HOW TO CREATE WEBWORK PROBLEMS

DANIEL DUGGER

Contents

1. Introduction 1
2. Basic structure of a WeBWorK problem 1
3. How to modify existing WeBWorK problems 5
4. Problems containing randomized constants 6
5. Tolerance issues 8
6. Debugging problems, and where to go to learn more 10
7. Getting your problem into the Oregon library 11
8. Writing multiple-choice and matching questions 13
9. Useful facts and functions 15
10. FAQ 17

1. Introduction

This document is intended to be a quick guide to coding WeBWorK problems. The most important thing we have to say is: coding WeBWorK problems can be done quickly and efficiently once you learn just a few basic things! It can even be fun! Like any new thing, it can seem scary and intimidating at first—but in fact it is far easier than solving math problems. The point of this document is to give you a crash course in what you need to know.

In practice people will often code problems by copying an existing problem and then modifying it, and to do that one needs to know fairly little. You can almost get by knowing *nothing*, but it does help to understand a little about the structure of WeBWorK code and certain basic issues that come up over and over. This document will cover those basics, will take you through the steps of modifying an existing problem, and will explain how to go about getting your finished problem into the Oregon library for all to share.

After reading this short document you will not be an expert WeBWorK coder, but it will get you to the point where (with a little practice) you can code about 90% of the things an instructor would ever want.

2. Basic structure of a WeBWorK problem

The structure of a typical WeBWorK problem is as follows, with actual code in black and descriptive text in red:

```
Frontmatter (optional): author, date, keywords, other comments
# This is a sample WeBWorK problem
DOCUMENT();
loadMacros( stuff );
Setup: instructions for setting up constants, variables, etc.
TEXT(beginproblem());
BEGIN_TEXT
Text: main text of the problem
END_TEXT
Answer: instructions for processing the answers
ENDDOCUMENT();
```

Here are some initial guidelines to keep in mind when reading WeBWorK code:

(1) Lines that start with # are for comments and are unprocessed by WeBWorK. All of the frontmatter will consist of such lines.
(2) The "`TEXT(beginproblem());`" command tells WeBWorK to print the problem number, number of points the problem is worth, and problem filename at the beginning. You don't have to have this command, but the problem can look a bit strange without it.
(3) WeBWorK treats spaces and line breaks the same. Inside the TEXT portion of the problem (between the BEGIN_TEXT and END_TEXT commands) you will use the commands $BR for line break and $PAR for "line break plus skip a line".
(4) WeBWorK allows one to write LaTeX code in the TEXT portion, but unfortunately dollar signs are used by the WeBWorK language to mean something different than they do in LaTeX. Instead of using dollar signs for math mode, you need to use \(and \) for opening and closing math mode. Displayed math is done using \[and \] just as in ordinary LaTeX.
(5) The symbols $, %, ^, and _ have special meanings in WeBWorK and so you cannot just type them, even in the Text portion. You will have to use $DOLLAR, $PERCENT, $CARET, and $US. See Section 9 for other special commands.
(6) Outside of the TEXT portion you will mostly see commands to WeBWorK, each of which will end with a semi-colon.

With the above guidelines in mind, let's now look at a complete example of the most basic kind of problem:

```
DOCUMENT();
# This is a sample algebra problem.
loadMacros(
  "PGstandard.pl",
  "MathObjects.pl"
);
# This problem is so simple that we don't need a Setup section.

TEXT(beginproblem());
BEGIN_TEXT
Practice with solving linear equations:
$PAR
Solve the equation \(2x = 5\).
$PAR
Answer: \(x=\) \{ans_rule(20)\}

END_TEXT

$ans=5/2;
ANS(Real($ans)->cmp());

ENDDOCUMENT();
```

Let us talk about the various new components to be found here:

(1) In the `loadMacros` command we are loading the two packages PGstandard.pl and MathObjects.pl. These are two standard packages which will be part of nearly every WeBWorK problem. If you are editing a copied WeBWorK problem you usually don't need to know anything about what the packages are—just use what was already there! For now let us just say that PG is the basic programming language used by WeBWorK, and MathObjects provides an extension of the PG language to provide "object orientation" (don't worry if you don't know what this is).

(2) Now move on to the TEXT portion of the problem. The new thing here is the \{ `ans_rule` \} command. This command produces a blank "answer box" where the user can input an answer; the number 20 specifies the width of this box.

