Data Wrangling in Social Science and Psychophysiology using Matlab

Thomas E. Gladwin

Introduction

One of the basic skills in doing research is data wrangling: understanding everything about how your observations are stored and being able to do anything necessary with them to get to the point where you can do statistics. If you fail to save your data correctly, or can't correctly access it, or don't know how to go from raw data to something you can do statistics on, everything else fails – hopefully noticeably, but you might end up unwittingly drawing completely false conclusions from junk. It goes further though: if you don't have the skills to wrangle data at the preprocessing step, you probably don't have the skills to perform more complex analyses, let alone develop new ones. So it's good that you're here, starting on this book!

This brief introduction to data preprocessing is aimed at providing the minimal groundwork for handling real data and further developing fluency. The specific language / system used here is Matlab; however, if you want to perform the same conceptual steps using, e.g., R or Python, the translation should be relatively simple.

In the following sections, it's essential to **actually type in the code as it's presented**. The book isn't intended to just be read though: it has to be **actively** worked through. A lot of ideas are presented as code at least as much as explanatory text. It's further expected that readers have the attitude of not hesitating to search for additional information on anything that's unclear. I assume a basic knowledge of Matlab commands, .m-files, and functions; there are lots of decent online tutorials, and otherwise check the help / docs or Google for anything unfamiliar. An important and very useful capability to check out is the Matlab editor's debug function.

Unless otherwise specified, every time a function is introduced, type it into its own m-file. Where new lines have to be placed somewhere in existing code, the new lines will be printed in bold with some lines of the context around them.

Behavioural data

This section is concerned with behavioral performance data: reaction times and accuracy.

Simulating reaction times and accuracy

The first step is always simulating data, and everyone always gets lazy and skips it. Try not to! The programs are relatively complex and the best way to make sure there isn't a dysfunction in them is to use data about which you know exactly what the correct result has to be.

First, we'll simulate the data for a single subject. We'll do this in a function that takes one argument, the subject number. We'll use 120 trials, and make up a 2 x 3 within-subject full factorial design, resulting in 6 within-subject conditions. Make a new m-file called beh_sim_data_subject and start off with this:

```
function beh_sim_data_subject(iSubj)


%% Set up parameters

nTrials = 120;

nFactors = 2;

nLevelsPerFactor = [2 3];
```

Now we'll set up the pattern of results we'll need to see when we check the data. This pattern will be placed in vectors with 6 elements. Elements 1:3 contain the values for the 3 levels of Factor 2 for level 1 of Factor 1. Elements 4:6 contain the values for the 3 levels of Factor 2 for level 2 of Factor 1. That is, we've **nested** Factor 2 in Factor 1.

Since nesting is such an important concept, I'll elaborate on it to a possibly irritating degree here. Assume that your experiment has a number of independent variables, and every condition in the experiment is defined by its combination of values on each independent variable. Nesting is a standard method of ordering the conditions according to their combination of values of independent variables (or levels of factors, in slightly different (e.g., SPSS) terminology).

Say you have three factors, A, B, and C. A and B have two levels, C has three. If you wrote a function like this:

```
condition = 1

for A = 1 to 2

        for B = 1 to 2

                for C = 1 to 3

                        print condition A B C
```

```
                    condition = condition + 1
                end
            end
        end
```

You'd see a list appear like this (adding some imaginary formatting):

Condition 1: A = 1, B = 1, C = 1

Condition 2: A = 1, B = 1, C = 2

Condition 3: A = 1, B = 1, C = 3

Condition 4: A = 1, B = 2, C = 1

Condition 5: A = 1, B = 2, C = 2

Condition 6: A = 1, B = 2, C = 3

Condition 7: A = 2, B = 1, C = 1

Condition 8: A = 2, B = 1, C = 2

Condition 9: A = 2, B = 1, C = 3

Condition 10: A = 2, B = 2, C = 1

Condition 11: A = 2, B = 2, C = 2

Condition 12: A = 2, B = 2, C = 3

This is called "nesting C in B in A", or A is the highest level, B the second level, and C the third. Note the structure of the factor levels. As the condition index increments, factor C loops through its factor levels each time. The level of factor B only increases every time C has gone through a loop. Factor A only increases once factor B has gone through a loop.

Alternatively to the pseudocode representation of nesting, you can draw a tree diagram with A on the top, with two branches for its levels; then factor B's two branches at the ends of each of A's branches; and then factor C's three branches at the end of each of those branches. Reading off the ends of those branches from left to right gives you the condition number.

The point of this is that if you organize (usually within-subject) data in columns, where the *m*-th column represents the data from condition m according to a specified nesting structure, we can use a single number to represent any combination of factor levels.

With all that in mind, add the following lines:

```
acc_pattern = [0.7 0.7 0.7 0.5 0.7 0.9];

rt_pattern = [500 500 900 300 300 300];
```

Our code specifies that, for accuracy, there's no effect of Factor 2 given level 1 of Factor 1; but given level 2 of Factor 1, accuracy increases over the levels of Factor 2. For RT, we see that Factor 2 has no effect given level 2 of Factor 1; but given level 1 of Factor 1, level 3 of factor 2 is much slower than the other levels. Further, level 2 of Factor 1 is also slower than level 1, averaged over levels of Factor 2.

Now we add code to simulate the data, with these patterns plus some noise. To do so, we loop over trials. On each iteration of the loop, we randomly select a level for each of the Factors. The randomization uses the *rand* function, which returns a real number between 0 and 1. If we want to get a range of whole numbers from 1 to N, we multiply that (0, 1) number by N, and remove the trailing decimal places. This results in number from 0 to N – 1, so we add 1.

```
%% Simulate data trials

Data = [];

for iTrial = 1:nTrials,

   % Select condition at random

   Factor1 = 1 + floor(rand * nLevelsPerFactor(1));

   Factor2 = 1 + floor(rand * nLevelsPerFactor(2));


end;
```

We then use these levels to select the associated RT and accuracy from the pattern matrix, by calculating the index based on the nesting structure. Remember that Factor 2 is nested within Factor 1: so we count 1 – 2 – 3 for the level of Factor 2, from the appropriate starting point given the level of Factor 1. Add this code within the loop, after the Factor randomization code:

```
   % Find base value for accuracy and RT

   iCond = (Factor1 - 1) * nLevelsPerFactor(2) + Factor2;

   acc_base = acc_pattern(iCond);
```

```
rt_base = rt_pattern(iCond);
```

