14:440:127– Introduction to Computers for Engineers

Notes for Lecture 12

Rutgers University, Spring 2010 Instructor- Blase E. Ur

1 Pattern Matching- Regular Expressions

(This section is not covered in your book. You can refer to
`www.mathworks.com/access/helpdesk/help/techdoc/matlab_prog/f0-42649.html`)
Regular Expressions are a class of tools that allow you to do pattern-matching (identifying strings of letters or numbers that match a certain pattern). Some of the most powerful tools for regular expressions are in the Perl programming language; you also might encounter people writing regular expressions with the Unix command *grep*, which uses regular expressions to find files on a system. Matlab also allows you to use regular expressions with the following series of functions:

- *regexp* matches a pattern (case sensitive)
- *regexpi* matches a pattern (case insensitive i.e. *A* and *a* are the same)
- *regexprep* replaces a pattern with something else

1.1 Matching the most basic patterns

The arguments to *regexp* are 1) a string in which you're searching for matches, and 2) a pattern (also given as a string). In the most basic case, let's find where *cat* is located in the string *"the cat in the hat"*:

```
mystring = 'the cat in the hat';
regexp(mystring,'cat')
     ans =     5
```

This result tells us that the pattern 'cat' begins with the 5th character of the string.

You could instead call *regexp* or *regexpi* as follows, requesting multiple outputs (and specifying what they are):
`[mat ix1 ix2] = regexp(pstr, expr, 'match', 'start', 'end')`- *pstr* is your string, and *expr* is your regular expression. *mat* will be a cell array of the matches themselves, *start* will be a vector of the starting points of the matches, and *end* will be a vector of the ending points of the matches. If you just wanted the matches you could simply say `regexp(str,regexp,'match')`:

```
mystring = 'the cat in the hat';
regexp(mystring,'cat','match')
     ans =  'cat'

[a b c] = regexp(mystring,'cat','match','start','end')
     a = 'cat'
     b = 5
     c = 7
```

1.2 Matching Symbols

Of course, it's not all that useful to only match words you can identify already. Thus, Matlab has a number of special symbols you can use for creating patterns. Note that whitespace means empty spaces, the characters that represent tabs, or new line characters, etc.

```
.  matches any single character, including white space
[abc] matches ANY single one of the characters in [ ]
[a-z] matches ANY single character in that range (a,b,c,d...,x,y,z)
[^abc] matches any single character NOT contained in [ ]
\s  matches any white-space character: [ \f\n\r\t\v]
\S  matches any non-whitespace character: [^ \f\n\r\t\v]
\w  matches any single alphanumeric/underscore character: [a-zA-Z_0-9]
\W  matches any character that's not alphanumeric or an underscore
\d  matches any numeric digit: [0-9]
\D  matches any non-numeric character
```

Let's look at an example. Let's find all words that contain a letter, an *a*, and then another letter:

```
pstr = 'FAT CAT SANDWICH, MA.';
expr = '\wa\w';
[mat ix1 ix2] = regexpi(pstr, expr, 'match', 'start', 'end') % note regexpi
   mat =  'FAT'    'CAT'    'SAN'
   ix1 =  1     5     9
   ix2 =  3     7    11
```

Note that each `\w` matched exactly one letter, no more, no less. Therefore, only the "san" in "sandwich" was matched. Similarly, "MA" was not a match since there's no letter following the A.

1.3 Grouping

However, Matlab lets you specify that some characters should be repeated. For instance, if you wanted to instead match 5 B's in a row, you'd have to group them (using parentheses) `(B){5}` . The `{5}` in squigly braces means it must match exactly 5. You can instead have a range of numbers inside the squigly braces, i.e. `(B){3,6}` , which matches between 3 and 6 B's.

