ronly = [1 0 0];  
lonly = [0 1 0];  
roverl = [1 -1 0];  
loverr = [-1 1 0];  
  
%-----  
% batch variables defined here for analysis 'contrasts'  
%-----  
  
contrasts(1) = struct( ...  
    'names',    {{'Ronly', 'Lonly', 'RoverL', 'LoverR'}}, ...  
    'types',    {{'T',      'T',      'T',      'T'}}, ...  
    'values',   {{ronly,  lonly,  roverl,  loverr}} ...  
);
```

## **4 Summary**

Batch mode allows a lot of processing to be done nearly automatically by eliminating the SPM GUI for user input. However, because of its enormous flexibility, it can be quite complicated at first. A good understanding of Matlab syntax is essential to take advantage of batch mode. Batch mode requires two text files to specify all aspects of the processing. The analyses file defines which types of processing are done and where to find the variable description files. The variable description file contains all of the input that would be required in the SPM GUI for each step of the processing. Batch mode is easiest when one uses files that have already worked and changes the items that need to be changed. The batch manual 'spm\_bch.man' provides a description of all the variables that are used in any type of processing. By using that file and this guide, batch mode will provide a tremendous increase in efficiency in the analyses of functional brain imaging with SPM99.