Finally, we add some noise to these base values, using *rand* again, and then add the trial-data as a new row to the Data matrix. Note that now we're using *rand* to adjust the values proportionally, from a factor of 0.9 to 1.1. Add this code to the loop:

```
% Add some noise

acc_base = acc_base * (0.9 + 0.2 * rand);

rt_base = rt_base * (0.9 + 0.2 * rand);


% Add trial to data matrices

this_row = [Factor1 Factor2 acc_base rt_base];

Data = [Data; this_row];
```

Finally, below the loop, we save the data to a simple text (ASCII) file. The filename contains the subject number. The content of the file starts with one line of text containing the variable names, and then the numbers in the Data matrix, all separated by tabs.

```
%% Save data to a text file.

% Create filename based on subject number

filename = ['sim_beh_subj' num2str(iSubj) '.txt'];

% Write a header line with variable names

fid = fopen(filename, 'w');

fprintf(fid, 'Factor1\tFactor2\tAcc\tRT\n');

fclose(fid);

% Append data matrix to file

dlmwrite(filename, Data, 'delimiter', '\t', '-append');
```

Life will be a lot easier if you save your data as numbers. You can save a second file containing the mapping of numbers to string values if you like, but mixing up numbers and strings in the primary data file can make things a lot messier and more error-prone. You can't just use the simple dlmread function to get at the data, for example.

Run the function for a subject number you like.

beh_sim_data_subject(1)

This will result in a file in the working directory; open with a text editor to check its contents.

Reading a data file

Now we have a text file to read. The basic command to read nicely-saved numerical data is dlmread. Since we've added a header line, we have to add arguments to skip that line in the dlmread function (if this is unfamiliar: type *help dlmread*).

```
function D = beh_read_subject_data(filename)


% Read matrix of numbers

D = dlmread(filename, '\t', 1, 0);
```

Run the function with the name of the file with the simulated data. Check the filename by typing ls or dir; it will depend on the subject number you chose.

D = beh_read_subject_data('sim_beh_subj1.txt');

Check for a few lines whether the content of the matrix D (for Data) match the numbers in the text file.

Parsing data: getting the means per combination of factor levels

At this stage, the data are organized as a long matrix with trials for each condition all mixed together. We'll generally want to get the means for each condition; I use the term parsing for this kind of minimal transformation of raw data into something more usable. Let's do this for RT: we loop over all possible conditions, find the trials belonging to each condition, and get the mean RT over that subset of trials. Remind yourself what the different columns mean in the simulated data.

```
function rt_vec = beh_parse(D)


rt_vec = [];

for Factor1 = 1:2,

   for Factor2 = 1:3,

      f = find(D(:, 1) == Factor1 & D(:, 2) == Factor2);

      mean_RT = mean(D(f, 4));

      rt_vec = [rt_vec mean_RT];

   end;
```

```
end;
```

Run the function on some simulated data (first read the data from file into a matrix D, as in the previous section, if it's not already in the workspace):

```
rt_vec = beh_parse(D)
```

Note that the means vector contains a noisy version of the pattern we put into the simulated data: so now we know everything's working as expected.

Now we add the code to also get accuracy data; see if you can copy and adjust the code for RT data yourself first.

```
function [rt_vec, acc_vec] = beh_parse(D)


rt_vec = [];

acc_vec = [];

for Factor1 = 1:2,

   for Factor2 = 1:3,

      f = find(D(:, 1) == Factor1 & D(:, 2) == Factor2);

      mean_RT = mean(D(f, 4));

      rt_vec = [rt_vec mean_RT];

      mean_acc = mean(D(f, 3));

      acc_vec = [acc_vec mean_acc];

   end;

end;
```

Run the code and check whether the correct pattern is in there:

```
[rt_vec, acc_vec] = beh_parse(D)
```

Reading and organizing data from a sample of subjects

Now we can simulate, read, and parse data for one subject, let's expand to a sample of multiple subjects. First, we'll create a simple batch-file that runs the simulation function for multiple subjects:

```
function beh_batch_sim_data
```

```
nSubj = 10;

for iSubj = 1:nSubj,

   beh_sim_data_subject(iSubj);

end;
```

The term “batch” file means that it contains a sequence of similar actions we’re to perform in one run. Now we want to read the sample data. The first step is to find all the relevant files in our directory, and then to loop over them. In the workspace, play around with the dir function:

```
dir_struct = dir('sim_beh_subj*.txt')

dir_struct(1)

dir_struct(1).name
```

Note that a struct is a variable that contains other variables. The dir function returns an array of structs; each struct contains information about one of the files returned by the dir command.

So what we do is, use dir to find our simulated files, and loop over them. On each iteration, we read the data of the *n*-th file in the struct. The RT and accuracy vectors for that file are added as addended rows to a big matrix that will eventually contain the parsed data for all the subjects. Finally, we save the data to a text file that we could read into further analysis functions or, e.g., SPSS.

```
dir_struct = dir('sim_beh_subj*.txt');

RT = [];

Acc = [];

for iFile = 1:length(dir_struct),

   filename = dir_struct(iFile).name;

   D = beh_read_subject_data(filename);

   [rt_vec, acc_vec] = beh_parse(D);

   RT = [RT; rt_vec];

   Acc = [Acc; acc_vec];

end;
```

```
dlmwrite('RT.txt', RT, 'delimiter', '\t');

dlmwrite('Acc.txt', Acc, 'delimiter', '\t');
```

Data checks

Let's read the parsed data for the whole sample and see whether everything looks good. We can read the data using dlmread, since we kept everything nice and simple. Let's check RT first. We read the data, and print means, minima and maxima, and standard deviations per column.

```
function beh_data_check


filename = 'RT.txt';

RT = dlmread(filename);


fprintf('Means:\t');

scores_per_column = mean(RT);

fprintf([num2str(scores_per_column) '\n']);


fprintf('Mins:\t');

scores_per_column = min(RT);

fprintf([num2str(scores_per_column) '\n']);


fprintf('Maxes:\t');

scores_per_column = max(RT);

fprintf([num2str(scores_per_column) '\n']);


fprintf('SD:\t');

scores_per_column = sqrt(var(RT));

fprintf([num2str(scores_per_column) '\n']);
```

It's also always good to plot means, together with standard deviations (arguably, when presenting data that are tested using within-subject methods, standard errors based on scores after removal of between-subject variance are more suitable). Add the code to the beh_data_check m-file.

```
figure;

means0 = mean(RT);

sds0 = sqrt(var(RT));

plot(means0, 'k-');

hold on;

plot(means0 - sds0, 'k--');

plot(means0 + sds0, 'k--');
```

Note on running the file that we see the pattern that was imposed on the data in the simulation.

