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

Version: 01-Jan-2017

To view this document type `prin` (i.e. with no arguments) at the Matlab command prompt.

Introduction

Matlab's *sprintf* and *fprintf* functions behave almost exactly like the identically named functions from the standard C library. The advantage of these two formatting functions is that they are familiar to most programmers. They have been widely used since they were created in the late 1960's with roots from Fortran a decade earlier. However these functions are somewhat mismatched to Matlab since it is a higher level language designed for rapid prototyping compared to the lower level system programming C language.

(Because *sprintf* and *fprintf* use the same output formatting rules, for brevity in what follows only the function *sprintf* will be mentioned). Here are the *sprintf* shortcomings that inspired me to create *prin*, a more powerful output formatting function for Matlab:

- The floating point conversions employed by *sprintf* (%f, %e, %g) allow one to request a specific precision (number of digits in total, or number of digits after the decimal point) as well as a field width (length of the output character string). Often there is a disparity between the precision and the field width, which *sprintf* resolves by giving priority to the precision. This means conversions never output more precision than requested even if that is possible given the field width. Likewise the conversions never produce less than the requested precision even if the output must exceed the field width. This convention is fine for systems programming and even for most scientific programming; However it can be frustrating to use for GUI programming where one often has graphic objects that can accommodate a limited number of characters. Another situation where these conversions are problematic is when one is creating a numerical table that must have aligned fixed-width columns. In these situations we need to give priority to the field width, which is not possible with *sprintf*.

- Complex numbers are used extensively in nearly all scientific programming, which explains why Matlab's most primitive data type is the complex number or complex array. Yet when one passes a complex number to *sprintf*, the imaginary part is ignored − a behavior that is rarely desired.

- Matlab programmers often view vectorization as the most important programming style to increase Matlab program efficiency, compactness, readability, and development speed. However the lack of repeat counts or other vector operators in *sprintf* format strings inhibits vectorization.

- Matlab programming makes heavy use of its native cell array data type, which is not native to the C language and is not supported by *sprintf*. Cell arrays of strings are used for some Matlab graphical objects and for nearly every collection of graphical objects, but creating these strings using *sprintf* is cumbersome. *sprintf* also does not allow a cell array as an input argument.

- Often one wants to save a collection of numbers or strings contained in an array or cell array to a file in human readable form (i.e. a text file), and normally this requires separate statements for opening the file (*fopen*), saving the contents and finally closing the file (*fclose*). In complex situations these individual steps may be warranted, but often this amounts to needless extra work. It would be preferable to have an I/O interface that allowed all three of these steps to be completed in a single statement.

In the remainder of this document I describe how I address each of these concerns with the *prin* command which is intended as a replacement for both *sprintf* and *fprintf*. My goal is to improve the productivity of Matlab programmers by simplifying the most common output formatting tasks. I have been using *prin* for long enough now that I think I have arrived at something that should be part of the Matlab language. If the Mathworks shares this view, let me know. I would be pleased to transfer the source code copyright to the Mathworks if they were interested in making *prin* a part of their standard Matlab release.

Calling sequence

prin is called just like *sprintf, i.e.:* `str = prin('FormatString',OptionalArguments);`
OR the same as *fprintf:* `str = prin(FileID,'FormatString',OptionalArguments);`

Floating point conversions

When a floating point number must be displayed inside a graphical element such as an edit box, the GUI layout limits how many characters it can contain. Attempting to go beyond that limit can produce an illegible display (chopping off part of the exponent for example). The concept of fixed width output is demonstrated first by considering the following table produced by *prin*:

```
  24.9688    6.02e23    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

You may not actually want to format a table like this, but the values in this table are destined to appear at various times in a small GUI edit box that has room for only 7 characters. This table shows how *prin* could format this data to fit inside the edit box. Note that the typical values in this particular data set are in the range from 1 to 10,000 but the data set a large dynamic range because of occasional outliers. It's this large dynamic range which makes it nearly impossible to use the conventional *sprintf* formatting strings for such a task. For instance, let's try to use the `'%g'` format for this. With some experimentation, you will find that the best you can do with this format is to print only one significant digit, (i.e. `'%7.1g'`) since otherwise the output would sometimes exceed 7 characters and the most important part of the exponent would get chopped off. Using the `'%7.1g'` format on the data that produced table 1 above would produce table 2 below (a very disappointing result).