To instead match an unlimited number of repetitions, you can use either * or +. The star * matches 0 or more occurrences of the grouping, whereas the plus + matches 1 or more occurrences of the grouping.

```
quote = 'The cat in the hat sat on Saturday';
regexpi(quote, '\wat\w*', 'match') % case insensitive
   ans =  'cat'    'hat'    'sat'    'Saturday'

regexpi(quote, '[cs]at\w*', 'match') % case insensitive
   ans = 'cat'    'sat'    'Saturday'
```

Note in the first example `\wat\w*` I look for a letter, followed by an a, followed by a t, followed by 0 or more other lettters. Therefore, three letter words and longer words both match this pattern. In the second example, note that [cs] matches a *c* or an *s*.

In quote1 below, let's say I wanted to match either "I'm" or "It's." The pattern here is that we start with an I, then perhaps another letter, then we have an apostrophe, and then another letter:

```
quote1 = 'I''m gonna make this pencil disappear...Ta-daa! It''s... it''s gone.';
regexpi(quote1, 'I\w*''\w', 'match') % note regexpi
   ans =  'I'm'    'It's'    'it's'

regexpi(quote1, 'I.*''\w', 'match') % note regexpi
   ans =  'I'm gonna make this pencil disappear...Ta-daa! It's... it's'
```

Whoah! In the second example, we use the period (which matches any character) rather than `\w`, which matches a letter, number, or underscore. Suddenly, we have one giant match, beginning with "I'm" and ending with the second "it's." WTF?

1.4 Greedy Matching

What's happening the previous example is that Matlab performs what's considered greedy matching, which means it tries to match the longest string possible when you use the * or + operators. So, let's say you had the string '(12) (15)' and wanted to find the numbers in parentheses. If you tried to use the regular expression `'\(.*\)'`, it would match (12) (15)... the whole thing! To instead match them separately use a question mark `?` to stop greedy matching i.e. `'\(.*?\)'` will match (12) and also (15). Thus, to fix the above example, we can do:

```
quote1 = 'I''m gonna make this pencil disappear...Ta-daa! It''s... it''s gone.';

regexpi(quote1, 'I.*?''\w', 'match') % note regexpi
   ans =    'I'm'    [1x34 char]    'it's'
ans{2}
   ans = 'is pencil disappear...Ta-daa! It's'

% Oops, didn't work... let's try:
regexpi(quote1, '(^I.*?''\w)|(\sI.*?''\w)', 'match') % note regexpi
   ans = 'I'm'    ' It's'    ' it's'
```

Ok, what's going on in these examples? As you can see, regular expressions get really complex really quickly. In the first, incorrect, example above, we start matching at an *i* in the middle of the string. We keep going with our match until we no longer fit the pattern. Unfortunately, we keep fitting the pattern for *is pencil disappear...Ta-daa! It's* because we have an *I*, zero or more (any) characters in between, an apostrophe, and then a word character.

In the second example, I try to match the pattern only when the I begins a word. I assume that a word is preceded by a space... OR starts off the string. Here, I introduce two new concepts:

- The pattern `^EXPR` matches EXPR only when it begins the string. Similarly, the pattern `EXPR$` matches EXPR only when it ends the string.
- You can have separate groups of patterns separated by the "or symbol": `|`. If one OR the other pattern is matched, we have a match.

Also note that something funny happens when you try to match "special characters," such as periods, that have other meanings in the chart above:

```
quote1 = 'He\she is hardly working hard. ';

regexpi(quote1, 'He\she', 'match') %ex1
   ans =    { }

regexpi(quote1, 'He\\she', 'match') %ex2
   ans =    'He\she'

regexpi(quote1, 'hard.', 'match') %ex3
   ans =    'hardl'   'hard.'

regexpi(quote1, 'hard\.', 'match') %ex4
   ans =    'hard.'
```

What's going on? Well, in `%ex1`, `\s` isn't a slash followed by an s, it's the special character that matches a white-space character. To fix this, use `\\` as in `%ex2`, which matches a *single* slash. Similarly, in `%ex3`, recall that a period matches ANYTHING... thus, an *l* and a period are both matched. To match a period, use `\.`.