For exercises, do the same data checks for accuracy, and see what happens to the final plot when the noise is increased in the simulations. Also return and check the number of files per condition, in addition to RT and accuracy.

Using the header

At this point, you're able to take simple raw data and save it in a form suitable for statistical analyses. However, real data will tend to be more complex, and you won't want to have to remember which columns contain what, especially if there's chance things could change at some point, in which case you'd have to go through all your code to make sure there are no incorrect column numbers. If you can trust the header line, there's a better way to do this. We'll first make a second version of the data-reading function that returns a cell array of strings containing the variable names. If you're not familiar with them, cell arrays are ordered lists that can contain any type of data as their elements. Look them up to check out how to use them.

```
function [D, varnames] = beh_read_subject_data_2(filename)


%% Read variable names

% Read the first line

fid = fopen(filename);

header_line = fgetl(fid);

fclose(fid);
```

```
% Parse the line into a cell array of strings

varnames = regexp(header_line, '\t', 'split');


%% Read matrix of numbers

D = dlmread(filename, '\t', 1, 0);
```

The regexp function performs regular expressions, very powerful methods of manipulating string data. Our application is very simple though: we just tell it to split up a line of text, and put the words separated by tabs into separate cells. We return these variable names along with the data.

Now, we'll use the variable names in the new version of the parsing function. Instead of hard-coding the column numbers, we'll use a helper function to return the column number based on the variable name. We can simply add such a helper function to the bottom of the code, and it will be locally available. So this is an exception to the rule that every function gets its own m-file, since it's only useful in the context of the main function of the m-file.

```
function [rt_vec, acc_vec] = beh_parse_2(D, varnames)


colFactor1 = var2col('Factor1', varnames);

colFactor2 = var2col('Factor2', varnames);

colRT = var2col('RT', varnames);

colAcc = var2col('Acc', varnames);


rt_vec = [];

acc_vec = [];

for Factor1 = 1:2,

   for Factor2 = 1:3,

      f = find(D(:, colFactor1) == Factor1 & D(:, colFactor2) == Factor2);

      mean_RT = mean(D(f, colRT));

      rt_vec = [rt_vec mean_RT];

      mean_acc = mean(D(f, colAcc));
```

```
        acc_vec = [acc_vec mean_acc];
    end;
end;


function column_number = var2col(label, varnames)
column_number = 0;
for iVar = 1:length(varnames),
    if strcmp(label, varnames{iVar}) == 1,
        column_number = iVar;
    end;
end;
if column_number == 0,
    error('Unknown variable name.');
end;
```

With our very simple data file, this might seem like overkill, but when files might be saving dozens of variables a helper function like this is well worth it.

Organizing data with any nesting structure: the generalized parser

In the simulated data we knew we had two factors, and we knew that they have 2 and 3 levels each, and the parsing function assumed that knowledge. It's also possible to write a generalized parsing function, where you don't have to hard-code variables and levels. You can then just provide a cell array with the variable names of factors you want to base the means on. This is a maybe little more complicated than necessary in a first introduction, so I present it here as an exercise for the interested and perhaps not totally beginner-level reader. Feel free to skip, but it might be good to be aware that there's a level of programming beyond the very simple scripts we're developing here.

First, we need a function that will run through the space of all combinations of levels of an arbitrary set of columns, and get the mean (and some other useful information) from each associated set of trials. This is where the complex stuff is happening; there are various helper functions and recursion is used: calling a function from within the same function. It's very powerful but takes some getting used to. As a unique exception, don't feel obliged to type in this code or fully understand it; just download it from the companion website.

```
function [meanvec, idmat, Nvec, SDvec] = rec_parser(varargin)


depvec = varargin{1};

indepvec = varargin{2};


% Make sure all combinations are included and estimated if necessary

levels = [];

for iVar = 1:size(indepvec, 2),

   u = unique(indepvec(:, iVar));

   levels = [levels length(u)];

end;


indepvec0 = expand_combinations(levels, 0);

for iVar = 1:size(indepvec0, 2),

   u = unique(indepvec(:, iVar));

   tmp = indepvec0(:, iVar);

   for iu = 1:length(u),

      f = find(tmp == iu);

      indepvec0(f, iVar) = u(iu);

   end;

end;

depvec0 = NaN * ones(size(indepvec0, 1), 1);

depvec = [depvec0; depvec];

indepvec = [indepvec0; indepvec];

[meanvec, idmat, Nvec, SDvec] = rec_combo_inner(depvec, indepvec, [], 1, [], [], [], []);
```

```
function [meanvec, idmat, Nvec, SDvec] = rec_combo_inner(depvec, indepvec, meanvec, depth,
idmat, idval, Nvec, SDvec)

if (depth == size(indepvec, 2)),
   u = unique(indepvec(:, depth));
   for iu = 1:length(u),
      f = find(indepvec(:, depth) == u(iu));
      idval(1, depth) = iu;
      selected = depvec(f);
      selected(find(isnan(selected))) = [];
      if ~isempty(selected),
         newmeanvec = mean(selected);
         meanvec = [meanvec newmeanvec];
         SDvec = [SDvec sqrt(var(selected))];
      else,
         meanvec = [meanvec NaN];
         SDvec = [SDvec NaN];
      end;
      idmat = [idmat; idval];
      Nvec = [Nvec length(selected)];
   end;
else,
   u = unique(indepvec(:, depth));
   for iu = 1:length(u),
      f = find(indepvec(:, depth) == u(iu));
      idval(1, depth) = iu;
```

```
      [meanvec, idmat, Nvec, SDvec] = rec_combo_inner(depvec(f), indepvec(f, :), meanvec, depth
+ 1, idmat, idval, Nvec, SDvec);
   end;
end;

function id_matrix = expand_combinations(levels, perm0)
% id_matrix = expand_combinations(levels, perm0)
%
% levels = [n1 n2 ... nm]
% perm0: if 1 only permutations are returned
%
% Thomas Gladwin, 2007.
level_vec = 0 * levels;
id_matrix = rec_loop2_inner(levels, level_vec, [], 1, perm0);
if perm0 == 1,
   frem = [];
   for ir = 1:size(id_matrix, 1),
      u = unique(id_matrix(ir, :));
      if length(u) < size(id_matrix, 2),
         frem = [frem; ir];
      end;
   end;
   id_matrix(frem, :) = [];
end;

function id_matrix = rec_loop2_inner(levels, level_vec, id_matrix, current_factor, perm0)
```

```
if current_factor == length(levels),
   loopto = levels(current_factor);
   for iLevel = 1:loopto,
      level_vec(current_factor) = iLevel;
      % check
      beenhere = 0;
      if perm0 == 1,
         s0 = sort(level_vec);
         for iCheck = 1:size(id_matrix, 1),
            s1 = sort(id_matrix(iCheck, :));
            if length(s0) ~= length(s1),
               continue;
            end;
            if s0 == s1,
               beenhere = 1;
               break;
            end;
         end;
      end;
      if beenhere == 0,
         id_matrix = [id_matrix; level_vec];
      end;
   end;
else
   loopto = levels(current_factor);
```

```
    for iLevel = 1:loopto,
        level_vec(current_factor) = iLevel;
        id_matrix = rec_loop2_inner(levels, level_vec, id_matrix, current_factor + 1, perm0);
    end;
end;
```