```
2e+001   6e+023        4    6e+00
     1   7e+002   6e+002    5e+00
   0.1   4e-011        3    3e+00
     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

```
  24.97    6.022e23       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

Suppose in an attempt o get a reasonable precision we accept a smaller font size and expand the edit box to 11 characters instead of just 7. So now we can try the `'%11.4g'` format, which results in table 3 shown above.

Although all the numbers are readable, that is about the best you can say for it. Despite the extra space we have allotted it still rarely shows as much precision as the original 7 character display from *prin* in table 1. Even worse is that sometimes the result is down right ugly. For example the `3.448e+004` in row 8 could be displayed more readably and more precisely without exponential notation (i.e. `34476.42`) and the `0.0003478` in row 4 would be

easier to interpret in exponential notation (i.e. `3.478e-4`). Of course you could try using a mixture of %f, %e, and %g formats but even after hours of experimentation you will likely remain unsatisfied with the results. The *sprintf* formats are simply unsuitable for this task. If you have used these formats a lot, you probably have often been frustrated with the tradeoffs involved in other situations as well. A main goal of *prin* is to remove this source of frustration.

prin accepts all the *sprintf* format specifiers along with 4 new ones to get around these limitations. The first of these is %W (stands for the "Width" format) which strengthens the field width specification of the familiar %e, %f, %g formats. With the e,f,g formats we specify the precision and accept whatever field width (i.e. # of characters) that *sprintf* produces. But with the %W format we do it the other way around. We specify the desired width and *prin* determines 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* outputs exactly 7 characters … never fewer … never more! That's why the %V format is so useful for creating straight columns of numbers. (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 appears in the output. This may seem a strange way to count, but it is an advantage when the characters are displayed in a proportional width font (common for GUI objects). In those fonts the decimal point is quite narrow compared to the other characters and we can take advantage of this to output somewhat better precision. In fact the %w format is usually the best choice for displaying numbers in a graphical object.

There are two optional modifiers (+/-) for the %W format that allow the output to be padded with blanks. This applies only to the situations when the %W format outputs fewer characters than the field width (which doesn't happen with %V). This example demonstrates the three ways you can pad the output to the desired number of characters:

```
prin(' [%5W] [%-5W] [%+5W] [%5V] ',37,37,37,37)
```

```
ans = [37] [37···] [···37] [37.00]
```

(note: The red dots represent spaces)

As you can see, the first format used above (`%5W`) outputs only two characters despite the 5 character field width specification. The remaining three format specifications, demonstrate the three ways to expand this to 5 characters:

`%-5W` Pad on right with blanks
`%+5W` Pad on left with blanks
`%5V` Pad with non blank characters

The +/- modifiers may be used with the %w format as well, although I'm not sure that is very useful. With the %V format, the precision is increased until the width specification is satisfied, so generally there is no need to pad the output with blanks. However there are four special values with fixed precision (zero, NaN, Inf, -Inf) where padding with blanks is appropriate, which means the +/- modifiers only effect the output for those four values as follows:

	'[%6V]'	'[%-6V]'	'[%+6V]'
0	[0]	[0]	[0.0000]
NaN	[NaN]	[NaN]	[NaN]
Inf	[Inf]	[Inf]	[Inf]
-Inf	[-Inf]	[-Inf]	[-Inf]

One final point to mention with the new floating point formats is that the field width is optional. If omitted, the default field width of 7 is used. So for example %v and %7v are equivalent, as are %+W and %+7W.

Formating complex numbers

In *sprintf* all the number conversions ignore any imaginary component. By default *prin* does the same thing, however with *prin* you may use the j or J modifier to convert complex quantities. (At least for engineers, the letter "j" should be easy to remember since they typically use that letter to represent $\sqrt{-1}$). The upper case modifier (J) will always print both the real and imaginary parts of the number even if one or both of them are zero, whereas the lower case modifier (j) will print only the nonzero components. The separator used between the real and imaginary parts is either a plus or a minus sign surrounded by a single space on both sides.

The "k" or "K" modifiers behave similarly except that the spaces in the separator mentioned above are not included. (You may want to remember this by assuming that the K modifiers are more "Kompact".)

You may use the J/K modifiers with any numeric conversion code (even the integer %d format). The following table should make the behavior of these modifiers more clear. In this table, we have assumed that:

```
z = [3.2+4.56i, 3-4i, 6, -2i, 2i, 0];
```

prin('%J6.3f\n',z)	prin('%K4W\n',z)	prin('%k4W\n',z)	prin('%j4W\n',z)	prin('%J+4W\n',z)
3.200 + 4.560i	3.2+4.56i	3.2+4.56i	3.20 + 4.56i	3.2 + 4.56i
3.000 + 4.560i	3-4i	3-4i	3 - 4i	3 - 4i
6.000 + 0.000i	6+0i	6	6	6 + 0i
0.000 - 2.000i	0-2i	-2i	-2i	0 - 2i
0.000 + 2.000i	0+2i	2i	2i	0 + 2i
0.000 + 0.000i	0+0i	0	0	0 + 0i

When using a J/K modifier and a `+/-` modifier at the same time, the J/K modifier must come first (as shown in the last column of this example).

Repeating format blocks

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('%s: %6.4gK -> %6.4gK -> %6.4gK -> %s','TestB',pi,9/7,8/7,'complete')
ans =
    TestB:   3.142K ->  1.286K ->  1.143K -> complete
```

Annoyingly, we needed to repeat the same conversion code and delimiters three times. If we really wanted to avoid repeating that portion of the format string, we could use *repmat* to do the repeat for us (although I don't think this is much of an improvement):

```
sprintf(['%s:' repmat(' %6.4gK ->',1,3) ' %s'],'TestB',pi,9/7,8/7,'complete')
```

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

```
prin('%s: 3{ %6.4gK ->} %s','TestB',pi,9/7,8/7,'complete')
```

The number in front of the brace is called a *repeat count* and 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 a *numbered repeat*. It may include any number of formatting codes (%), or even 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.

Note that in the *prin* command above, the "3" in front of the brace was needed because otherwise *prin* wouldn't know how many values from the argument list it should apply to the repeat block. However if the values are contained in an array instead then we don't need the "3" because *prin* will know to apply the repeat block to every element of the array. We call this type of repeat block a *vector format* because when paired with a vector argument, it will be used repeatedly for every element of the vector. For example, this command outputs the same string as before:

```
prin('%s: { %6.4gK ->} %s','TestB',[pi 9/7 8/7],'complete')
```

(Note that we now have a vector in the argument list which we didn't have before.) It would still work to put the "3" in front of the above command (turning the *vector format* back into a *numbered repeat*) but it is better to use the *vector format* when possible because then we don't need to change the format string when the vector changes length.

The *vector format* has two additional restrictions (neither of which are restrictions of the *numbered repeat:*

1. 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 preceded by either a backslash character (\), a caret character (^) or an underscore (_) character even if the expression in braces otherwise meets the restrictions of a vector format. The last two exception are useful because TeX processing uses braces to enclose a group of sub-scripted (_) or super-scripted (^) characters.

2. Other repeat groups (*vector formats* or *numbered repeats*) are not allowed inside a *vector format*. However as mentioned above, a *numbered repeat* may have any number of *vector formats* and other *numbered repeats* inside it, and in fact these may be nested to any depth.

The `!` character has a special meaning when it is inside a *numbered repeat* or a *vector format*. It terminates the output early on the last iteration. This may seem arcane at first, but you will soon see how often this feature is needed.

As an example consider the following *prin* statement (which includes a *vector format*) and it's result:

```
prin('{area %c! + } = %W square miles','Q'+(0:3),pi/3)

ans =
area Q + area R + area S + area T +  = 1.0472 square miles
```

Obviously this is not quite what we wanted. The plus sign serves as a delimiter between each element of the character vector, but we don't want the last plus sign. Now the usefulness of the `!` feature becomes clear:

```
prin('{area %c! + } = %W square miles','Q'+(0:3),pi/3)

ans =
area Q + area R + area S + area T = 1.0472 square miles
```

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).

Cell array support

sprintf does not support cell arrays as input arguments, nor can it produce cell arrays as output. *prin* does not suffer either of those deficits.

When *prin* encounters a cell array as one of its optional arguments, it sequences thru the elements of the cell array columnwise as if each element of the cell array were in the argument list individually. The cell arrays may even be nested. For example, these two *prin* commands do the same thing:

```
prin('fmtString',a,{{b c} d; e {f;g}},h)
prin('fmtString',a,b,c,e,d,f,g,h)
```

If the order of the variables in the second line was a surprise to you, remember that a cell array is processed column-wise.

The ability of *prin* to output cell array's of strings is even more important since such cell arrays have many uses in Matlab graphical programming. Creating such objects with *sprintf* requires awkward loops.

