Problem Generation with WeBWorK

Introduction 3

How to Use Templates 3

PG vs. PGML 4

- Problem Text 4
- Answer Entry 5
  - Answer Blanks 5
  - Answer Boxes 5
  - Answer Arrays 6
- Solution Text 6

PGML 7

- Formatting 7
  - Bolding 7
  - Italics 7
  - In-line Code 7
  - Verbatim 8
- Basic Lists 8
- Sublists 9
- Equations 9
- Common Errors 10
  - Warning: unknown block type 'balance' in PGML::Format::html::format 10

Types of Comments 10

Variables vs. MathObjects 10

MathObject Declarations 11

Random Values (MathObjects) 12

- Basic Randoms 12
- Do-Until Randomization 12
- Non-Zero Randoms 13
- Randoms from Lists 13

Equation Entry 13

- PG Equations 13
- PGML Equations 14
  - Calculator Notation 14
  - LaTeX Notation 14

Symbols **14**

Strings **14**
Adding Strings to the Context 15

Graphing **15**
Initialization 15
Labels 16
Label Loops 16
Custom Colours 17
Adding Functions 17
Adding Points 17
Adding Shading 18
Inserting the Graph 18

Answers **19**
Basic Answer Checkers 19
Eval vs. Cmp vs. Substitute 19
Tolerance Values 20
Answer Hints 20
Multiple Answers 21

AnswerFormatHelp **21**

Graders **22**
Average (Default) 22
Standard 22
Full-Partial 23
Weighted 23
Weighted with Credit Answer Option 23
Incremental Weighted 24

Formatting **25**
Infinity Formatting 25
Other “Special” Formatting 26

GeoGebra **26**
Inserting a GeoGebra Applet 27
Known “Gotcha”s 28
Multiple GeoGebra Applets 28
Updating Applets 28

Appendix A: Macro File Descriptors **29**

Introduction

The following documentation provides general instructions and information regarding the modification of WeBWorK problems. However, it (and its associated templates) are primarily targeted at University of Lethbridge Math/Comp. Sci. professors. This can be seen in the template files, which have been tagged using `#-ULETH-#` / `#-ENDULETH-#` tags. All code that is enclosed within these tags is effectively "flagged for modification". This includes things like the actual problem text, the solution text, values of variables that are used in the question, and so on. There will also usually be some other comments in the area to explain what the function of the following code is, just to give an idea of how you might want to change it. Of course, there's nothing wrong with modifying code outside of the tags, but you may want to make a backup of the file before you do so. If you know what you're doing, you can get creative with the question formatting, but it's also easy to completely break things. Be careful!

How to Use Templates

See BasicTemplate.pg for a coded example of this.
All files from the ULeth database have been tagged using `#-ULETH-#` / `#-ENDULETH-#` tags. These enclose areas of code that you may want to modify to adjust the question.
Problem files are generally divided up into five main sections, each of which is described below.

1. Initialization: This acts as a sort of "preamble" to the code that follows in later sections. It includes opening statements for the document (e.g. `DOCUMENT();` and `TEXT(beginproblem());`), as well as macro loading statements. The latter are responsible for importing various packages that will be used for the problem. If you are working with templates, it's likely that you will not have to modify this list at all. However, if you are designing your own problems, you may want to refer to Appendix A.
2. Setup: This is where the setup for the problem is done, including setting up values, defining formulas, and so on. You can also place restrictions on various values, including the student's input.
3. Problem Text/Main Text: This section is where you will enter the actual text for the question you want the student to answer. This can include values that you initialized in the Setup section if they are used in the question rather than/in addition to the answer.
4. Answer Evaluation: Here is where you set up "answer checkers". These are explored in more detail in a later section, however it's worth noting for now that these are responsible for analyzing the student's input to see if it matches the correct answer to the problem.
5. Solution: Lastly, this section is where you can include an explanation of the problem's solution. This can be useful for students who are struggling with the problem or who want to check their answers. You can choose when/if the solutions are available by changing settings in the WeBWorK application. Additionally, this section also has an

`ENDDOCUMENT();` tag, which is simply used to mark the end of the overall document text.

PG vs. PGML

One common distinction that is made within this documentation is the syntax differences between PG and PGML. Just what are those, though? Well, to put it simply, PG stands for "Problem Generation", while PGML stands for "Problem Generation Markup Language". PG was developed first, and PGML came about once it became clear that the layout of questions could be just as important as their content. As a result, most of the templates you come across will use PGML (or at least, have it as an option), as it is the more "modern" version of creating WeBWorK problems. An easy way to tell is to check if "PGML.pl" is included in the loadMacros function near the top of the .pg file.

When it comes to actually developing problems, most of the underlying code is the same, regardless of whether PGML or PG is being used. Additionally, many of the core syntax differences are outlined in the different sections of this documentation. However, there are some key things that are worth noting. These can also be used to distinguish between files that implement PG, and those that implement PGML.

Problem Text

In PG files, problem text is surrounded by the following lines:

```
Context()->texStrings;
BEGIN_TEXT

…

END_TEXT
Context()->normalStrings;
```

To unpack this, the first line basically declares that, until further notice, all subsequent lines of code should be treated differently from the standard code strings that make up your problem. Then, the `BEGIN_TEXT` line marks the start of the problem text, which is entered where the … is. Then, the last two lines mark the end of the problem text and switch back to reading the file lines as regular code strings.

In PGML files, this is greatly simplified. Problem text is simply surrounded by:

```
BEGIN_PGML

…

END_PGML
```

Here, each line simply declares the beginning/end of a section that should be read as markup instead of regular code. Once again, the problem text just gets entered where the … is.

Answer Entry

Answer blanks are the areas of a problem where students are able to enter their answers. As is the case with so many things, the syntax for inserting them differs between PG and PGML. However, the common practice for both cases is to declare the blanks within the main problem text. Also of note is the fact that all of the answer blank declaration methods that are specified for PG can be implemented with PGML; simply format the text as in-line code. While this isn't particularly useful for the regular answer blanks (unless you like the PG syntax better), it is for answer boxes and the like, as there is currently no good way to insert these in PGML.

Answer Blanks

In PG, answer blanks are declared using `\{ans_rule(#)\}`, where `#` is replaced with a positive integer value. This value determines the width of the answer blank, so `\{ans_rule(10)\}` would give a blank that is 10 characters wide. However, this does not actually restrict the amount of text that can be entered into the blank; it will just scroll once the carat reaches the end of the blank.

In PGML, answer blanks are declared using `[_]`, where _ can be any number of underscore characters. Similar to the bracketed number for PG answer blanks, the number of underscores determines the width of the answer blank. For example, `[_ _ _ _ _ _]` would insert an answer blank that is six characters wide. Note that the spaces have been added between the underscores here for clarity's sake; do not add spaces between the underscores in your problem! Also note that the width of the blank does not actually restrict the amount of text that can be entered into the blank; it will just scroll once the carat reaches the end of the blank.

Answer Boxes

Answer boxes are simply multi-lined answer blanks. They are functionally identical to regular answer blanks, but can be helpful for students if an answer makes more sense when displayed over multiple lines. An example where answer boxes are useful is for Matrix entry. A multi-line answer box gives students the space to place each row of the Matrix on a separate row of the answer box, making it easier to figure out which values go where.