We then use this function in a parsing function that allows us to provide simple input and then prepares things to call the generalized parser for us. Note that the complexity is all nicely packaged in the general function; the new bolded code in the shell-function is nice and simple. We just have to transform the labels of the desired factors into column numbers; extract a dependent vector and a matrix of independent variables; and call the generalized parser function.

```
function rt_vec = beh_parse_3(D, varnames, factor_labels)

factor_columns = [];
for iFactor = 1:length(factor_labels),
    factor_columns(iFactor) = var2col(factor_labels{iFactor}, varnames);
end;

dep = D(:, var2col('RT', varnames));
indeps = D(:, factor_columns);

rt_vec = rec_parser(dep, indeps);

function column_number = var2col(label, varnames)
column_number = 0;
for iVar = 1:length(varnames),
    if strcmp(label, varnames{iVar}) == 1,
        column_number = iVar;
```

```
    end;
end;
if column_number == 0,
    error('Unknown variable name.');
end;
```

Applying this parsing function would be done as usual:

```
fn = 'sim_beh_subj1.txt';
[D, varnames] = beh_read_subject_data_2(fn);
rt_vec = beh_parse_3(D, varnames, {'Factor1', 'Factor2'});
```

Feel free to use the generalized parser, at your own risk and after you've convinced yourself via testing that it does what it's supposed to.

Preprocessing during parsing

The parsing step is often used to manipulate the data in some way. For example, you might want to remove practice blocks, reaction times that are outliers, or trials following errors. This can be done by adjusting the Data matrix prior to calculating the mean. A note of warning is that this step is very easily abused for *p*-hacking. Be as strict as possible and if at all feasible decide once and for all, before any data analysis, how you'll do the preprocessing. Otherwise the only ethical option is to report results for multiple variants of preprocessing.

Reading complex files with non-numerical values

So imagine you have data that aren't saved as nice, consistent, numerical matrices. After hunting down and brutally murdering whoever is responsible, you're still stuck with having to deal with the files. The precise approach will depend on how the details of how your specific data were saved, but you'll basically use the same functions as you did for reading the header.

Let's say the contents of the data file looks like this:

```
Trial1  ConditionA  454
Trial2  ConditionB  631
Trial3  ConditionA  304
...
```

You'd need to loop over the lines of the file, and parse each line using regexp and string manipulation functions. Just to illustrate the mess and provide some tips in case you end up in this situation, it would

look something like this. We start off with simulating some data containing text, and then parse it into something numerical.

```
function beh_sim_text_data


filename = 'beh_sim_text.txt';


fid = fopen(filename, 'w');

fprintf(fid, ['Trial1\tConditionA\t354\n']);

fprintf(fid, ['Trial2\tConditionB\t1621\n']);

fprintf(fid, ['Trial3\tConditionA\t264\n']);

fprintf(fid, ['\n']);

fprintf(fid, ['Trial4\tConditionA\t208\n']);

fprintf(fid, ['Trial5\tConditionB\t1200\n']);

fprintf(fid, ['Trial6\tConditionBA\t1534\n']);

fprintf(fid, ['\n']);

fclose(fid);
```

Check whether this does indeed create a text file called 'beh_sim_text.txt' in which we have these "Trial1" etc values as data. So now we have to deal with that. Using dlmread won't work – try it and bask in the error message.

We'll now work through the code of a function to parse the text-contaminated data into a matrix NumData. We start off by opening the file, and reading lines until we hit the end of the file (check the *fopen* and *feof* help if necessary). We'll add a command to print the incoming lines so we know what's going on, and where, if anywhere, something goes wrong.

```
function NumData = parse_texted_data(filename)

NumData = [];
fid = fopen(filename, 'r');
while ~feof(fid),
   this_line = fgetl(fid);
   fprintf([this_line '\n']);
end;
```

We can this using NumData = parse_text_data('beh_sim_text.txt'). Check whether the lines that are read in match the content of the file.

Now we can deal with the incoming lines. What we want to do is take the characters strings and transform them into numbers: the trial number, condition number, and reaction time. To do so, we use the lovely regexp function again:

```
NumData = [];
fid = fopen(filename, 'r');
while ~feof(fid),
   this_line = fgetl(fid);
   fprintf([this_line '\n']);

   if isempty(this_line),
      continue;
   end;
   words = regexp(this_line, '\t', 'split');
   disp(words);
end;
```

Run the file to see how we get the cell array of separated words from the character string. We have to check for empty lines to avoid getting an error later when trying to parse on empty lines.

Now we've reduced the problem to a set of words we have to deal with. We're slowly but steadily getting there! First let's parse the word containing the trial strings. Add this code inside the loop, after the code that creates the *words* variable.

```
% Parse Trial number
trialString = words{1};
trialString = trialString(6:end); % Skip the prepended "Trial" characters
trialNumber = str2num(trialString);
disp(trialNumber);
```

For convenience, we put the first word into a variable with a recognizable name. This variable will contain text like "Trial1", "Trial2", and so on. We have to isolate the numerical part of the string: that is, we have to skip the letters of "Trial". This numerical string can be transformed into a number variable using str2num.