In general, the slash escapes the "specialness" of a character. Here's a modified chart with basically every special character you need:

```
.  matches any single character, including white space
[abc] matches ANY single one of the three characters in [ ]
[a-z] matches ANY single character in that range (a,b,c,d...)
[^abc] matches any single character NOT contained in [ ]
\s  matches any white-space character: [ \f\n\r\t\v]
\S  matches any non-whitespace character: [^ \f\n\r\t\v]
\w  matches any single alphanumeric/underscore character: [a-zA-Z_0-9]
\W  matches any character that's not alphanumeric or an underscore
\d  matches any numeric digit: [0-9]
\D  matches any non-numeric character
+   matches one or more of something
*   matches zero or more of something
{5} matches exactly 5 of something

^   matches the beginning of a string
$   matches the end of a string
\<expr  matches expr when it occurs at the beginning of a word.
expr\>  matches expr when it occurs at the beginning of a word.
\<expr\> matches expr when it is the entire word

%%%%% The following characters are specified in a special manner:
\  inhibits the specialness of a character
\.  matches a period
\\  matches a backslash
\$  matches a dollar sign
\?  matches a question mark
\n  matches a new line
```

1.5 Substitutions- regexprep

You can use the function *regexprep* to substitute based on a pattern. Let's make all of our course staff as cool as "Soulja Boy Tell 'Em":

```
quote = 'I see Blase, Bo, Cyrus, and Wen. Where is Vishnu?';

regexprep(quote,'[A-Z][a-z]+','Tell ''Em')
     ans = 'I see Tell 'Em, Tell 'Em, Tell 'Em, and Tell 'Em. Tell 'Em is Tell 'Em?'

regexprep(quote,'([A-Z][a-z]+)','$1 Tell ''Em')
     ans = 'I see Blase Tell 'Em, Bo Tell 'Em, Cyrus Tell 'Em,
            and Wen Tell 'Em. Where Tell 'Em is Vishnu Tell 'Em?'

regexprep(quote,'([^\.]\s[A-Z][a-z]+)','$1 Tell ''Em')
     ans = 'I see Blase Tell 'Em, Bo Tell 'Em, Cyrus Tell 'Em,
            and Wen Tell 'Em. Where is Vishnu Tell 'Em?'
```

Notice that I do something funny in the example above. I use `$1`, which is called a back-reference. It refers to the text matched in the first set of parentheses of the pattern string. Similarly. `$2` would refer to the text matched in the second set of parentheses. In the first example, we first match "Blase," and then replace that match with "Tell 'Em." Instead, we want to replace the match with "Blase Tell 'Em," which utilizes what we matched. Therefore, we use a back-reference to reuse what we had matched.

1.6 More Regular Expression Examples

```
pstr = '(123)345-5421 (154)122-1235';
expr = '\((\d){3}\)(\d){3}-(\d){4}';
[mat ix1 ix2] = regexp(pstr, expr, 'match', 'start', 'end')
  mat =   '(123)345-5421'    '(154)122-1235'
  ix1 =    1    15
  ix2 =   13    27
```

```
y = 'denise and dennis went to denny''s to eat dinner';
regexp(y,'den','match')
   ans =  'den'    'den'    'den'
regexp(y,'den\w*','match')
   ans =  'denise'    'dennis'    'denny'
regexprep(y,'denise','blase')
   ans =
   blase and dennis went to denny's to eat dinner
regexprep(y,'den\w*','blase')
   ans =
   blase and blase went to blase's to eat dinner
regexprep(y,'den(\w*)','blase$1')
   ans =
   blaseise and blasenis went to blaseny's to eat dinner
```

```
y = 'denise and dennis went to denny''s in the den with aden to eat dinner';
regexp(y,'den\w*','match')
   ans =
    'denise'    'dennis'    'denny'    'den'    'den'
regexp(y,'\w*den\w*','match')
   ans =
    'denise'    'dennis'    'denny'    'den'    'aden'

% greedy vs non-greedy matching
y = 'blase is a greedy person, and so is bo.  no he is not.';
regexp(y,'blase.*is','match')
   ans =
    'blase is a greedy person, and so is denise.  no he is'
regexp(y,'blase.*?is','match')
   ans =
    'blase is'
```