The syntax for inserting an answer box is actually very similar to the regular answer blanks. All you have to do is enter `\{ans_box(#1,#2)\}`, where `#1` dictates the number of lines in the box, and `#2` determines how wide the box is. Both need to be positive integers. As is the case with the answer blanks, this doesn't actually prevent students from entering more characters than this into the box; it determines how much text can be entered before it has to be scrolled through.

Answer Arrays

Answer arrays are similar to answer boxes, but provide a separate answer blank for each coordinate or entry of a Point, Vector, or Matrix. The syntax is `\{$mo->ans_array(#)\}`, where `$mo` is a Point, Vector, or Matrix MathObject and `#` is a positive integer that determines how wide each answer blank will be. This doesn't restrict how many characters students can enter; just how wide the blank appears on the page. For example, given the following Matrix:

```
$mtx = Matrix([[3,5],[1,2]]);
```

Calling `\{$mtx->ans_array(3)\}` would insert a 2x2 grid of answer blanks into the problem text, each of them three characters wide.

There is one hitch with answer arrays: you can't create an answer array from a MathObject that is being used by more than one answer checker. In other words, in the above example, `$mtx` could not be used in any other answers. The reason for this is that `ans_array` permanently changes the way the MathObject is processed by the answer checker: namely that each coordinate/entry is treated as a standalone value. If you used `$mtx` as the answer value for a question that only had a single answer blank, whatever value was in that blank would be compared against the first value for `$mtx`: 3 in this case. However, if the question was asking for a full Matrix, you would be checking if a Matrix was equal to 3; most likely not the case! As such, if you want to ask multiple questions with the same answer value, create separate MathObjects for any questions that will be using answer arrays.

Solution Text

In all problems, the solution text has to be specified differently from the problem text. This allows it to be hidden from the student until the answers become available.

For PG files, the solution text is surrounded by:

```
BEGIN_SOLUTION

…

END_SOLUTION
```

This simply declares that everything in place of the … should be treated as solution text.

Unlike with the problem text, the syntax for PGML is very similar.

```
BEGIN_PGML_SOLUTION

…

END_PGML_SOLUTION
```

Again, everything in place of the … should be treated as solution text.

PGML

Formatting

For a more thorough coverage of the different PGML formatting options (including some not covered here), refer to the PGML cheatsheet.

One thing to note with custom formatting in PGML: Formatting methods often don't mix. For example, if you try to put some bolded text in the middle of some LaTeX, it will likely cause problems. Now, in some cases, such issues can be circumvented by switching around brackets. An example is that you can insert code within LaTeX, while the opposite tends not to work well. Basically, if you're trying to do something special, it doesn't hurt to experiment, but for the best results, you're better off sticking to the basics.

Bolding

If you enclose some text in asterisks (*), that text will appear in bold. Note that this will only work when you have asterisks in the regular text of the problem. In other words, asterisks that are entered within LaTeX equations and the like will generally be interpreted as multiplication symbols. Asterisks are also used as options for certain other formatting options, so it's a good practice to leave a space between your "bolding" asterisks and the surrounding text so that it's clear what they apply to.

E.g: `In this sentence, *this text is bolded*.` Will appear as: In this sentence, **this text is bolded**.

Italics

Text that is enclosed in underlining (_) will appear in italics. Similar restrictions and suggestions apply as with asterisks: use spacing to make it clear what is being italicized and make sure to not confuse the italic underlines for any other underline characters in your markup.

E.g. `In this sentence, *this text is italicized*.` Will appear as: In this sentence, *this text is italicized*.

In-line Code

In some cases, you may need to run some code within the main text of the problem. This may be because of some special text you want to display, as is the case with the AnswerFormatHelp that is present in many problems next to the answer blanks. Alternatively, it may be because you're using special values that can't be accessed through the usual square brackets that are used in PGML. See the main text of

ULethQuestionBank/Functions_in_R2/findTheFormulaOfTheParabola.pg for an example in which the values need to be pulled out of an array and therefore have to be accessed using inline code.

At any rate, to add inline code in PGML, simply enclose the Perl code to run within `[@ @]*`. Note the asterisk that follows the closing `@]*`.

Also, in-line code can generally be inserted within LaTeX equations in your PGML, though you need to take care to switch notations properly. LaTeX notation will not be parsed properly when the in-line code tries to run, and likewise, the code will not parsed as LaTeX. If the question breaks, check that your brackets are properly placed and that you're not mixing up syntax!

Verbatim

On occasion, PGML is a bit *too* smart. What I mean is that it'll sometimes try to parse text in a way that you don't want it to. In particular, when entering Interval and Matrix objects using square brackets, PGML will often try to interpret the contents of the brackets as a MathObject. This will often break the problem or (at minimum) generate errors if the text within cannot be parsed in this way. Luckily, there's an easy fix to this: verbatim brackets.

Any text that is enclosed in `[| |]` will be printed verbatim onto the page; it will not be "processed" in the same way that the other text for the problem is. This allows you to print out things like [2,5] without running into errors.

Basic Lists

In many cases, you'll want to have a list of subproblems available on one question. The implementation of these is quite straightforward in PGML. The basic syntax is simply:

```
a. Item A

b. Item B

c. Item C
```

Note that you have to leave a blank line between each list item. Also, the following "list markers" are available:

```
- 1.
- i.
- I.
- a.
- a)
- A.
- A)
```

```
- +
- -
- *
- o
```

In a few cases, the list numbering will be misinterpreted when it gets displayed. For instance, in a letter-based list that goes up to i/I, that item may be interpreted as item 1 in a Roman numeral-based list. Then, all the items below will restart the lettering back at a/A! Obviously, that is generally not desired, so the solution is to take advantage of PGML's auto-incrementing. If you're entering a list that uses letters, just substitute in some other letter for i/I and v/V; for instance, just have two questions labeled h/H or u/U in a row.

```
g. Item 1

h. Item 2

h. Item 3

j. Item 4
```

Then, when this text appears in your actual problem, it will appear with the appropriate g, h, i, j lettering. If you're using a bullet point-based list, you obviously don't have to worry about any of this.

Also, if you want to have a new line in a list entry, put the new line below the entry in the code, but indent it with four spaces. Example:

```
1. Item 1
  This is also part of item 1.

2. Item 2
```

Then, to designate a new item in the list, just leave an empty line.

Sublists

To add lists within lists, PGML relies on indentation. In the previous section, it was mentioned that placing four spaces ahead of a line right below the list entry will add it as a second line to the list entry. However, if you instead want this new line to be the first entry in a sublist, just hit Enter to leave a blank line between it and the "root" list entry. Then, use whatever numbering/lettering scheme you wish for the sublist! You can do this as many times as you want; for each new sublist, just preempt the first line with a blank line, then insert four extra spaces (compared to the parent list entry) before each item in the sublist.

Equations

For more information on inline equations in PGML, please refer to this section.

Common Errors

Warning: unknown block type 'balance' in PGML::Format::html::format

This error most commonly occurs if you have stray square brackets in your PGML markup. For example, maybe you added an extra closing bracket at the end of some inline code or a MathObject substitution? Maybe you hit the wrong key by accident? Whatever the case, the solution is usually to just carefully comb through your PGML markup and ensure that every opening bracket has a closing bracket and vice versa.

Now, you may also run into this problem when typing Matrix objects and the like, since they use square brackets for their basic notation. Unfortunately, WeBWorK will often complain about this, as it will try to process the values between the square brackets when (in general) you just want them displayed. Luckily that's what verbatim brackets are for! Just follow the instructions here, and you should be good to go!