For the condition string, we need to transform the "ConditionA" and "ConditionB" strings into numbers that represent the respective conditions. For this, we add some code doing a string comparison:

```
% Parse Condition
conditionString = words{2};
if strcmp(conditionString, 'ConditionA'),
   condition = 1;
```

```
else,
   condition = 2;
end;
disp(condition);
```

Getting the RT is easy now: we just apply str2num on the third word:

```
% Get RT
RT = str2num(words{3});
disp(RT);
```

Finally, at the bottom of the code contained within the loop, we add a row to the NumData matrix:

```
NumData = [NumData; trialNumber condition RT];
```

Run the function and see how the NumData now contains the numerical representation of the data, which allows us much easier further analysis. Since data like this usually have nasty surprises, I added some empty lines. See what happens if you comment out the check for empty lines at the top of the loop code.

Psychophysiology

This section is concerned with psychophysiological signals: bodily data that are recorded continuously over time, which we expect to change in response to certain psychologically relevant events such as stimulus presentation. Examples are heart rate, pupil dilation, or the voltage at EEG electrodes.

Simulating continuous data and event-related waveforms

Again, we start with simulating data. This might also help get a grip on what kind of effects we expect to find. We'll be simulating a measurement at 100 Hz, for two signals. Events will occur at intervals of 1000 to 3000 ms, and will be saved to a marker file. One of the event-types will cause an enlarged waveform in the first signal; the other event will cause a 300 ms-delayed waveform in the second signal. We'll also save a basic header file: header files contain information about the measurement, usually as plain ASCII text.

We'll start the simulating function with the header. We need to know the sampling frequency and the number of channels for further analysis. We also need to specify the duration of measurement time to be simulated; here that will be two minutes.

```
function phys_sim
```

```
base_filename = 'sim_physio';
```

```
%% Header information


% Parameters
Fs = 100; % Sampling frequency
T = 120; % Time of measurement
nSamples = floor(T * Fs); % Number of samples (or time points).
nSignals = 2;


% Save to header file
header_filename = [base_filename '.hdr'];
fid = fopen(header_filename, 'w');
fprintf(fid, ['Fs = ' num2str(Fs) '\n']);
fprintf(fid, ['NSignals = ' num2str(nSignals) '\n']);
fclose(fid);
```

Check the header file to see if it looks correct. Now we add events markers. Event markers allow you to connect the continuously measured signals to the events occurring in your task. Without markers **you have no data**. The alternative to using event markers is when you use a triggering marker, which synchronizes the start of either the task or the recording. This is the usual approach in fMRI: the task starts when the *N*-th scan is detected. In that case, it's safest to save the clock time of every event including the arrival of the triggering scan. In psychophysiology like EEG or ECG recording, the use of event markers is more common.

We'll be simulating events that have two event codes, representing two different conditions. Often, the type of event will be saved in a separate file: this simulates the usual situation in practice, where you're recording using one computer, and presenting a task on another one. Task information and performance data are saved only on the second computer. You have to read in both the marker information in the recording data and the output from the task to analyze the data. We're keeping it simple here.

Add the following code. What we're doing is iteratively creating markers that will appear at a time point, *next_event_time*, 1 to 3 s after the previous marker. We convert these times (in seconds) to the corresponding sample-point in the signal, by using the sampling frequency *Fs*. For example, if *Fs* is, say,

1000 Hz, then 3.5 seconds into recording occurs at sample 3500. We also generate a random event_type, that can be 1 or 2, using the rand function.

```
%% Events

% Parameters for event timing
min_event_interval = 1;
max_event_interval = 3;

% Simulate events, and save event type and event sample to Markers
Markers = [];
next_event_time = 5;
while next_event_time < T,
   event_sample = ceil(next_event_time * Fs);
   event_type = 1 + floor(2 * rand);

   disp([event_type event_sample]);
   Markers = [Markers; event_type event_sample];

   random_interval = min_event_interval + (max_event_interval - min_event_interval) * rand;
   next_event_time = next_event_time + random_interval;
end;
```

After the markers have been simulated, we save them to a simple file.

```
% Save to marker file

marker_filename = [base_filename '.mrk'];

dlmwrite(marker_filename, Markers, 'delimiter', '\t');
```

Now we create the signals. A signal is a long vector of values representing measurement at consecutive time points. The time between measurements is defined by the sampling frequency; if the sampling frequency is *Fs*, then the time between samples is 1 / *Fs*. For example, if the sampling frequency is 200 Hz, i.e., 200 samples per second, then the time between samples is 1 / 200 = 0.0050 s or 5 ms.

We're going to put simple waveforms in the signals, locked to the events we just simulated. In order to test the preprocessing pipeline we're going to build, the waveforms will be different for the different signals and conditions, i.e., event codes. Start off by adding this code, which will create the waveforms for event type 1:

```
%% Signals
```

```
% Create a Signals matrix with zeros for now

Signals = zeros(nSamples, nSignals);


% Create waveforms for event type 1: bigger in signal 1

find_events = find(Markers(:, 1) == 1);

for iEvent = 1:length(find_events),

   this_sample = Markers(find_events(iEvent), 2);


end;
```

This first finds all events of type 1, which are saved in the first column of the Markers matrix. Then it runs through all these events and gets the sample at which it occurs, from the second column.

Now we're going to create a basic waveform. At the bottom of the code, add this helper function:

```
function basic_waveform = make_basic_waveform()
basic_waveform = [1:100 100:(-4):0];
basic_waveform = basic_waveform(:); % turn the waveform into a column for consistency.
basic_waveform = basic_waveform ./ max(basic_waveform); % Normalize the amplitude of the
waveform.
```

This creates a very simple waveform, which ramps up to a peak and then relatively quickly decreases. We call this function once, before looping over events:

```
%% Signals

% Create a Signals matrix with zeros for now
Signals = zeros(nSamples, nSignals);

% Create a basic waveform
basic_waveform = make_basic_waveform();
```

Now we add this waveform to the signal around events. We have to precisely specify into which range of samples it will go, and make sure the event doesn't occur so close to the end of the sampling period that it would go out of bounds.

```
% Create waveforms for event type 1: bigger in signal 1
```

```
find_events = find(Markers(:, 1) == 1);
for iEvent = 1:length(find_events),
   this_sample = Markers(find_events(iEvent), 2);

   sample_start = this_sample;
   sample_end = sample_start + length(basic_waveform) - 1; % Note that if the waveform was
1 sample long, you would add zero to get the element of its end-point relative to starting point.
   if sample_end > nSamples,
      continue;
   end;

end;
```