(3) Now let us look past the END_TEXT command. Here we define a variable $ans (WeBWorK uses dollar signs in front of all its variables) to be the number 5/2. The `ANS` command tells WeBWorK how to check whether the user's input was correct or not. Here we take $ans, convert it to a real number (via the `Real` command), and then feed that into the `cmp` routine. This "comparison" routine compares the user's input to what was fed into it.

Anyone who has used WeBWorK knows that the thorny issue is the line between which answers are considered correct and which are not. Most of the annoyances in WeBWorK come down to this line not being coded sensibly. For the above code, the answers 5/2 and 2.5 will both be marked correct by WeBWorK—as will the answers 2.49975 and 2.50025, or anything in between. This range of which answers

are considered correct is controlled by the **tolerance**. In Section 5 below we will talk about how to set the tolerance for the `cmp` routine, and in particular about how to override the default settings (which are not always the best ones).

2.1. **cmp versus num_cmp.**

Some WeBWorK problems are coded to use the `num_cmp` function instead of `cmp`, when the answer is just a number. You will see code like

```
ANS(num_cmp($ans));
```

instead of ANS($ans–>cmp());

The `cmp` function is a slightly newer function that incorporates the older `num_cmp` together with other types of "answer comparison" functions. While `num_cmp` is fine, it is probably better to use `cmp` these days.

3. How to modify existing WeBWorK problems

We will assume that you are up and running on WeBWorK and that you know how to create homework assignments and browse the library (these things are fairly self-explanatory, in the sense that if you don't know how to do them you will figure it out with just a little exploration).

There are at least three different ways you can launch the WeBWorK code editor:

(a) If you are in the WeBWorK homework sets editor and looking at a list of problems, you can select the "Edit" icon (a pen) on any problem to bring up the code editor. Usually this will open a new tab on your browser.
(b) If you are looking at a problem on a homework set (but are not in the Hmwk Set Editor), then at the bottom of the problem you will see "Edit2" and "Edit3" options. Clicking on either of these will open a new tab and launch the code editor. The differences between Edit2 and Edit3 are subtle, so I recommend just using Edit2 all the time unless you know what you are doing.
(c) If you are using the Library Browser, each problem that comes up in a search will have an "Edit" icon in the upper right corner. Clicking on this brings up the code editor in a new tab.

There is nothing wrong with option (c), but I tend not to do it. My general practice is that I only edit a problem after adding it to one of my homework sets (possibly a dummy set).

After launching the code editor you are free to edit the code, View the problem (i.e. test it out), save a NewVersion, or Append the existing version to a homework set. Some incarnations of the editor will allow also give you an Update option, which is like Save. The following is important to know:

WeBWorK will never allow you to change an existing problem in the library. Any edits that you perform are entirely local to you. They will not be saved at all unless you click on NewVersion or Update, and even then they will only be saved into a local version of the problem. So you can feel free to play around editing any problem you want, without fear that you are going to do lasting damage.

Typical steps for modifying an existing problem are:

(1) Add the problem into an existing homework set.
(2) Go to the problem in the set and open the code editor.
(3) Click NewVersion and save the problem under a new name in your local directory. I would typically choose a name like "local/DD251_chainrule_1.pg". Note that .pg is the standard extension for filenames of WeBWorK problems.

    Also, notice that when you create a New Version of a problem WeBWorK gives you the options to have the new version replace the old one in your homework set, or get appended to the end of your homework set, or to have no relation to your homework set. Each of these options is useful at times.
(4) Start editing the problem, and use the View command at the bottom to try out your edits. The default option is to have the tryout open in a new tab/window, which is useful.
(5) When you are done editing (or at various stages while you are editing) use the Update command to save your edits.

4. Problems containing randomized constants

The following example will demonstrate

- the use of randomized constants,
- questions whose answer is a function rather than a number,
- multiple questions in a single problem,
- the use of the WeBWorK `Compute` function.