Types of Comments

Within the .pg files, you will notice a number of lines preceded by a '#' sign. These mark code comments, which do not get executed when the problem is being run. However, they do provide information on the functionality of various parts of the problem, as well as separating parts of the problem (e.g. Setup, main text, etc.) for clarity's sake. Comments can be placed almost anywhere in the code; the exception is between BEGIN_TEXT/BEGIN_PGML and END_TEXT/END_PGML tags, as well as the solution tags. The reason that comments cannot be placed between these is that everything between the tags gets treated differently from standard code. As such, if you try to insert comments here, they will be printed along with your problem text. Obviously, this can be quite confusing to use! For this reason, comments regarding the problem and solution text are placed outside of the tags.

Variables vs. MathObjects

When looking at a .pg file, there will often be two different types of tokens in use: variables and MathObjects. Below is a table detailing the key differences between them.

Variables	MathObjects
Name is a string of characters whose first character must be a letter.	Name is a string of characters whose first character must be a '$', and whose second character must be a letter.
Used in the question "foreground". Will be used in	Used in the question "background". Will

the problem text, the student's answer, or both. E.g. If the student is creating a formula for f(x), they will need the Context variable 'x'.	be used in the problem code, but will not actually be visible to the student. E.g. You may define a MathObject $answer that stores the value of the question's answer, which will then be compared against the student's answer when they submit theirs.
Sample declaration in code: `Context()->variables->add(x=>'Real');` Declares variable x in the problem and defines it as being a real number.	Sample declaration in code: `$answer = Compute("15");` Declares MathObject $answer in the code, and gives it a value of 15 as a Real number.
Methods of reference change depending on the situation. If you refer to them within a Formula statement (e.g. `Formula("3 x")`), then they will be treated simply as algebraic variables (e.g. 3x). Otherwise, they will be evaluated, and by default they are set to 0. E.g. `t+2` will display a value of 2.	You can always reference the MathObject by its name, regardless of situation. Just note that you may need to "escape" it by surrounding it with square brackets (PGML) or `\( \)` (PG) if you are using it within your problem text.
Variable limits have to be specifically defined in the Context if being used. Use `Context()->variables->set(var => {limits => [min, max]});`, where `var` is the variable name, `min` is the minimum possible value, and `max` is the maximum possible value.	MathObjects limits get specified when the object is initialized. For instance, if a random value is being assigned, the range of possible values gets specified (see here). Otherwise, you will most likely just be specifying static values.
For more information: http://webwork.maa.org/wiki/Introduction_to_Contexts#Variables	For more information: http://webwork.maa.org/wiki/Introduction_to_MathObjects#How_to_create_a_MathObject

MathObject Declarations

There are some general rules to keep in mind when declaring MathObjects in your code. While there are exceptions to these, it's best to follow these rules when initially setting up your problem, then experiment later if something isn't working quite right.

1. Use `Compute` when declaring MathObjects instead of the more specific constructors (e.g. `Vector`, `Matrix`, etc.) This is to maintain consistency in your code. Otherwise, you can run into situations where some Vector objects are declared using `Compute` and others with `Vector`, with seemingly no rhyme or reason as to why each was used. In general, the only time you will have to use a specific constructor is if WeBWorK isn't parsing your input into the proper MathObject. For example, you may want to have

`5*3+4` as a Formula object, but WeBWorK is more likely to parse this as a Real object, thus simplifying it down to `19`.

2. Call `Compute` and other MathObject constructors on strings, not straight values. In other words, use this syntax: `Compute("5/3");` instead of this syntax: `Compute(5/3);`. The reason for this is that the former tends to be parsed as a whole string, meaning that WeBWorK won't automatically evaluate it. In this example, it means that you would end up with $\frac{5}{3}$ instead of `1.66666667`. Of course, there may be situations where you want the latter, but in that case, you would be better off simply defining a Perl variable (technically it's not a MathObject, even though it looks roughly the same) with `$answer = 5/3;` and the writing your answer checker as `ANS(Compute($answer)->cmp());`. This evaluates the expression `5/3`, stores it in the `$answer` Perl variable, and then "casts" (temporarily treats) it as a MathObject so an answer checker can be retrieved for it.

Random Values (MathObjects)

Sometimes, you may want to assign a randomized value to a code variable. The benefit of these is that a new random value gets generated for each student who opens the question. Thus, while the question will remain the same between students, the answers they need to provide will not.

Basic Randoms

The syntax for declaring a random value in a .pg file is $a = `random(min, max, step);`, where min is the minimum possible number, max is the maximum possible number, and step is the interval that separates each possible number in the range. The step value can also be omitted, which will cause the randomization function to only deal in integers (essentially defaulting to a step value of 1).

E.g. `random(1, 10, 0.5);` will generate a random number between 1 and 10 (inclusive), with 0.5 steps. Thus, the numbers that could be generated include 1, 1.5, 2, 2.5, etc.

Do-Until Randomization

The syntax for basic randomization can be combined with a code structure called a do-until loop to do things like ensuring that, when generating multiple randoms, none are equal.
Here is the syntax for doing so: `do { $b = random(1, 10); } until ($b != $a);`
Note that we are assuming that a value has already been assigned to $a.

This works as follows:

1. The statement enclosed in curly braces (the “do” part) is executed, generating a random integer between 1 and 10.
2. The statement inside the brackets (the “until” part) is checked. In this case, the value of $b is compared to the value of $a to see if they are not equal to one another. Then, there are two possible outcomes:
    a. $b = $a: If the two are equal to one another, the “until” condition evaluates to false. Thus, things go back to step 1.
    b. $b ≠ $a: If the two are not equal to one another, the “until” condition evaluates to true, both $a and $b keep their values, and the code execution continues on past the do-until loop.

Non-Zero Randoms

Suppose that you want to generate a random value between -10 and 10, but you don’t want the possibility of 0 being generated. You could use a do-until loop to keep generating random values until the value was not equal to 0. However, this is impractical in the grand scheme of things. A better approach is to use the non_zero_random function. This does exactly as it suggests, generating a random number that is not equal to 0. The syntax is otherwise identical to the basic random function, so it’s just `$g = non_zero_random(-10, 10);`.

Randoms from Lists

There are instances where you want to select a random number from a very specific set of numbers; one which can’t be easily defined with ranges and steps. This is where the list_random function comes in. The syntax is `$k = list_random(<values>);`, where <values> is a comma-separated list of numbers. When this function gets called, one number from the list of values will be selected to be returned and used.

Equation Entry

There are two different ways that equations can be formatted: inline and displayed. Inline equations (as the name suggests) are placed inline with the text, while displayed equations are placed on their own line and centred on the page. These require slightly different syntax when entering equations, which will be covered in each section.

PG Equations

For inline equations, use `\(  \)` with the equation (in calculator or LaTeX notation) within the round brackets. For displayed equations, use `\[  \]` with the equation between the square.

PGML Equations

Calculator Notation

To insert inline equations in calculator notation, use the following syntax: `[: :]`, where the equation is written between the colons. To insert the equation as a displayed equation, use: `[:: ::]`, with the equation written between the double colons.

LaTeX Notation

Inline equations are inserted in LaTeX notation with `[` `]`, with the equation between the ticks. This is the commonly-used notation in the various ULeth problems. For displayed equations, use `[`` ``]`, with the equation between the double ticks.