Now we add the basic waveform to the signals, from sample_start to sample_end. In signal 1, we further *increase the size of the waveform. Add the code for that to complete the simulation for event type 1:*

```
% Create waveforms for event type 1: bigger in signal 1
find_events = find(Markers(:, 1) == 1);
for iEvent = 1:length(find_events),
   this_sample = Markers(find_events(iEvent), 2);

   sample_start = this_sample;
   sample_end = sample_start + length(basic_waveform) - 1; % Note that if the waveform was 1
sample long, you would add zero to get the element of its end-point relative to starting point.
   if sample_end > nSamples,
      continue;
   end;

   Signals(sample_start:sample_end, 1) = 2 * basic_waveform;
   Signals(sample_start:sample_end, 2) = basic_waveform;
end;
```

For event type 2, we'll be using slightly different *sample_start* and *sample_end* values for the two signals to create the time-shift we want to simulate. The rest of the code is analogous to the code for event type 1.

```
% Create waveforms for event type 2: 0.3 s later in signal 2
find_events = find(Markers(:, 1) == 2);
for iEvent = 1:length(find_events),
   this_sample = Markers(find_events(iEvent), 2);

   sample_start = this_sample;
   sample_end = sample_start + length(basic_waveform) - 1;
   if sample_end > nSamples,
      continue;
   end;
```

```
    Signals(sample_start:sample_end, 1) = basic_waveform;

    sample_start = this_sample + ceil(0.3 * Fs);
    sample_end = sample_start + length(basic_waveform) - 1;
    if sample_end > nSamples,
        continue;
    end;
    Signals(sample_start:sample_end, 2) = basic_waveform;
end;
```

Finally, we add some noise to the data and save it to a binary (as opposed to text) file, in which the values are represented as 32-bit floating point numbers (look up floating point numbers on Wiki if necessary). We'll need to read in these data using the same floating-point representation.

```
% Add noise
Signals = Signals + randn(size(Signals));

% Save signal data to a 32-bit floating point binary file
signal_filename = [base_filename '.sig'];
fid = fopen(signal_filename, 'w');
fwrite(fid, Signals, 'float32');
fclose(fid);
```

Reading header, signals, and markers

The details of reading data will depend on the system you're using, but one way or another will come down to reading the sampled measurements of continuous data, and a list of time-points when events happened. I like using simple events – one marker per trial - and then combining them with the rich data saved by the stimulus presentation software, as this reduces dependence on hardware-specific capabilities. This does necessitate saving EVERYTHING, especially the time of every thinkable event within trials, so that only the one marker is needed to reconstruct when anything happened in the task.

Reading the header uses the same kind of code as reading a nasty text-containing data file:

```
function H = read_header(header_filename)


fid = fopen(header_filename, 'r');

while ~feof(fid),

    this_line = fgetl(fid);

    words = regexp(this_line, ' = ', 'split');
```

```
    if strcmp(words{1}, 'Fs') == 1,
      H.Fs = str2num(words{2});
    end;
    if strcmp(words{1}, 'NSignals') == 1,
      H.NSignals = str2num(words{2});
    end;
  end;
```

Run it as follows to load the header information into the workspace:

```
H = read_header('sim_physio.hdr')
```

Reading markers, for our nice simulated data, is a matter of using dlmread. Often the marker file will also require more complex text parsing (as with BrainVision marker files).

```
function M = read_markers(marker_filename)


M = dlmread(marker_filename, '\t');
```

And finally, we read the signals. An important variation in how signals are saved is vectorized versus multiplexed. In vectorized data, all the time points of a signal are saved consecutively: first all of one signal, then all of the next, and so on. In multiplexed data, all the signal values for the first time point are saved; then all the signal values for the second time point; and so on. Now, when Matlab reads data into a matrix, it fills in each column from top to bottom at a time. So when reading or reshaping vectorized data, the size of the matrix should be as follows: as many rows as there are samples, and as many columns as there are signals. If we want to select the *n*-th signal from this matrix, we'd select the *n*-th column. In contrast, when reading multiplexed data, we need to read it into a Signals x Timepoints matrix, and then transpose it before we can select columns to select signals. We simulated the data to be vectorized, and in general you need to be aware of the saving method. If you're not sure, it may be indicated in the header (this is the case for BrainVision data for instance). Sounds complicated, but it boils down to a simple function:

```
function Signals = read_signals(signal_filename, nSignals)


fid = fopen(signal_filename, 'r');
Signals = fread(fid, 'float32');
```

```
nSamples = length(Signals) / nSignals;


Signals = reshape(Signals, nSamples, nSignals);
```

Note that we need to use a bit of information we got from the header: the number of signals. We call the function as follows:

```
Signals = read_signals('sim_physio.sig', H.NSignals)
```

Since we'll usually want all the information from the files, we'll cluster these functions into an overall read_data function:

```
function [H, M, Signals] = read_data(basename)


H = read_header([basename '.hdr']);

M = read_markers([basename '.mrk'])

Signals = read_signals([basename '.sig'], H.NSignals);
```

Note that we now only provide the base filename, without the extension, as an argument.

```
basename = 'sim_physio'; [H, M, Signals] = read_data(basename)
```

Parsing: Extracting epochs based on task conditions and response data

The parsing we have to do for physiological data involves getting stretches of samples in an epoch, or time window, surrounding all events of a certain type. We specify the epoch in terms of how much time before and after the event we include. We'll be providing the Signals, Markers and Header information as arguments.

```
function Waveforms = get_epochs(Signals, M, H)

% Set up the epoch time window: we'll extract intervals from 1 s before the
% event, to 3 s after it.
pre_event_time = 1;
post_event_time = 3;

% Convert the time window into samples.
pre_event_samples = ceil(pre_event_time * H.Fs);
post_event_samples = ceil(post_event_time * H.Fs);
```

For each event, we're going to find the range of samples representing the surrounding epoch. These samples are added to matrices containing the data of all the epochs. In Matlab, this is convenient to do with cell arrays, with event type and signal as indices.

```
% Set up the Epochs cell array with empry matrices
Epochs = {};
for iEventType = 1:2,
   for iSignal = 1:size(Signals, 2),
      Epochs{iEventType, iSignal} = [];
   end;
end;
```