Here is our WeBWorK code:

```
DOCUMENT();
loadMacros(
   "PGstandard.pl",
   "MathObjects.pl",
);

$a=random(2,20,1);
$b=random(2,20,1);
$c=random(2,20,1);
$f=Compute("$a*x^2+$b*x+$c");
$deriv_f=Compute("2*$a*x+$b");
$crit=Compute("-$b/(2*$a)");

Context()->texStrings;

TEXT(beginproblem());
BEGIN_TEXT
Find the derivative of the function \( f(x)= $f \).
$PAR
\( f'(x)= \) \{ans_rule(30)\}
$PAR
Find the value of \( x\) where the function \( f \) has a horizontal tangent line.
$PAR
\( x= \) \{ans_rule(20)\}

END_TEXT

ANS($deriv_f->cmp());
ANS($crit->cmp());

ENDDOCUMENT();
```

Comments:

(1) Variables a, b, and c are randomly chosen from the interval $[2, 20]$, with the distance from the left endpoint (2 in this case) being a multiple of 1. In other words, these are randomly chosen integers from the interval $[2, 20]$. If we had

used the command `random(2,20,0.5)` they would be randomly chosen half-integers, whereas `random(200,250,10)` would give a randomly chosen multiple of 10.

(2) The function `Compute` is sort of magical: is takes a mathematical expression and converts it into an "object" that WeBWorK can work with. In particular, the output of `Compute` functions can be put into the `cmp` routine directly. In the above code we used `Compute` to construct two function objects ($f and $deriv_f) and one number object ($crit).

Note that we could have changed two lines in the code as follows:

```
$crit=-$b/(2*$a);
ANS(Real($crit)->cmp());
```

Here we have left out the `Compute` in the definition of $crit, but then we need to convert $crit into an appropriate form before sending it to the `cmp` routine.

Note also that `Compute` recognizes that the role of "x" is to be a function variable (not a WeBWorK variable—it is not "$x"!), so that the resulting object $f is a function. This would not work out of the box with another variable like "u"—the letter "x" is specially coded in WeBWorK as an automatic variable. If we want to make a function of u then we need to use the two commands

```
Context()->variables->add(u=>'Real');
Compute("$a*u^2+$b*u+$c");
```

The first command adds u to WeBWorK's list of variables, and the Real says that it is a variable that can take real numbers as values.

(3) The `Context->texStrings` command is not essential, but the text of the problem will look odd without it. Basically, this command tells WeBWorK that when it encounters a variable (like $f) it should display it in a LaTeX-appropriate format. Without this command, WeBWorK would write $f(x) = (14 * x^2 + 3 * x + 4)$ instead of $f(x) = 14x^2 + 3x + 4$ when displaying the text of the problem.

(4) This problem has two `ans_rule` commands, and so the user will have to enter two answers. Note the two corresponding `ANS` commands in the final section of the code; the first `ANS` goes with the first `ans_rule`, and so on. You can have any number of `ans_rule` commands, but if you don't have a corresponding number of `ANS` commands you will get errors.

Supplement:

Here we give some sample commands, with brief explanations.

```
do {$a=random(1,10,1);} until (($a != 3) and ($a !=6));
```
[keep picking random numbers until you get one that is not 3 or 6]

```
$a=list_random(0.05,0.075,0.85);
```
[pick a random element of the list]

5. Tolerance issues

Tolerance is a much dicier topic than it might first appear. There are two basic types of tolerance: *absolute* and *relative*. If the correct answer is A, an absolute tolerance of ϵ will recognize any answer in $(A-\epsilon, A+\epsilon)$ as being correct. In contrast, a relative tolerance of ϵ will recognize any answer in $((1-\epsilon)A, (1+\epsilon)A)$ as being correct.

It might be tempting to take the approach of "I will always use absolute tolerance of 0.001". This will work great most of the time, but what if the answer to a problem is $3.2 \cdot 10^{-5}$? Then it fails miserably. The second temptation is to say "Well then, I will always use a relative tolerance of 0.01. Surely that is reasonable." But then what if the answer to a problem is 123512? Do you really want to regard an answer of 122279 as correct?

These examples demonstrate something very important: **there is no simple approach to tolerance that will work well in every problem**. When writing WeBWorK problems, especially those that use randomization, it is important to know what the range of answers will be, and to use that knowledge to inform your choices about tolerance settings. A large percentage of "errors" in WeBWorK problems are due to the coder not making good decisions about this.

