prin - an alternative to Matlab's sprintf and fprinf functions

Version: 9-Dec-2013

There are 3 areas where the limitations of the sprintf function led me to create a more powerful alternative called prin. The calling sequence is the same as for *sprintf*, i.e. `str = prin(FormatString, OptionalArguments)`
or the same as *fprintf*, i.e. `str = prin(FileID, FormatString, OptionalArguments)`

1.) Floating point conversions

The first area is the limited flexibility in how floating point numbers are converted to strings. The %e, %f, and %g formatting options are useful under the following conditions:

1.) We are requesting a specific precision (total # of digits, or # of digits after the decimal point)
2.) Displaying more precision than requested has no value.
3.) Numbers are never printed with less than the requested precision. This means that the number of characters output may be greater than the specified field width.

These conditions sound pretty reasonable and certainly cover many (maybe even most) situations where floating point numbers must be output as text. However there are some situations where these conditions are not suitable. The most important of these is when a floating point number must be displayed inside a graphical element such as an edit box. The edit box is typically a fixed size determined by the GUI design which limits how many characters it can show. Going beyond that limit can produce an illegible display (chopping off part of the exponent for example). For another example consider the following table produced by `prin`:

```
24.9688   130.253   4.03723   55.0913
1.01582   652.682   648.450   52.6309
0.12785   4.2e-11   2.62447   26.1841
1.67417   596.219   7.05377   3.48e-4
455.854   330.960   .019674   2.59148
28.4859   14.2515   25672.3   0.11407
2.94343   65.8804   4.247e7   1.17610
16.2275   96.6397   34476.4   216.765
9.37267   36.6329   0.69132   429.560
2.04876   4.80613   2.20573   1.33834
28.0669   9.17463   6219.46   6.16408
```

Table 1

For the table to remain legible (especially if the table is much larger) each number must be printed using the same number of characters. Otherwise the columns become misaligned making it difficult to tell which numbers belong to which column. Seven characters per number was chosen for this table. The typical values in this particular data set are in the range of 1 to 10,000 with occasional outliers. Note that most of the numbers have been printed using six significant digits, with progressively fewer digits being available as the numbers move farther outside the typical range.

If you were restricted to the seven character limit used above, the %g format would be nearly useless. The best you could do would be `'%7.1g'` (i.e. one significant digit) which would produce this disappointing output:

```
  2e+001    1e+002         4    6e+001
       1    7e+002    6e+002    5e+001
     0.1    4e-011         3    3e+001
       2    6e+002         7    0.0003
  5e+002    3e+002      0.02         3
  3e+001    1e+001    3e+004       0.1
       3    7e+001    4e+007         1
  2e+001    1e+002    3e+004    2e+002
       9    4e+001       0.7    4e+002
       2         5         2         1
  3e+001         9    6e+003         6
```

Table 2

Attempting to display more digits would misalign the columns unless you increased the field width. If we wanted to get the full 6 digits of precision shown for most of the numbers in table 1 we would need `'%13.6g'` but this requires nearly twice as many characters per digit, possibly taking up more space than we have. What usually happens is we make some sort of compromise between space and precision. For instance if we use a `'%11.4g'` format (allocating 4 extra characters per number compared to the table 1) we get:

```
      24.97         130.3         4.037         55.09
      1.016         652.7         648.5         52.63
     0.1279    4.238e-011         2.624         26.18
      1.674         596.2         7.054    0.0003478
      455.9           331       0.01967         2.591
      28.49         14.25    2.567e+004        0.1141
      2.943         65.88    4.247e+007         1.176
      16.23         96.64    3.448e+004         216.8
      9.373         36.63        0.6913         429.6
      2.049         4.806         2.206         1.338
      28.07         9.175          6219         6.164
```

Table 3

A readable result, although in many respects less pleasing than table 1. Besides taking far more space, sometimes the e format is used when the f format would be more pleasing (e.g. 3.448e004 vs 34476.42 in row 8 above) and sometimes the f format is used when the e format would be more pleasing (e.g. 0.0003478 vs 3.478e-004 in row 4). Although Matlab's `sprintf` number formats have a venerable history from the c language (and even Fortran before that) I suspect most of you have at one time or another been frustrated by the tradeoffs involved in choosing a floating point format. This is just one example.

Of course `prin` allows all the format specifiers allowed with `sprintf`, but also provides some new formats to get around the frustrations mentioned above. `W` stands for the "Width" format which turn the tables on `sprintf`. With the e,f,g formats we specify the precision and accept the width (i.e. # of characters) that `sprintf` gives us. But with the `W` format we specify the desired width and `prin` will determine the maximum precision possible while satisfying the width specification. For example, the format `'%7W'` tells `prin` to output no more than 7 characters while giving us as much precision as possible. Note that with this format, `prin` may sometimes output fewer than 7 characters. The `'%7V'` format is similar to `'%7W'` except that in this case `prin` must output exactly 7 characters … never fewer … never more! That's why the `V` format is so useful for creating straight columns of numbers and in fact the `'%7V'` format was used to create table 1 above.