Symbols

Certain problems may require the use of special variables, such as Θ, β, and Δ. Luckily, WeBWorK has an easy method of implementing these. Add the desired symbols to the Context using `Context()->variables->add(symbol=>type(value));`, where `symbol` is the name of the symbol (e.g. delta, tau, etc.), `type` is the variable type (e.g. Real, Formula, etc.), and `value` is an optional parameter that specifies the value the variable is equal to. Then, if students enter the same text you entered for `symbol`, it will be treated as the symbol variable instead of a collection of letters.

Later, if you want to insert special symbols into your problem or solution text, just use LaTeX notation. See here for more information.

Strings

Text strings are very common within .pg files. In most cases, they're used to specify LaTeX equations for display or equations to create MathObjects from. However, in some cases, you may want the answer to a problem to be a string. Now, the syntax for this is quite straightforward: `ANS(str_cmp($answer));` where `$answer` is the MathObject or Perl variable containing your answer.

However, there's one important thing to note with strings: only use them for your answer if both of the following are true:

1. The answer cannot be readily represented by some MathObject.
2. You only want one typing of the answer to be marked correct.

The reason for this is that string answer checkers are much stricter than the regular MathObject checker. The only leeway they really give is that letters in the student's input can be uppercase

or lowercase; the strings get converted to all uppercase before they are compared. Otherwise, any spaces and punctuation marks present in your string must be present in the student's answer for it to be marked correct. For instance, if the answer string was "34 is in the set {3, 4, 34}" and the student entered "34 is in the set {3,4,34}" (i.e. no spaces between the set elements), their answer would be marked incorrect.

Adding Strings to the Context

When you try to call an answer checker on a string, you may run into errors because of the string being undefined. This is because WeBWorK has a small selection of pre-defined strings that exist in different Contexts (e.g. "DNE"), and is only set up to recognize these strings by default. However, there's an easy to way to work around this: simply call `Context()->strings->add("your string here");`. This will add the string to the Context, allowing you to use it in your answer checker. Note that in some cases, you may have to modify the answer checker to use `String` instead of `Compute`; however, only resort to the former if the latter is not working.

Graphing

While it is entirely possible to insert graphs into problems using GeoGebra applets, it's generally unnecessary if you don't need students to interact with the graph. In these situations, it's best to use WeBWorK's built-in dynamic graph functions.

Note: If you are dynamically generating graphs using the code below, you will have to include the PGgraphmacros.pl macro!

Initialization

Before you can start graphing formulas, you need to initialize the graph object. To do this, use `$gr = init_graph(minX, minY, maxX, maxY, origin, gridSize, graphSize);`
To break this statement down:

- `$gr` is the MathObject that will contain the graph object once the initialization is finished. Use it for all future references to the graph.
- `minX` is some integer value that represents the smallest x-axis value that will appear on the graph.
- `minY` is some integer value that represents the smallest y-axis value that will appear on the graph.
- `maxX` is some integer value that represents the largest x-axis value that will appear on the graph.
- `maxY` is some integer value that represents the largest y-axis value that will appear on the graph.

- `origin` is entered in the form `axes=>[int1,int2]`, where `int1,int2` is the coordinate point on the graph that represents where the graph's origin is. In many cases, this will just be 0,0.
- `gridSize` is entered in the form `grid=>[int1,int2]`, where `int1,int2` is the dimensions of the grid that will be overlaid on the graph. Note that it's best to make these even numbers; otherwise the graph alignment can get a bit odd. For instance, using 10,10 will overlay a 10x10 grid on the graph, which can make it easier to judge where points are.
- `graphSize` is entered in the form `size=>[int1,int2]`, where `int1,int2` is the desired dimensions for the graph as a whole. For example, using 400,300 will generate a graph that is 400 pixels wide and 300 pixels tall. Note that this can be adjusted when you actually insert the graph.

After the initialization statement, all you need to add is `$gr -> lb('reset');`, where `$gr` is the name of the graph MathObject you defined in the initialization. This will reset all of the labels on the graph so that you can easily position them where needed.

Labels

Depending on the graph, you will likely want various elements labeled. Sometimes, positioning these properly can require a bit of trial and error, but you should get the hang of it after a little while! The syntax is `$gr -> lb(new Label (xPosition, yPosition, 'labelText', 'colour', 'xJustification', 'yJustification'));` where:

- `$gr` is the name of the MathObject that contains the graph.
- `xPosition` is a floating-point number that determines where on the x-axis the label will be placed. This positioning corresponds to the grid that you defined in the initialization.
- `yPosition` is a floating-point number that determines where on the y-axis the label will be placed. This positioning corresponds to the grid that you defined in the initialization.
- `labelText` is a string that will be the actual text that appears on the graph.
- `colour` is a string representing the colour of the label text. Note that if you want something outside of "basic colours" (e.g. black, white, etc.), you may have to define custom colours.
- `xJustification` is a string that determines the alignment of the text on the x-axis. Permitted values include 'left', 'right', and 'center'.
- `yJustification` is a string that determines the alignment of the text on the y-axis. Permitted values include 'top', 'bottom', and 'middle'.

Label Loops

Obviously, it would be quite inconvenient to have to copy-paste multiple label statements just so you could add usable scales to the graph. Luckily, the foreach loop can help with that!

```
foreach my $i (min..max) {
	$gr -> lb(new Label (<label args>));
}
```

Note that `$i` is a new MathObject; don't use one that you're using for your question! Also, `min` and `max` are positive integers to represent the upper and lower bounds of the scale. Then, `<label args>` is your standard set of label arguments (see above), with two key exceptions. First, for whichever axis the label is on, replace the `Position` value with `$i`. Then, also replace the `labelText` value with `$i`. This will then loop through, placing scale labels at regular intervals along the desired axis, each with an incremented value.

Custom Colours

In certain cases, you may want to make use of colours that are not defined in WeBWorK by default. To define these, you'll need some knowledge of the RGB scheme of defining colours. See this link for assistance. Otherwise, the syntax is quite straightforward: `$gr->new_color("colourname", <RGB value>);`, where `$gr` is the name of the graph MathObject, `colourname` is whatever string you want to use to reference the new colour later on, and `<RGB value>` is the comma-separated RGB value for the colour you want to define.

Adding Functions

Once the rest of the graph is set up, it's time to add your functions! Use `add_functions($gr, "<function> for <var> in <domain> using color:<colour> and weight:<weight>");`.

- `$gr` is the name of the MathObject that contains the graph.
- `<function>` is the formula for the function that you want to graph. Note that if you are using predefined MathObjects in the function, just enter them here normally (e.g. `$A`).
- `<var>` is the name variable in your function (in many cases, this will be x).
- `domain` is a comma-separated list of the lower and upper bounds that `<var>` will be displayed with on the graph. Note that you need to keep the triangle brackets around this one! E.g. `<-5,5>` will graph the function with `<var>` values ranging from -5 to 5.
- `<colour>` is the name of the colour that you want the graphed function to use. Note that if you defined a custom colour, you will enter that name here.
- `<weight>` is an integer that effectively represents how thick the function's line will be when graphed.

Adding Points

If you want to add points to the graph, you can use open or closed circles. Regardless, the overall syntax is the same. Use `$gr->stamps( circle_type( xLoc, yLoc, "colour") );` where `$gr` is the name of the MathObject that contains the graph, `circle_type` is either `open_circle` or `closed_circle`, `xLoc` and `yLoc` are floats for the x and y coordinates of the point, respectively, and `colour` is a string that represents the colour of the point.