You can specify the tolerance settings when you call the `cmp` routine. Here are some examples:

```
ANS($ans–>cmp(tolType=>'absolute', tolerance=>0.001));
ANS($ans–>cmp(tolType=>'absolute', tolerance=>0));
ANS($ans–>cmp(tolType=>'relative', tolerance=>0.01));
```

In the second case, note that the answer will have to be exact in order to be marked correct.

The `cmp` routine comes with some default tolerance settings built in, but these can change depending on the Context settings in some problems. Theoretically the default is for relative tolerance with a setting of 0.001, but I have seen cases where this didn't seem to be working as advertised. I think it is good practice when writing WeBWorK problems to never use the default settings, and instead explicitly code what you want the tolerance to be. Additionally, my personal suggestion is to never use relative tolerance; it is too confusing for students to understand why some answers are correct and some are incorrect. Relative tolerance has never served me well.

It is possible to set the tolerance settings globally throughout a problem, which can be useful if there are multiple questions and answers. To do this insert the following code near the beginning of the problem:

```
Context("Numeric");
Context()–>flags–>set(tolerance=>0.001, tolType=>'absolute');
```

At the end of the problem you can now write things like ANS($ans–>cmp()); and it will use the global tolerance setting that was specified above.

Remark 5.1. This is getting down into the weeds, but the relative tolerance option does come with a "cutoff" setting so that if the answer is too small then absolute tolerance is used instead. These settings are via `zeroLevel` and `zeroLeveltol`; the defaults are 10^{-14} and 10^{-12}. Theoretically this could make relative tolerance

a reliable default, but I am skeptical. I still think it produces behavior that is too difficult for most students to understand.

5.2. **Rounding.**

An unfortunate aspect of WeBWorK is that neither of the tolerance types do what we often want students to do, which is something like "round your answer to the nearest one-hundredth". For this one is tempted to use an absolute tolerance of 0.005, but it doesn't work. If the unrounded answer is 2.357, any answer in the range (2.352,2.362) will be marked correct. That of course includes the correct answer of 2.36, but it also allows students to submit 2.3615 or 2.353—both of which should really be marked wrong. Even worse, if the unrounded answer is 2.355 then the correct answer of 2.36 will not be marked correct. Ugh!

WeBWorK has the `round` function that rounds numbers to the nearest integer. If the unrounded answer to a problem is $ans, we could use the following code for rounding to the one-hundredth place:

```
$rounded_ans=1/100*round(100*Real($ans));
ANS(Real($rounded_ans)->cmp(tolType=>'absolute', tolerance=>0));
```

The `Real` conversion should not be necessary in the first line, but I have seen some buggy behavior without it.

Note, however, that rounding is subtle for negative numbers. Nothing is ever easy! The above code will round -2.345 to -2.34, since the 5 "rounds up". This is fine if that is your convention, but many people use the "round away from 0" rule (sometimes without understanding that is what they are doing). To round away from zero you would need to use the code

```
$rounded_ans=1/100*sgn($ans)*round(100*Real(abs($ans)));
ANS(Real($rounded_ans)->cmp(tolType=>'absolute', tolerance=>0));
```

Warning 5.3. Having WeBWorK round the answers and then using zero tolerance is appealing, but it comes with its own set of issues. First, you need to make sure the instructions of the problem explicitly tell students to round! Second, if the problem involves several steps and the students are rounding at each step, it is likely those errors are going to propogate and produce an incorrect final answer. Students need to be trained to keep a few more digits in their computations than the problem asks for in the final answer.

6. Debugging problems, and where to go to learn more

You have just written your first WeBWorK problem, and you excitedly go to try it out. But instead of running smoothly, WeBWorK gets angry and starts cursing at you. Lines and lines of incomprehensible error messages spew out. Though you don't understand the language, you can tell by the tone that the software hates you. What do you do?

First, take a deep breath. This is how coding always goes. The same thing probably happens half the time you compile a LaTeX document. You will get through this!

I find the WeBWorK error messages particularly frustrating. I hardly ever understand them. In some cases they sort of identify the general ballpark of where the error is located, but they are not super reliable or particularly descriptive. If you find yourself wanting to track down the people who wrote the software and physically assault them with a canteloupe, just know that you are not alone.