You tell *prin* to output a cell array of strings by using two special sequences of 4 characters (one for a row delimiter and the other for a column delimiter). These two sequences were selected because it would be unlikely to want to include either of these sequences in a character string:

Row delimiter: `' ~, '`
Column delimiter: `' ~; '`

For example the statement `prin('aaa ~, bb ~, c')` returns the cell array `{'aaa' 'bb' 'c'}`.

The statement `prin('AA ~, BB ~; 33 ~, 44')` returns this 2x2 cell array:

```
{'AA'  'BB';
 '33'  '44'}
```

and the statement `prin('Line%d ~, Line%d ~, Line%d ~, Line%d',3:6)`

outputs the cell array `{'Line3' 'Line4' 'Line5' 'Line6'}`

Notice that the above statement is repetitive, which is why *vector formats* were created, so instead we can write:

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

The construction above turns out to be quite common, so an alternate and somewhat less cryptic method is available using the `row` delimiter as follows:

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

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

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

File handling

str = prin(fid, FormatString, OptionalArguments)

prin may contain an additional input argument (`fid`) specifying the file identifier, in which case it behaves like *fprintf* in that the generated string is written to the a 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). Here is an example of the use of the `fid` parameter:

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

This creates a file called file1.txt containing the following text:

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

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 three line sequences similar to this 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 (i.e. append mode) use a plus sign instead of the minus sign. (This tells *prin* it should use the `'at'` file permission):

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

If you don't want the file1.txt file to be created in the current directory, the file name in quotes may include a full or relative 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. *prin* closes file2.txt and file3.txt, but will not close file1.txt since it was opened outside the call to *prin*. You might guess that the plus sign in front of `'../file3.txt'` in the above example can be omitted, but this is not correct. This is because *prin* assumes that the first character string in the argument list is the format string, so without the plus sign, the file name would be interpreted as a format string.

As another example, suppose you wanted to create a file with three lines:

```
prin(-'foo.txt','Line1\nLine2\nLine3\n');
```

That would do it of course, but suppose that you needed to write the 3 lines in different parts of your program. The three lines you would need would be:

```
prin(-'foo.txt','Line1\n');
prin(+'foo.txt','Line2\n');
prin(+'foo.txt','Line3\n');
```

The first line creates the file and the next two lines append to it. This is probably the best solution because it is so simple however the following method is somewhat more concise and efficient:

```
prin(-' foo.txt','Line1\n');
prin( 0,'Line2\n');
prin(-1,'Line3\n');
```

That's more efficient is because the file doesn't have to be opened and closed for each line. The space before the file name in this example tells *prin* to exit without closing the file. The next line uses fid=0 which is a special fid that basically means "*use the same file as with the previous prin statement*". The last line uses fid=-1 which behaves the same as fid=0 except that the file is closed when *prin* exits. If you find this sequence too cryptic, rest assure that the efficiency gained by this method would rarely be noticed. Or of course if you prefer to use the traditional Matlab and C form of file handling, the following five lines are equivalent to the 3 lines above:

```
fid = fopen('foo.txt','wt');
prin(fid,'Line1\n');
prin(fid,'Line2\n');
prin(fid,'Line3\n');
fclose(fid);
```

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. (Likewse %%%% will pass %% 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*. In the unlikely event that 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 sequence (`\41`). In the even more unlikely event you need to pass one of the length 4 strings `' ~, '` or `' ~; '` without them being interpreted as cell array delimiters, replace the tilde character with its octal ascii code sequence (`\176`).

str = Pftoa(format string, number)

Pftoa is a subroutine used internally by *prin* to implement the new `V,v,W,w` formats as well as the complex modifiers `J,j,K,k`. Normally you won't call *Pftoa* directly since calling it thru *prin* is more general and just as concise. Also included is a script file *PftoaTest.m* which creates a test file (*PftoaTest.txt*). If you have any confusion about how the new format specifiers behave, looking at some of the examples in this test file may help you better understand them.

Examples

If you want to see more examples of the use of *prin*, you can find many of them in my file exchange submission called "**plt**". *prin* is used several times in each of these 13 programs found in the demo folder: gui1, gu2, julia, movbar, plt50, pltmap, pltn, pltsq, pub, tasplt, trigplt, weight, wfall, winplt. Also in *plt.m* (the main source file of the plotting package) *prin* is used on about 20 occasions, each of which eliminates several lines of needless code.

I hope using *prin* enhances your Matlab experience. Please let me know about any *prin* related questions or suggestions. You can reach me at paul@mennen.org