Adding Shading

In certain graphs, you may want to define a shaded area. The first thing when doing this is to ensure that the area on the graph that needs shading is completely enclosed by one colour. For instance, if the function line on the graph is blue, you will need an area that is entirely enclosed in blue to add shading (unless you want the shading to just extend off the graph). Then, enter `$gr->fillRegion([xLoc, yLoc, "colour"]);` where `$gr` is the name of the MathObject that contains the graph, `xLoc` and `yLoc` are floats for the x and y coordinates of the starting location for the shading, respectively, and `colour` is a string that represents the colour of the shaded area. Effectively, this will start by shading the point specified by `xLoc` and `yLoc`, before spreading out in all directions until it collides with a line whose colour matches that of the function. In other words, it will cross over other functions and axis labels with no problem, only shading the area used by the function. For this reason, you may need to add a horizontal line along the x-axis if you have a function that doesn't actually generate one itself. See http://webwork.maa.org/wiki/File:GraphShading1.png#.WTc5pK1z2Cg for an example.

Inserting the Graph

In order to keep everything properly formatted when inserting a graph, it is highly recommended that you use a union table. This will ensure that your text doesn't wrap weirdly around the graph image. To do this, first include the unionTables.pl macro. Then, in the problem text:

- If you are using PG, insert `\{ColumnTable(` immediately after the `BEGIN_TEXT` tag. Then, insert your problem text. Afterwards, add a comma, then enter `image(insertGraph($gr), width=>intW, height=>intH, tex_size=>intT).` To clarify, `intW` is an integer representing the width of the graph image on the problem page, `intH` is an integer representing the height of the graph on the page, and `intT` is a percentage multiplier that determines how large the graph will print if a hard copy of the problem is produced. Also, note the period at the end of the line! It's important to make sure you include that in the code! After entering all the information you need for the graph, add `,indent => <indent>,separation => <sep>,valign => "align")\}`, where `<indent>` is an integer representing how much indentation you want in each column of the table, `<sep>` is an integer representing how much spacing you want between the table columns, and `align` is a string representing how you want the contents of the columns to be aligned vertically (e.g. `"TOP"`).
- If you are using PGML, instead of using the `BEGIN_PGML` tag, use `$column1 = PGML::Format(<<END_PGML);`, then add the problem text. Use the `END_PGML` tag afterwards as normal, but then follow it with `$column2 = image(insertGraph($gr), width=>intW, height=>intH, tex_size=>intT).` To clarify, `intW` is an integer representing the width of the graph image on the problem page, `intH` is an integer representing the height of the graph on the page, and `intT` is a percentage multiplier that determines how large the graph will

print if a hard copy of the problem is produced. Also, note the period at the end of the line! It's important to make sure you include that in the code! Lastly, enter `TEXT(ColumnTable($column1, $column2, indent => <indent>, separation => <sep>, valign => "align"));` where `<indent>` is an integer representing how much indentation you want in each column of the table, `<sep>` is an integer representing how much spacing you want between the table columns, and `align` is a string representing how you want the contents of the columns to be aligned vertically (e.g. `"TOP"`).

Answers

One of the most important parts of any math question is the answer, and there are a number of ways that answers can be checked in WeBWorK. The following sections will go over each of these methods. Something to note: keep track of which values (if any) are being randomized in questions.

E.g. If your question is: What is 10 - 5 ?, you can enter your answer value in the code as `Real(5).` However, if the question is: What is 10 - $a, where `$a = random(1, 5)`, then the answer value should be entered into the code as `Real(10 - $a)`, as the answer will differ for each student, depending on what value they receive for $a.

Basic Answer Checkers

The most straightforward answer checker can be used across all files, whether they're formatted using PG or PGML. The syntax is `ANS($answer->cmp());`, where `$answer` is a MathObject reference that contains the value of the answer or a MathObject itself.
You can also specify options for the answer checker. See http://webwork.maa.org/wiki/Answer_Checker_Options_(MathObjects)#.WSX55q1z2Cg for more information on the available options.

There are instances where a basic cmp answer checker will not work, and a variant needs to be used. The variants that are available are arith_num_cmp, frac_num_cmp, fun_cmp, num_cmp, std_num_cmp, str_cmp, and strict_num_cmp. However, to maintain consistency between problems, these should be used sparingly. If you are, say, attempting to set a string as the answer to a problem, and errors keep getting thrown when you try to call use the basic cmp checker, try switching to str_cmp. Just don't rely on this, and try to avoid problem templates that use such evaluation methods.

Eval vs. Cmp vs. Substitute

When evaluating a student's answer, it makes sense that you want to use the cmp function (or its variants) to compare their input with the correct answer. However, what if you want to

actually get the value of some expression? Maybe you want to evaluate a formula in the problem's solution, but do so with values assigned to each variable? Here's where the eval function comes in. The syntax is `$mathObj->eval(var=>"val");`, where `$mathObj` is the MathObject that contains your expression, `var` is a variable that is in the expression, and `val` is the value to make that variable equal to. Note that you can have a comma-separated list of `var=>"val"` if you have multiple variables you need to give values to. You can only call eval if you provide values for all the variables. The result of this code is that the expression will be evaluated to produce a numeric value based on the assigned values for the expression's variables.

Alternatively, you may only want to partially evaluate an expression by substituting a value in for only some of the variables. This is where the substitute function comes in. The syntax is almost identical to eval's: `$mathObj->substitute(var=>"val");`, where `$mathObj` is the MathObject that contains your expression, `var` is a variable that is in the expression, and `val` is the value to make that variable equal to. There are two key differences: you don't have to include all the variables from the equation (only the ones you're substituting for) and the resulting value is a Formula MathObject instead of a single Numeric value.

Tolerance Values

In some questions, it may not be reasonable to expect the student to get the exact same answer that you provide. This could be due to rounding differences, estimations from graphs, and so on. To handle this, you can include a tolerance value in the answer checker. Within the brackets of the answer checker's `cmp()` function, include the following parameters: `tolType=>'type', tolerance=>'float'`, where `type` is either `absolute` or `relative` (depending on the type of tolerance you want to use), and `float` is some floating point number that represents the amount that the student's answer can be "off" by.

Answer Hints

For some questions, you may want to provide students with specialized hints if they enter certain answers. For instance, if a student is supposed to be entering a linear approximation formula, you may want to remind them to not enter equations for horizontal lines as their answer. To do this, you first need to load the answerHints.pl macro. Then, you have to apply the answer hints to the appropriate answer checkers. For the above linear approximation example, for instance, you might use the following:

```
ANS($answer->cmp()->withPostFilter(AnswerHints(
[Formula("1/$a2"),Formula("y=1/$a2")] =>
["Your answer should be an equation for a non-horizontal line.",
replaceMessage=>1])));
```

Basically, the different `Formula()` statements can be replaced by whichever answers you want to address with the hint system. Then, the message string can be replaced with whatever you want to show to the student.

Multiple Answers

Due to a number of factors, certain problems can have multiple answers, or the answers may depend on one another. Now, while creating custom answer checkers is outside of the scope of this documentation (see http://webwork.maa.org/wiki/Custom_Answer_Checkers#.WSYMoa1z2Cg for more information), there are some problems that implement multi-answer checkers. These are functionally identical to the basic answer checkers, however, instead of having a standard MathObject to compare, we use a multi-answer object. These have the same $name format as MathObjects, but actually direct to a function that defines which answers are valid.