Now we're going to loop over the event types, and for each event type, loop over all events of that type. For each event, we want to get the sample at which the event occurred.

```
% Loop over event types and all events per type
for iEventType = 1:2,
   find_events = find(M(:, 1) == iEventType);

   for iEvent = 1:length(find_events),
      event_sample = M(find_events(iEvent), 2);

      end;

   end;
end;
```

Now we add code to determine the range of samples containing the epoch surrounding the event.If those get us into illegal ranges, we ignore the event.

```
% Loop over event types and all events per type
for iEventType = 1:2,
   find_events = find(M(:, 1) == iEventType);

   for iEvent = 1:length(find_events),
      event_sample = M(find_events(iEvent), 2);
      sample_start = event_sample - pre_event_samples;
      sample_end = event_sample + post_event_samples;

      if sample_start < 1 || sample_end > size(Signals, 1),
         continue;
      end;

   end;
end;
```

Now we have the samples that belong to this epoch, we can run over the signals, grab the samples from each signal, and put them into the matrix in the appropriate cell of the Epochs cell array. Epochs are organized rowwise because the default way the mean function works is over rows, and because the visualization seems more natural to me.

```
% Loop over event types and all events per type
for iEventType = 1:2,
   find_events = find(M(:, 1) == iEventType);

   for iEvent = 1:length(find_events),
      event_sample = M(find_events(iEvent), 2);
      sample_start = event_sample - pre_event_samples;
      sample_end = event_sample + post_event_samples;

      if sample_start < 1 || sample_end > size(Signals, 1),
         continue;
      end;

      % Extract epoch per signal.
      % Append it as a row to the matrix in the Epochs cell array.
      for iSignal = 1:size(Signals, 2),
         epoch = Signals(sample_start:sample_end, iSignal);
         Epochs{iEventType, iSignal} = [Epochs{iEventType, iSignal}; epoch(:)'];
      end;

   end;
end;
```

After the loop over event types, the matrices in the Epochs cell array have been filled. We now calculate the average waveform over each set of epochs. That is, we calculate the average over events for each separate time point of the epochs. These averages then form a new waveform.

```
% Make the waveform matrix: each column will be an average timecourse over

% epochs

Waveforms = [];

for iEventType = 1:2,

   for iSignal = 1:size(Signals, 2),

      mean_waveform = mean(Epochs{iEventType, iSignal});

      Waveforms = [Waveforms mean_waveform(:)];

   end;
```

```
end;
```

Data checks

Now we can plot the average waveforms! We'll make separate plots for the two events, and overlay the two signals per event, since that will show us whether we've successfully reconstructed the patterns we placed in the data.

```
function plot_waveforms(Waveforms)


figure;
column_index = 1;
for iEventType = 1:2,
   for iSignal = 1:2,
      subplot(2, 1, iEventType);
      if iSignal == 1,
         plot(Waveforms(:, column_index), 'k');
      else
         plot(Waveforms(:, column_index), 'r');
         legend({'Signal 1', 'Signal 2'});
      end;
      hold on;
      column_index = column_index + 1;
   end;
   title(['Event type ' num2str(iEventType)]);
end;
```

And we can see that, as we wanted, for event type 1 the waveform is greater for signal 1, and for event type 2 the waveform is delayed for signal 2.

At this point, the data are ready for whatever statistical analyses will be applied in the next step.

Abstracting waveforms to trial-to-trial scores

In the previous section we extracted waveforms. This is a necessary step to visualize your data and know what you're doing, but sometimes you also want to reduce each trial's waveform to a single value. This allows, for instance, tests of trial-to-trial covariation between signals on those scores, or easier analyses of time courses, or the transformation to a substantively more informative measure.

We'll adjust the signal simulation from the previous section so we have three signals, two of which contain event-locked waveforms with the same random scaling factor per trial. Save the *sim_physio* m-file as *sim_physio_2*, and make the following minor changes to first few lines:

```
function phys_sim_2

base_filename = 'sim_physio_2';

%% Header information

% Parameters
Fs = 100; % Sampling frequency
T = 1200; % Time of measurement
nSamples = floor(T * Fs); % Number of samples (or time points).
nSignals = 3;
```

The T(ime) is a lot longer, since we're going to be looking at correlations, which requires a lot more trials. I recommend playing around with this parameter later to get a feel for that. We also need to make sure waveforms don't overlap now, because we need to make sure our effects only influence the specific trials we expect them to; so we'll give them some more separation:

```
% Parameters for event timing
min_event_interval = 4;
max_event_interval = 6;
```

This is also something to play with: if you do let events overlap, then you'll see that the association we simulate to be related only to event type 1 will also cause an association for event type 2.

Then go to the code for creating the waveforms for Event Type 1, and adjust it as follows: we want signals 1 and 3 to have waveforms with covarying sizes.

```
% Create waveforms for event type 1: bigger in signal 1
find_events = find(Markers(:, 1) == 1);
for iEvent = 1:length(find_events),
    this_sample = Markers(find_events(iEvent), 2);

    sample_start = this_sample;
    sample_end = sample_start + length(basic_waveform) - 1; % Note that if the waveform was 1
sample long, you would add zero to get the element of its end-point relative to starting point.
```

```
    if sample_end > nSamples,
       continue;
    end;

    random_scaler = 3 * rand();
    Signals(sample_start:sample_end, 1) = random_scaler * basic_waveform;
    Signals(sample_start:sample_end, 2) = basic_waveform;
    Signals(sample_start:sample_end, 3) = random_scaler * basic_waveform;
end;
```

We don't signal 3 to have a waveform for Event Type 2, so we can stop here. Run the file using *phys_sim_2* to generate the new simulated continuous-measurement data.

We don't need to change the data-reading functions: they were generalizable enough to deal with an extra signal. Open *get_epochs* and save it as *get_epochs_2*. In this new version, we're going to calculate a trial-to-trial score and return it, in addition to the waveforms. We change the function definition as follows to do this:

```
function [Waveforms, Trial2Trial] = get_epochs_2(Signals, M, H)
```

Trial2Trial will contain a matrix of size (number of events, number of signals + 1). The first column will contain the event type. The other columns will contain each signal's trial-to-trial single-value score.