`prin` also includes two more formats called `v` and `w` which differ from their upper case counterparts in only one respect. For the lower case formats, `prin` does NOT count a decimal point when determining field width. For example `'%7v'` will output 7 characters if a decimal point is not needed, but will output 8 characters if a decimal point is one of them. This strange way of counting gives a slight advantage when the characters are displayed using a proportional width font (quite common for GUI objects). In those fonts the decimal point is quite narrow compared to the other characters in the output and we can take advantage of this to achieve (marginally) higher precision outputs. In fact the `w` format is usually the best choice for all text carrying graphical objects.

2.) Repeat counts

Especially for those of you who remember the repeat counts allowed in Fortran formatting commands it may be particularly annoying to write repetitive `sprintf` formats such as:

```
sprintf('%d)%7.4gK ->%7.4gK ->%7.4gK ->%7.4gK -> =%s',8,sqrt(3:6),'test')
```

which would return the string:

```
8)  1.732K ->      2K ->  2.236K ->  2.449K -> =test
```

Of course we could write the format string this way (although I don't think this is much of an improvement):

```
sprintf(['%d)' repmat('%7.4gK ->',1,4) ' =%s'],8,sqrt(3:6),'test')
```

`prin` makes this easier by providing "*numbered repeats*" allowing us to create the same output string this way:

```
prin('%d)4{%7.4gK ->} =%s',8,sqrt(3:6),'test')
```

The ! character has a special meaning inside a numbered repeat. It terminates the output early on the last iteration. (This feature may seem arcane at first, but you may be surprised how often the need for it arises). For example suppose we modified the previous command by adding the ! character inside the braces as follows:

```
prin('%d)4{%7.4gK !->} =%s',8,sqrt(3:6),'test')
```

The output from this command is as follows (note that it is the same as before except the last "->" is missing):

```
8)  1.732K ->      2K ->  2.236K ->  2.449K  =test
```

Repeat counts must be one or two digits long. A repeat count of zero has the effect of commenting out a formatting section. There are no restrictions on what can be inside the numbered repeat. It may include any number of (%) formatting codes as well as no formatting codes at all. A numbered repeat may even contain other numbered repeats and in fact you may nest them as deeply as you need. Numbered repeats may also contain vector formats which are described next.

`prin` supports an additional type of repeating expression (also indicated by braces) called a "*vector format*". If the character in front of the left brace is a decimal digit, the braces are interpreted as a *numbered repeat* and if not a decimal digit it is interpreted as a *vector format*. The vector format must abide by one additional restriction – there must be exactly one (%) formatting code inside the vector format. If this condition is not satisfied, the left and right braces are treated as ordinary characters and are passed to the output stream without any special processing. The braces are also treated as ordinary characters if the left brace is proceeded by either a caret character (^) or an underscore (_) character even if the expression in braces otherwise meets the restrictions of a vector format. This exception is useful because TeX processing uses braces to enclose a group of sub-scripted (_) or super-scripted (^) characters. There is also a brace escape sequence (see the section on escape sequences near the end of this document).

The vector format, when paired with a vector argument, will be used repeatedly for every element in a vector argument, as demonstrated by the following example:

`prin('Day %d: {The %dth trial  }--%s',22,5:8,'Critical')` will produce the string:

```
Day 22: The 5th trial  The 6th trial  The 7th trial  The 8th trial  --Critical
```

A numbered repeat could also be used in the above example, but the vector format is better since we don't need to know the length of the vector in advance .

If a two dimensional array argument is used, the elements will be processed column-wise (i.e. if A is a matrix, then supplying A in the argument list is equivalent to supplying A(:) in the argument list).

The ! character has the same meaning mentioned above, although here you may want to think of it as the beginning of a delimiter to be inserted between vector elements. For example:

`prin('{Line%dA -- }?',1:4)` outputs: `Line1A -- Line2A -- Line3A -- Line4A -- ?`

whereas

`prin('{Line%dA! -- }?',1:4)` outputs: `Line1A -- Line2A -- Line3A -- Line4A?`

As mentioned above, a numbered repeat may have both vector formats and other numbered repeats inside it. However the opposite is not true. The vector format may not have a numbered repeat or another vector format inside it.