E.g. 1: http://webwork.maa.org/wiki/MultiAnswerProblems#.WSYL-61z2Cg
E.g. 2: http://webwork.maa.org/wiki/SeriesTest1#.WSYOkK1z2Cg (Yellow "Set-up" section)

These are implemented in similar, but different ways, depending on whether PG or PGML is being used. Note: Regardless of which is used, you will have to load the "parserMultipleAnswer.pl" macro in the loadMacro function.

It's worth noting that if you want to convert a multi-answer problem to a single-answer one, just remove the parserMultipleAnswer.pl macro, delete the multi-answer checker code (generally, everything in the same code block as `$multianswer = `), and then "undo" the steps below, depending on whether the file is using PG or PGML.

1. PG: Wherever the answer blanks are defined (`ans_rule(int)`), insert the following before each `ans_rule`: `$multianswer->`, where `$multianswer` is the name of the MathObject that points to your multi-answer checker. Then, wherever you are doing your answer evaluation, use `ANS($multianswer->cmp());`, where once again, `$multianswer` is the name of your multi-answer MathObject.
2. PGML: When multi-answer checkers are used with PGML, it's best to follow the syntax outlined in this section. This makes it so that all you need to do is enter `$multianswer` in between the answer curly braces, where `$multianswer` is the name of the MathObject that points to your multi-answer checker.

AnswerFormatHelp

Different people have different ways of writing things, and this certainly holds true in mathematics. While that may not be an issue when a discerning human is marking assignments, it can cause issues when a computer is attempting to do so.

To help remedy this issue, WeBWorK offers the AnswerFormatHelp.pl macro. This macro file gives access to various "input helpers", which appear as links next to answer boxes on problems. Clicking on one of these links will open a window that provides information on how

WeBWorK expects the answer to be entered. This can save the trouble of having to write formatting instructions on every question.

There are an assortment of helpers available, but each is implemented in the same way.

1. In the loadMacros function near the top of your .pg file, add a line at the end that says `"AnswerFormatHelp.pl"`. This will import the macro that provides the needed functionality.
2. This step will differ, depending on whether you are using PG or PGML.
    a. PG: Wherever you would like your help link to appear, add the following: `\{ AnswerFormatHelp("type") \}`, where `type` is the type of help you want to provided (E.g. intervals, inequalities, numbers, etc.); make sure to keep the double quotes around the type!
    b. PGML: Wherever you would like the help link to appear, add the following: `[@ AnswerFormatHelp("type") @]*`, where `type` is the type of help you want to provided (E.g. intervals, inequalities, numbers, etc.); make sure to keep the double quotes around the type!
3. For the full list of available types, please see the pink "Main Text" section at http://webwork.maa.org/wiki/AnswerFormatHelp#.WSXy0K1z2Cg

Graders

WeBWorK allows for the implementation of various "graders", which handle the assignment of grades for each question in a problem set. There are an assortment of different types of graders, and each has has their own uses and setup process.

Average (Default)

If a question has n answer blanks, the average grader assigns each a weight of 1/n. Thus, each part of the answer is worth the same amount. This is the grader that is used by default, so you don't need to do anything special to implement it.

Standard

The standard problem grader is also sometimes called the "all-or-nothing" grader. This is because it only gives credit for the question if all parts are correct; if any are incorrect, no credit is given. This effectively removes weighting from the question, and can be useful for true/false and multiple choice questions that could potentially be solved through trial and error.

To implement the standard grader, first ensure that the PGstandard.pl macro is loaded. In general, this will be loaded for each problem anyway. Then, include the following text in your file (generally in the setup or answer evaluation sections): `install_problem_grader(~~&std_problem_grader);`. Afterwards, you can do your

answer evaluation as normal, and they will automatically be graded with an “all-or-nothing” mentality.

Full-Partial

This grader gives full credit if the last answer is correct, regardless of what the other answers are. However, if the last answer is incorrect, it gives partial credit if any of the previous answers are correct, using the average grader to do so. This can be useful for “show your work” style questions, where there are intermediate steps that can provide the student with partial marks if it seems they were on the right track with their answer.

To implement this grader, first load the PGgraders.pl macro. Then, include the following line of code (generally either in the setup or answer evaluation section):
`install_problem_grader(~~&full_partial_grader);`. After, do your answer evaluation as normal, but make sure that the last answer blank is the one that you want to have as the “full credit” part of the question. It’s also recommended that you include some sort of notice in the problem text to let the student know how the question will be graded.

Weighted

With the weighted grader, each answer in the problem can have its own weight assigned to it, allowing less important parts to be weighted lightly, and more important parts to be weighted heavily.

To implement this grader, first load the weightedGrader.pl macro. Afterwards, immediately following the loadMacros function, type `install_weighted_grader();` to set up the grader. Lastly, you will have to change the syntax for your answer evaluation. Here is the normal answer evaluation syntax:
`ANS($answer->cmp());`
and here is the weighted answer evaluation syntax:
`WEIGHTED_ANS( ($answer)->cmp(), <weight> );`
where <`weight>` is a positive integer value that represents the weight for that answer. Note that all the weights for each problem should sum up to 100. It’s also recommended that you include some sort of notice in the problem text to let the student know how the question will be graded.

Weighted with Credit Answer Option

This grader combines the weighted grader and the full-partial credit grader, making one blank supersede all others (if correct) to award full credit, while the others provide credit based on their weight values in the event that the final answer is incorrect. Note that the full credit answer only awards full credit if all the other answers are either blank or correct.

Implementing this grader requires a bit more work than with some of the others. For starters, load the weightedGrader.pl macro file and install it right after the loadMacros function by entering `install_weighted_grader();`

Next, in the main text of the problem, find all of the answer blanks that are not the "full credit" answer. Instead of the regular answer blank syntax (`ans_rule(int)` or `[____]`), for each of these, change the blank to `NAMED_ANS_RULE('label', width)`, where `label` is some text label in single quotes (which will be used later), and `width` is an integer value to represent the size of the text box. Note that this line needs to be surrounded in `\{ \}` or `[@ @]*`, depending on whether PG or PGML is being used, respectively.

Lastly, in the answer evaluation section, use `NAMED_WEIGHTED_ANS( 'label', evaluator, weight );` for the non-credit answers, and `CREDIT_ANS( evaluator, [list], weight);` for the credit answer. For these two pieces, `label` is one of the labels you defined earlier (whose blank matches the answer we will be checking here), `evaluator` is whichever evaluator you are using for the answer (e.g. `$answer->cmp()`), `list` is a list of the labels for the non-credit answers (whose weights will be overridden if only the credit answer is filled in and correct), and `weight` is a positive integer to represent how much the answer is worth. Note that all the weights should sum up to 100. It's also recommended that you include some sort of notice in the problem text to let the student know how the question will be graded.

Incremental Weighted

Incremental weighted graders (aka "fluid graders") provide more customization for determining how a question will be marked and weighted, based on the number of parts that are answered correctly. For instance, you may want to have a question where if the student gets 0-1 correct answers, they get 0% on the question, 2-3 correct answers awards 40%, and 4 correct answers gives 100%. We'll use this as the example for how to set up an incremental weighted grader.

First, load the PGgraders.pl macro at the top of the file. Then, move to the answer evaluation section, as the rest of the work will be done here.
Insert the following code
`install_problem_grader(~~&custom_problem_grader_fluid);` to switch to the incremental weighted grader.