Okay, back to your deep breathing. Nine out of ten times the error is something silly. You forgot a semicolon at the end of a command. You forgot to put a dollar sign in front of a variable. You forgot to format a variable correctly (e.g. with `Real` or `Compute`) before sending it to `cmp`. You forgot to have the number of `ANS` commands match the number of `ans_rule` commands. You forgot to pair up the LaTeX commands for entering and leaving math mode. Go back and check through this stuff carefully.

One out of ten times (or less) it is a subtle error that takes effort to track down. Here you do the typical debugging things. Comment out portions of your code with # signs and recompile, and repeat until you can pinpoint the source of the error. Usually the error will make sense to you once you locate it, but if it doesn't then start scouring the internet for information. There is a ton of information about WeBWorK around the internet (including an MAA Wiki) and it is mostly sort of decent. My experience has been that *most* of the problems I have run into are things I have been able to solve fairly quickly—not all, but most.

What if you want to learn more about WeBWorK than is covered in the present document? Again, the internet is your friend...just let your fingers do the walking! As an example, if I search for "including graphs in webwork" then the first link I find is an MAA page that has examples as well as general discussion. Many of the MAA pages are set up this way.

Another "Getting Started" guide that seems to be pretty good is here:

`https://personal.math.ubc.ca/~cwsei/docs/UBC_Math_WeBWorK_Manual.pdf`

Could the overall documentation situation be better? Yes—otherwise I would not have had to write this document! But it could also be a lot worse, so let's try to think positively.

7. Getting your problem into the Oregon library

To have your problem incorporated into the Oregon WeBWorK library you should follow the steps below (we give further details after the list):

(1) Thoroughly debug your problem.
(2) Add frontmatter material detailing the authorship, institution, date, intended course, keywords, etc.
(3) Adopt a sensible naming convention for the file and export it.
(4) Send the file to Jennifer Thorenson for inclusion into the library.

7.1. **Frontmatter.** The frontmatter for your code should look something like the following:

```
## DESCRIPTION
## Chain rule practice
## ENDDESCRIPTION

## DBsubject(Calculus - single variable)
## DBchapter(Differentiation)
## DBsection(Chain Rule (without trigonometric functions))
## Date(6/26/2022)
## Institution(University of Oregon)
## Author(Fred Flinstone)
## Level(3)
## KEYWORDS('calculus', 'derivative', 'chain rule')

# UO Course: Math 251
```

(1) The DB lines refer to where the problem should be classified in the WeBWorK database. You can see the options by looking through the Library Browser. These lines are **not** relevant if the problem is only to appear in the Oregon library, but you might as well write your problem in such a way that it could be incorporated in the Open Problem Library one day.
(2) The Level refers to Bloom's taxonomy, and is described here:
    http://webwork.maa.org/wiki/Problem_Levels
    I personally never pay attention to the level, but it is probably a good idea.
(3) The keywords are again relevant only if your problem is eventually part of the Open Problem Library. They are not usable if it is only in the Oregon library. The current list of searchable keywords is here:
    https://hobbes.la.asu.edu/Holt/keywords.html
    Searching through all of the options would take too much time, so just pick two or three reasonable ones.
(4) For inclusion in the Oregon database the main things you should pay attention to are the UO course number, the author, and the description at the beginning.

7.2. **Exporting your file, and naming conventions.** To export your file go to the File Manager in WeBWorK. Click on the "local" directory by clicking it twice. Click once on the file for your problem, so that it is highlighted, and then select "Download". This will download a text file with the code to your computer,

which can then be emailed to someone else. If you are exporting multiple files you can select a block of files and then "Make Archive". Then select the archive and download it.

It is not easy to make an archive with files that are not in an easily selectable block, though. The following steps work:

(i) Create a new folder.
(ii) For each file you want in your archive, select it and then choose "Copy". You will have to insert the new folder name at the beginning, and also delete the "_1" text that WeBWorK automatically adds to the filename (annoyingly).
(iii) Once you have copied all the files into your folder, move to the new folder, select all files, and then "Make Archive".

Going forward, we are going to try to have the Oregon Library organized so that the directory structure follows the subject/chapter/section organization of the national library. This will make it easier to find problems based on topic. If you are sending Jennifer Thorenson problems, the best thing is to make sure that any archive you send only includes problems for one subject/chapter/section combination. In your email, let Jennifer know that trio so that she can upload the problems to the appropriate spot in the library.