3.) Cell-array delimiters

The third limitation of `sprintf` addressed by `prin` is that `sprintf` does not have the ability to output cell arrays of strings, requiring you to use awkward loops to create such objects. Cell arrays of strings have many uses – for example to set the string property of graphical object collections. `prin` can generate these cell arrays directly by interpreting the four character string `' ~, '` as a cell array row delimiter and the string `' ~; '` as a column delimiter. This is illustrated with the following examples:

```
prin('aaa ~, bbb ~, ccc')              example 1
ans =   'aaa'    'bbb'    'ccc'

prin('aaa ~, bbb ~; ccc ~, ddd')       example 2
ans = 'aaa'    'bbb'
      'ccc'    'ddd'

prin('{Line %d! ~; }',3:6)             example 3
ans = 'Line 3'
      'Line 4'
      'Line 5'
      'Line 6'
```

Since most cell arrays generated by `prin` use the ! delimiter construction of example 3 above, an alternate and somewhat less cryptic method is available using the `col` delimiter. The following line produces the same output as example 3:

```
prin('{Line %d!col}',3:6)
```

And likewise, the transpose of this result is output using the `row` delimiter:

```
prin('{Line %d!row}',3:6)
```

Escape sequences

As with `sprintf`, to pass a % character to the output you must use %% since otherwise it will be treated as a format code. (Only a single % is passed to the output stream). In addition to the `sprintf` escape sequences (`\n,\r,\t,\b,\f,\\`) which pass linefeed, carriage return, tab, backspace, formfeed, and backslash characters respectively, `prin` also supports the `\{` and `\}` escape sequences which pass a brace to the output stream without interpreting it as a numbered repeat or a vector format. If you need to include the ! character inside a numbered repeat or vector format without it being interpreted as a delimiter, use its octal ascii code escape sequence (`\41`). In the unlikely event you need to pass one of the length 4 strings `' ~, '` or `' ~; '` without them being interpreted as cell array delimiters, replace the tilde with its octal ascii code escape sequence (`\176`).

str = Pftoa(format string, number)

`Pftoa` is a subroutine used internally by `prin` to implement the new `V`,`W`,`v`,`w` formats. Normally you won't call `Pftoa` directly since calling it thru `prin` is more general and just as concise. However calling `Pftoa` with just a single argument will create a test file which you may find helpful in understanding the new floating point formats.

str = prin(FID, FormatString, OptionalArguments)

`prin` may contain an additional input argument (`FID`) that specifies the file identifier, in which case it behaves like `fprintf` in that the generated string is written to the specified file. (The generated string is also returned in `str`.) Normally `FID` will be the number returned from an `fopen` command, although as with `fprintf` using 1 or 2 as the `FID` refers to the standard output device and standard error device respectively. (Both of those devices are redirected to the Matlab command window and are distinguished by the text color). An example of the use of the `FID` parameter follows:

example 4

```
fid = fopen('file1.txt','wt');
prin(fid,'15{-} Line %d 15{-}\n',35:39);
fclose(fid);
```

which will create a file called file1.txt containing the following text:

```
--------------- Line 35 ---------------
--------------- Line 36 ---------------
--------------- Line 37 ---------------
--------------- Line 38 ---------------
--------------- Line 39 ---------------
```

Note that the same file would be created if you replaced `prin` with `fprintf` and if you replaced the two numbered repeats in the format string with 15 dashes. Since 3 line sequences similar to example 4 are common, `prin` provides a way to write this in one line:

```
prin(-'file1.txt','15{-} Line %d 15{-}\n',35:39);
```

The leading minus sign in front of the file name tells `prin` to use the `'wt'` permission in the file open, which means the file is flushed before the write operation begins. (For a full discussion of the file permissions, type `help fopen`). If you don't want the file to be flushed first (i.e. append mode) use a plus sign in place of the minus sign:

```
prin(+'file1.txt','15{-} Line %d 15{-}\n',35:39);
```

This would be the same as if you changed the `'wt'` in example 4 to `'at'`. (Note that the letter `'t'` in the permission signifies text mode and is ignored in all operating systems except Microsoft Windows.) If you don't want the `file1.txt` to be created in the current directory, the file name in quotes may include the full path.

Actually the calling sequence is more general that what is indicated above in that `prin` allows you to specify multiple file names in the argument list. Consider the following example:

```
fid = fopen('file1.txt','wt');
prin(1,2,fid,-'file2.txt',+'../file3.txt','15{-} Line %d 15{-}\n',35:39);
```

This will write the same 5 lines shown above (Line 35 to Line 39) to the standard output and standard error devices as well as three different files. file3.txt will be written in append mode to the parent of the current folder while file1.txt and file2.txt will be written to the current folder and will be flushed before the operation. file1.txt will remain open while `prin` closes the other two files. You might guess that the plus sign in front of `'../file3.txt'` in the above example can be omitted, but this is not correct. The reason is that `prin` assumes that the first character string in the argument list is the format string, so without the plus sign, the file name would become the format string.

I hope using `prin` enhances your Matlab experience. Please let me know about any problems, questions, or suggestions you have relating to `prin`. You can reach me at paul@mennen.org