Next, set up the following statements. Note that everything after the '#' sign in each section is simply a comment to explain what the purpose of the preceding line is.

```
$ENV{'grader_numright'} = [2,4];
# Custom weighting is applied for 2 and 4 answers correct.

$ENV{'grader_scores'} = [0.4, 1];
```

```
# If the first numright condition is met (2 ≤ numright < 4), assign
40% as the grade.  If the second is met (numright = 4), assign 100%.
Otherwise, assign 0%.

$ENV{'grader_message'} = "You can earn 40% partial credit for 2-3
correct answers.";
# Display this message in the problem to explain the marking scheme
to the student.
```

Lastly, set up the answer evaluators as you normally would (e.g. `ANS( $answer->cmp() );`).

Formatting

Infinity Formatting

By default, WeBWorK expects the following formatting when inputting infinite values: -Inf and Inf. Note that these do not have to be capitalized. However, some problem templates may use different formatting, such as 'I', 'Infty', and 'Infinity'. Now, in some instances, this will have no impact on the format expected for student answers. However, this is not always the case. While it is possible to simply include a note in the problem text to explain the input format to the student, this is not always ideal. Plus, the AnswerFormatHelp macro will always specify the Inf formatting, which could get confusing if this is not consistent.

Luckily, this is an easy fix! Just follow the steps below:

1. Open the .pg file for the problem.
2. Find the answer checker that deals with checking for infinity. There should be one of two cases.
    a. If the answer checker line contains the infinity representation (e.g. `"(-I,I)"`), then you can just change the infinity representation to `-Inf` / `Inf` as needed. No big deal!
    b. If the answer checker line instead contains a MathObject (e.g. `ANS(@answers);` or `[____]{#answer}`, then you will have to search the file (you can use Ctrl + F for this) to find the initialization of the MathObject. Once you've done so, you should see the representation of the answer, including the representation of infinity. Simply change the infinity representation to `-Inf` / `Inf` as needed.
3. Make sure to check the problem and solution text so that you can change any input instructions there to reflect the new standard! Alternately, just remove any contradictory "input help" messages in the problem text and use the AnswerFormatHelp macro.

Other “Special” Formatting

In some questions, the original author may have seen fit to use special formatting for various user inputs. For instance, in the process of writing this documentation, a template was discovered that asked students to enter “-1000” instead of “DNE” to represent “Does Not Exist”. Now, obviously there are reasons for each, but it’s worth keeping in mind that consistency can be very important for students, particularly those who are struggling. It may be worth discussing such formatting with other professors so that some general “standards” can be set up for certain inputs.

However, once a format is settled upon (be it department-wide or just in your class), there needs to be a way to update problems to reflect this! Thankfully, this is generally quite straightforward.

1. Open the .pg file for the problem.
2. Find the answer checker that deals with the part of the question you are concerned with.
    a. If the answer checker line contains an actual value for comparison (e.g. `"-1000"` or `"pi"`), then do the following:
        i. Create an answer MathObject somewhere earlier in the file (generally in the Setup section is best). Initialize it with the value you want for the answer. E.g. `$answer = "DNE";`
        ii. Using the instructions in the Answers section, reconfigure the answer checker to use this new MathObject.
    b. If the answer checker line contains a MathObject (e.g. `ANS(@answers);` or `[____]{#answer}`, then you will have to search the file (you can use Ctrl + F for this) to find the initialization of the MathObject. Once you’ve done so, you should see the representation of the answer, where you can change the appropriate values as needed. Then, the answer checker will function on this MathObject as usual. Note that you may need to adjust the type of the value during the initialization. For instance, if the template answer is being set as `"x^2"` (just as a text string) and you want to change it to a formula, you will have to change the initial value to `Formula("x^2");`.
3. Make sure to check the problem and solution text so that you can change any input instructions there to reflect the new standard! Alternately, just remove any contradictory “input help” messages in the problem text and use the AnswerFormatHelp macro.

GeoGebra

GeoGebra is a useful (and free!) application that can be integrated into WeBWorK problems to provide interactive elements for a question. However, it’s only recommended that you use it if you actually need the interactivity. For instance, it is generally for easier to add a dynamically-generated graph to the problem using PG code than it is to add an interactive one

with GeoGebra. Obviously, though, the former cannot be modified by the student to assist them with their evaluation of the question.

At any rate, the general use of GeoGebra is outside the scope of this tutorial. If you would like to learn more, you can follow the tutorials at https://wiki.geogebra.org/en/Tutorials to learn about specific uses for GeoGebra. There's also a tutorial at http://webwork.maa.org/wiki/GeoGebra1#.WTrmDa1z2Cg that details how to make a graph with a slider in GeoGebra. Scroll down to the purple box labelled "Steps to Construct the GeoGebra Applet".

It's worth noting that 3D GeoGebra applets don't like to cooperate with WeBWorK. As a result, this documentation only applies to 2D applications. If you want to integrate 3D GeoGebra applets, you're on your own, unfortunately.

Inserting a GeoGebra Applet

Adding a GeoGebra applet to your question can be a complicated process, however things can be greatly streamlined by starting with a template! Open TemplateQuestions/GeoGebra/GeogebraFundamentalTheorem.pg to get a sense of the basic layout of things. Now, the comments in that file include instructions on how to insert the applet, but we'll just go over some pieces here for clarity's sake. Make sure that you check between the #-ULETH-# tags in the .pg file, though, as it's easy to forget things like changing the applet's width and height!

Once you have created your applet in the GeoGebra application, resize the GeoGebra window to fit the applet and all its data. You can also use the scroll wheel to zoom in, and middle-click + drag to pan around the view. While scaling the window, make note of the dimensions at the top of the GeoGebra window. After you have the applet all ready, you'll need these dimensions for the .pg file to specify how large the applet should be within the question. Also, close off all views and panels that you don't want displayed in your final question.

Next, save the applet, left-click within the Graphics window in GeoGebra (or go to View->Graphics to show the panel), and then press Ctrl + Shift + B. While nothing will appear to happen, this will actually copy a "base64 string" to your computer's clipboard. This string is essentially a text version of your application, detailing all the specifications of your application in a way that the computer can understand. Once you've obtained this string, find the line in your .pg file that starts with `ggbbase64=>`; it should be near the bottom of the file. Then, select everything within the double quotes that follows that statement. Don't select the double quotes themselves! Lastly, hit Ctrl + V to paste the new base64 string in. Make sure that it pastes in a lengthy string of (what appears to be) gibberish! If it doesn't, you may have to go back to GeoGebra and try re-copying the string. Assuming all goes well, though, you should have the applet inserted! Once you open the problem in WeBWorK, the applet should be there!

Known “Gotcha”s

Multiple GeoGebra Applets

For some reason, WeBWorK doesn’t seem to like displaying multiple GeoGebra applets on the same page. If there’s more than one on a page at a given time, the applets will generally be duplicated, so each will show three or four copies of the same applet. Obviously, this isn’t desirable for questions, so it is highly recommended that you limit yourself to a maximum of one GeoGebra applet per question. You can have multiple applets in a particular homework set; they just can’t be on the screen at the same time.

Also, if you have multiple questions involving GeoGebra applets in the same folder, you may have to deal with this issue in the library browser. Just ignore it for now, and as long as there’s only one applet (at most) in each question you add to the homework set, you should be fine!