For naming conventions, it is useful to adopt filenames for the problems are that identify the course, topic, and are also reasonably unique so that there is no conflict when uploading into the Oregon library. A good prototype is something like

UO251_chainrule_intermed_DD06262022.pg

The final characters are my initials and the date.

8. Writing multiple-choice and matching questions

8.1. **Multiple choice.** A basic multiple-choice problem will ask a question and then give a randomly ordered selection of responses to choose from. To use this feature you will need to load "PGchoicemacros.pl"in the `loadMacros` command.

Here are some sample sections of code for a multiple-choice problem:

```
loadMacros("PGstandard.pl","MathObjects.pl","PGchoicemacros.pl");
Setup:
$mc=new_multiple_choice();
$mc->qa('Who was the first U.S. president?','Washington');
$mc->extra('Jefferson','Lincoln','Einstein');
$mc->makeLast('none of the above');
Text:
\{$mc->print_q()\}$BR
\{$mc->print_a()\}
Answer:
ANS(radio_cmp($mc->correct_ans));
```

Let me stress that this is not a complete WeBWorK problem, but rather three snippets of code that would be inserted in the Setup, Text, and Answer sections.

Some comments:

(1) We begin by initializing a new "multiple choice object" and assigning it to the variable name $mc (you can choose whatever variable name you want).
(2) The next line sets the question and (correct) answer.
(3) The third line adds the "extra" answers which will form the multiple choices.
(4) The fouth line of code is not necessary, but sometimes there will be a "final choice" that should really be listed last and not have its order randomized with the others. That is what we have here.
(5) In the Text portion of the code we see two instructions, one for printing the question and one for printing the answers.
(6) The ANS code at the end will always have the same format except possibly for the variable name $mc, which will be whatever name you chose for your multiple-choice object.

That's all there is to it!

8.2. **Matching questions.** A matching question will give two lists and ask the user to match every item in the first list with a corresponding item in the second. Here are the code snippets:

```
loadMacros("PGstandard.pl","MathObjects.pl","PGchoicemacros.pl",
        "PGanswermacros.pl");
Setup:
$ml=new_match_list();
$mc–>qa('Who was the first U.S. president?', 'Washington',
     'Which president made the Louisiana purchase?', 'Jefferson',
     'Who was president during the U.S. civil war?', 'Lincoln');
$ml–>rf_print_q(~~&pop_up_list_print_q);
$ml–>ra_pop_up_list([No answer=>'?', A=>'A', B=>'B', C=>'C']);
$ml–>choose(3);
Text:
\{$ml–>print_q()\}$BR
\{$ml–>print_a()\}
Answer:
ANS(str_cmp($ml–>ra_correct_ans));
```

Much of the above code is self-explanatory, but here are a few comments:

(1) Note that for matching questions you need the two packages "PGchoicemacros.pl" and "PGanswermacros.pl".
(2) The second list (in this case, the names of the presidents) will be randomly ordered and labelled A, B, C, etc. The `ra_pop_up_list` command specifies how these labels will appear in the pull-down menus. I don't know why someone would want option A to appear as anything other than 'A', but you can change it if you want!
(3) It is possible to code in a list of ten matching items and then have WeBWorK randomly choose four of them to use in the problem. This is the reason for the `$ml->choose` command. In our example we coded in three options and WeBWorK will choose three of them, so that means all options will be used.

9. Useful facts and functions

9.1. Basic math commands.

Theoretically you can use either ^ or ** for exponentiation, but I have sometimes run into situations where the former did not work correctly. I think that inside a `Compute` function you can use these interchangeably, but if you are just having WeBWorK do arithmetic operations on its own (e.g. "$a=$b^2") you should use **; for some crazy reason ^ is often interpreted as + by WeBWorK.

All other arithmetic operations are standard. You can use parentheses ( ), square brackets [ ], and braces { } interchangeably.