Where the loop over event types begins, initialize Trial2Trial to be empty:

```
% Loop over event types and all events per type
Trial2Trial = [];
```

Before the loop over signals, prepare an empty vector to hold the score per signal for the current epoch:

```
% Extract epoch per signal.
% Append it as a row to the matrix in the Epochs cell array.
trial_scores_per_signal = [];
for iSignal = 1:size(Signals, 2),
```

Find where in the code the epoch is extracted; add a function call using that epoch, which will return the score for that epoch. Add that score to the vector collecting the scores per signal.

```
for iSignal = 1:size(Signals, 2),
        epoch = Signals(sample_start:sample_end, iSignal);
        Epochs{iSignal, iEventType} = [Epochs{iSignal, iEventType}; epoch(:)'];

        this_score = get_epoch_value(epoch);
        trial_scores_per_signal = [trial_scores_per_signal this_score];
end;
```

At the bottom of the m-file, we add the subfunction get_epoch_value:

```
function this_score = get_epoch_value(epoch)
this_score = max(epoch);
```

We're keeping it very simple: just get the maximum value over the epoch. Finally, we add the row to Trial2Trial, at the end of the loop over events:

```
for iEvent = 1:length(find_events),

        ...

   Trial2Trial = [Trial2Trial; iEventType trial_scores_per_signal];
end;
```

Now load the new simulated data and run get_epochs_2:

```
[H, M, Signals] = read_data('sim_physio_2');

[Waveforms, Trial2Trial] = get_epochs_2(Signals, M, H);
```

We want to check the waveforms - we always want to check the waveforms, because we're decent people and don't smell of stale skin flakes - but the old plot_waveforms had hard-coding in it so it won't work. We could make a new file and correct the number of signals by hand, but let's make it better. Make a new m-file *plot_waveforms_2.m*, starting with this code:

```
function plot_waveforms_2(Waveforms, nSignals, nEventTypes)

figure;
column_index = 1;
for iEventType = 1:nEventTypes,
   for iSignal = 1:nSignals,
      subplot(nEventTypes, 1, iEventType);
   end;
   title(['Event type ' num2str(iEventType)]);
end;
```

Now we need to plot a general number of signals, although there's a limit to what's realistic. We'll cycle through a list of colors; add this after the subplot line, in the loop over signals:

```
colorlist = {'k', 'r', 'g', 'b', 'y'};
colorIndex = 1 + mod(iSignal - 1, length(colorlist));
plot(Waveforms(:, column_index), colorlist{colorIndex});
```

Note how we're using the modulo function to loop back over indices, in case we have a lot of signals to plt and would otherwise run out of colors / go out of the bounds of the colorlist cell array.

We'll also generate the line legends. Create an empty cell array at the start of the loop over events:

```
for iEventType = 1:nEventTypes,
        legend_string = {};
```

After the line that *plot*s the signal, add this to append a new label to the legend_string:

```
legend_string{length(legend_string) + 1} = ['Signal ' num2str(iSignal)];
```

Following that line, use the full legend_string to plot a legend if the current signal is the final one:

```
if iSignal == nSignals,
        legend(legend_string);
end;
```

Run *plot_waveforms_2(Waveforms, H.NSignals, 2)* to check it out. Hopefully all looks as it should given the simulation.

Look at the Trial2Trial matrix. Given the simulation, we expect a correlation between the scores for signals 1 and 3, for event type 1. Make an m-file called check_trial2trial.m containing this code:

```
function check_trial2trial(Trial2Trial)

eventTypeVec = Trial2Trial(:, 1);
uniqueEvents = unique(eventTypeVec);

for iEventType = 1:length(uniqueEvents),
   thisEventType = uniqueEvents(iEventType);
   f = find(eventTypeVec == thisEventType);
   subMatrix = Trial2Trial(f, 2:end);
   corrmat = corr(subMatrix);

   fprintf(['Event type ' num2str(iEventType) '\n']);
   disp(corrmat);
end;
```

The use of *unique* is overkill here, since the values are simply 1, 2, but this would work even for non-sequential event label values. Run this and note that, indeed for event type 1 only, signals 1 and 3 are correlated. Note, although this goes beyond the scope of this book, that in real data time-on-task confounds are a potential problem: if, for instance, amplitudes simply decrease over time, that would cause an "association" between signals. This could be handled via high-pass filtering or considering trial scores corrected for neighbouring trials.

Wrangling data files via shell commands

Matlab can also help with batching shell commands for manipulating data files. The nice thing is that you can flexibly generate the string containing the command to be sent to the operating system.

For instance, one common task is copying and perhaps renaming sets of files. If you've worked through the book, you'll have a directory with simulated data files called sim_beh_data1.txt, sim_beh_data2.txt, etc. Under Linux, you could copy all files to a subdirectory as follows, in the directory containing the files:

```
!mkdir tmpdir

dirStruct = dir('sim_beh_subj*.txt');

for n = 1:length(dirStruct),

    command_string = ['cp ' dirStruct(n).name ' tmp2/newfile' num2str(n) ];

    dos(command_string);

end;
```

The exclamation mark preceding !mkdir tells Matlab to send the command after the ! to the operating system. The dos command also sends its argument to operating system, just as if you'd open a terminal window and entered that string yourself. Being able to flexibly generate commands provides a lot of power.

An easy mistake to make is to use *dir* to get filenames, but try to copy them without specifying the directory they're in (which isn't saved in the .name structure).

Another common problem is deleting all files with a given name from all subdirectories, and subdirectories of subdirectories, of a starting directory. This can be done using terminal commands as follows, for the typical example of deleting SPM.mat before running an SPM batch and hence avoiding constantly needing to confirm overwriting data.

```
find -name SPM.mat | xargs rm
```

This finds all files named SPM.mat, and sends them to the remove command *rm*. Running the first part of the command, before the pipe operator |, shows you the text output of the find function. Each of those lines will be sent to *rm*. Obviously something to take care with.

Conclusion

Hopefully this book has provided a sufficient introduction to and framework for data wrangling for you to get to grips with real-life data. A potentially huge advantage of having these skills is that you can plug in and try out more substantive analysis methods, quickly and as they come to mind and in an iterative process that you would never achieve by having to depend on other people to do your data processing for you. One risk is, perhaps inadvertent, *p*-hacking. If you try out 100 methods that can all generate

different results, random chance will probably get you some false positives, and creativity will provide you with a story that makes them totally logical, ad hoc. So go forth and wrangle your data mercilessly, but use protection: either by using independent probe and replication data sets, or by reporting honestly how many variants were tested before you arrived at your results, and how robust they were to multiple testing via multiple methods.

A webpage for this book is available on http://www.tegladwin.com/books.php.