`min`(a,b)	returns the minimum
`max`(a,b)	returns the maximum
`gcd`(a,b)	returns the gcd
`lcm`(a,b)	returns the lcm
`ceil`(a)	returns the ceiling
`floor`(a)	returns the floor
`sgn`(a)	returns the sign (-1, 0, or 1)
`abs`(a)	
`sqrt`(a)	
`exp`(a)	returns e^a
`ln`(a)	returns $\ln(a)$
`log`(a)	returns $\log_{10}(a)$
`sin`(a)	also cos, tan, sec, arcsin, arccos, arctan, sinh, cosh, tanh, sech (all use radians)
`fact`(n)	returns $n!$
`pi`	3.14159265358979
e	2.71828182845905

9.2. Text commands.

$DOLLAR	$
$PERCENT	%
$CARET	^
$US	_ (underscore)
$BR	line break
$PAR	line break and vertical space
$BCENTER text $ECENTER	centered text
$BBOLD text $EBOLD	bold text
$BITALIC text $EITALIC	bold text
$BUL text $EUL	underlined text

9.3. Statistics.

For statistics problems one needs to be able to compute the integral of Gaussians and t-distributions, as well as the "inverses" to those functions. For this you will need to load the "PGstatisticsmacros.pl" package in the `loadMacros` command. Once you have done that, the following commands are available:

`uprob`(z)	gives the integral from z to ∞ for the standard normal distribution
`udistr`(p)	gives the value of z such that `uprob`$(z) = p$
`tprob`(df,t)	gives the integral from t to ∞ for the t_{df} distribution
`tdistr`(df,p)	gives the value of t such that `tprob`$(df, t) = p$.

9.4. **Conditionals.**
The format for a conditional statement is as follows:

```
if ( $c < $a ) {
$ans="Yes";
}
else { $ans="No"; }
```

WeBWorK uses || for OR and && for AND. For example,

```
if ( $c < $a  && $d>0) {
$ans="Yes";
}
else { $ans="No"; }
```

9.5. **Looping.**
Maybe your problem is "Add the first $n terms of the harmonic series". To have WeBWorK compute the answer you could use the following code:

```
$sum=0;
for ( $i=1; $i <= $n; $i++) { $sum=$sum+1/$i;}
```

10. FAQ

(1) How do we handle Yes/No questions, or other questions where we want the answer to be a string?

Answer: First, make sure to load MathObjects.pl. Then in the setup to the problem add the following lines:

```
Context()–>strings–>add(No=>{},N=>{alias=>"No"});
Context()–>strings–>add(Yes=>{},Y=>{alias=>"Yes"});
```

Then in the Answer portion of the problem add something like

```
$ans="Yes";
ANS(Compute("$ans")–>cmp());
```

The default is that strings are case-insensitive (so that "yes" and "Yes" and "YES" all mean the same thing), but this can be overridden if you really want.

(2) How do we handle an array in WeBWorK?

Answer: There are cases where a problem might involve two constants $a and $b which are related by a non-mathematical formula, but where you want WeBWorK to randomly choose between a few pre-selected possibilities for the pair ($a,$b). Here is an example where this is accomplished via an array:

```
@c=([0.5,12.7],[0.9,14.1],[1.3,17.3]);
$i=random(0,2,1);
$pair=$c[$i];
$a=$pair–>[0];
$b=$pair–>[1];
```

The first thing you need to know is that any list-like object is indexed so that the first entry has index 0. The first line defines an array @c (arrays are preceded by the @ sign) where the 0th object is the pair [0.5,12.7], the 1st obect is $[0.9, 14.1]$, etc. We then define $i to be randomly chosen from 0, 1, and 2. $c[$i] selects the $ith element from @c, and then we define $a and $b to be the 0th and 1st elements of that pair. Why can't you just write $pair[0] and $pair[1] here, which would seem fairly reasonable? Ugh, don't ask.

(3) WeBWorK's "Answer Preview" only shows a few digits after the decimal point even if the student enters more. I want the Preview to better match what the student enters.

Answer: How WeBWorK treats numbers in a problem is governed by a global Context setting. The default is something like 6 significant digits—but this does not mean 6 digits after the decimal point, it means 6 digits altogether, starting from the left. So a number like 10532.281 would automatically be changed to 10532.3 as seen by WeBWorK. To change the setting you can put a command like the following into the Setup portion of the code:

Context()->{format}{number}="%0.8g"

for eight significant digits, or

Context()->{format}{number}="%0.5f"

for a fixed five digits after the decimal point.

For more information about this topic see

http://webwork.maa.org/wiki/Modifying_Contexts_(advanced)

Department of Mathematics, University of Oregon, Eugene, OR 97403
Email address: ddugger@uoregon.edu