Updating Applets

There’s an issue with WeBWorK where, if you update the values of a question (particularly if you are passing values from your code into the Geogebra applet), they won’t always show up on the question in the Library Browser. However, if you actually add the problem to a homework set and view it in the set, the values should be updated properly. If not, it’s recommended that you review your .pg code to make sure you’re not overwriting values or using an older version of the file.

Appendix A: Macro File Descriptors

The following is a non-comprehensive list of the different macro files that you can load into WeBWorK problems, as well as their general function. If you are looking for more information, it's recommended that you refer to the WeBWorK wiki, forums, or GitHub.
Any entries **in bold** are used in the ULethProgExamples, ULethTemplates, and/or the ULethQuestionBank as of August 31, 2017.

- **AppletObjects.pl**: Macro-based front end for the Applet.pm module.
- LinearProgramming.pl: Macros for the simplex tableau for linear programming problems.
- LiveGraphics3D.pl: Macros for handling interactive 3D graphics via the LiveGraphics3D Java applet. The applet displays a mathematica Graphics3D object from a .m file.
- **MathObjects.pl**: Macro-based fronted to the MathObjects system. Loads Parser.pl to encourage the usage of MathObjects as a naming convention instead of Parser.
- MatrixCheckers.pl: Provides subroutines for performing more complex answer checking on Matrix MathObjects.
- MatrixReduce.pl: Provides subroutines for elementary matrix computations using Matrix MathObjects. Includes reduced row echelon form, row operations, etc.
- MatrixUnits.pl: Generates unimodular n x n (n=2, 3, 4) Matrix MathObjects with real entries.
- **PG.pl**: Provides core Program Generation Language functionality, including the fundamental macros of the language. Not often loaded on its own, but included in the PGstandard.pl macro.
- **PGML.pl**: Provides core macros for defining the Program Generation Markup Language, which is used for formatting the layout of problem pages.
- PG_module_list: Defines the modules to be used by PGtranslator.
- PGanalyzeGraph.pl: Routines to support the analysis of graphical input from students.
- **PGanswermacros.pl**: Macros for building answer evaluators for a variety of formats (number, string, function, etc.). Not often loaded on its own, but included in the PGstandard.pl macro.
- PGasu.pl: Answer checker that marks any answer as correct. Useful for when you want to leave multiple answer blanks, only some of which will be used.
- **PGauxiliaryFunctions.pl**: Provides additional functions, such as ceiling, floor, max, min, and gcd. Not often loaded on its own, but included in the PGstandard.pl macro.
- **PGbasicmacros.pl**: Defines numerous macros that are mostly used for layout purposes. Examples include line breaks, italics, and bolding. Not often loaded on its own, but included in the PGstandard.pl macro.
- **PGchoicemacros.pl**: Macros for multiple choice, matching, and true/false questions.
- PGcommonFunctions.pl: Implements functions that are common to the new Parser.pm and the old PGauxiliaryFunctions.pl. Includes things like log, sin/cos/tan (and variants), and combinations/permutations.
- PGcomplexmacros.pl: PG language macros for Complex numbers.

- PGcomplexmacros2.pl: More macros for handling multivalued functions of a Complex variable.
- **PGcourse.pl**: Course-specific initalizations.
- PGdiffeqmacros.pl: Macros for Prills 163 problems.
- PGessaymacros.pl: Macros for building answer evaluators for essay-style responses.
- PGfunctionevaluators.pl: Macros that generate function answer evaluators. These take in a function, compare it numerically to a correct function, and return a score. They can require an exactly equivalent function, or one that is equal up to a constant.
- PGgraders.pl: Defines various types of “graders”, such as one that gives full credit to a correct answer and partial credit if only previous parts are correct. Another example is one that gives full credit for five answers correct, 60% credit for four correct, and 0% for three or fewer correct.
- **PGgraphmacros.pl**: Provides an easy ability to graph simple functions.
- PGinfo.pl: Provides macros for determining the values of the current Context in which the problem is being written.
- PGmatrixmacros.pl: Matrix macros for PG. Note that many are very rough at best.
- PGmiscevaluators.pl: Miscellaneous answer macros for radio buttons and checkboxes.
- PGmorematrixmacros.pl: Pretty self-explanatory, no? No clear info as to whether they’re more polished than the ones in PGmatrixmacros.pl.
- PGnumericalmacros.pl: Numerical methods for the PG language, such as plotting a list of points.
- PGnumericevaluators.pl: Macros that generate numeric answer evaluators. These take in a numerical answer, compare it to the correct answer, and return a score.
- PGpolynomialmacros.pl: Provides macros for handling polynomials, including polynomial multiplication, long division, addition, and subtraction.
- PGsequentialmacros.pl: Provides support for writing sequential problems, where certain parts of the problem are hidden until earlier questions are correctly answered.
- **PGstandard.pl**: Loads the standard PG macro packages.
- PGstatisticsGraphMacros.pl: Collection of macros that provides easy access for creating simple statistics graphs.
- PGstatisticsmacros.pl: Statistics- and probability-centric macros.
- PGstringevaluators.pl: Macros that generate string answer evaluators to compare a student string to the correct string.
- PGtextevaluators.pl: Macros that generate answer evaluators that handle questionnaires. These can contain textual answers and radio buttons.
- **PGunion.pl**: Helps with custom formatting of answer blanks and the like.
- Parser.pl: Macro-based front-end to the MathObjects system.
- StdConst.pg: Provides various mathematical constant values.
- StdUnits.pg: Provides various physical units, such as metres, litres, and hertz.
- Value.pl: Declares constructors for the different types of MathObjects.
- alignedChoice.pl: Prints questions with the answer rule at the right, all aligned.
- **answerComposition.pl**: An answer checker that determines if two functions compose to form a given function.

- answerCustom.pl: An easy method for creating answer checkers with a custom subroutine that performs the check for correctness.
- answerDiscussion.pl: Implements discussion-based questions, where a student can provide essay-style answers, and the professor can make comments on those.
- **answerHints.pl**: Answer checker post-filter that allows additional error messages to be produced for incorrect answers.
- answerVariableList.pl - Creates answer checkers that compare the student's answer to a list of variable names.
- bizarroArithmetic.pl: Enables bizarro arithmetic where, for example, 1+1 does not equal 2.
- compoundProblem.pl: Implements a method of handling multi-part problems that only show a single part at any one time. Students can work on one part at a time, and move on to the next part when they get it correct. Earlier parts cannot be returned to.
- compoundProblem5.pl: Defines a macro to create the structure needed to manage scaffolded problems.
- contextABCD.pl: Contexts for matching problems.
    - "ABCD": Answer matches against A, B, C, D, etc.
    - "ABCD-List": Allows entry of lists of strings.
- contextAlternateDecimal.pl: Defines Contexts in which decimal numbers can be entered using a comma rather than a period as the decimal separator.
- contextAlternateIntervals.pl: Defines Contexts in which Interval objects with open endpoints can be specified using reversed brackets rather than parentheses.
- contextArbitraryString.pl: Implements a Context in which the student's answer is treated as a literal string, and not parsed further.
- contextComplexExtras.pl: Adds the ability to include transpose, conjugate transpose, trace, and determinants in student answers in the Complex-Matrix Context, and adds conjugation to all Complex Contexts.
- contextComplexJ.pl: Adds features to the Complex Context that allow both i and j notation for Complex numbers.
- contextCurrency.pl: Implements a Context in which students can enter currency values that include a currency symbol and commas every three digits.
- **contextFraction.pl**: Implements a Fraction object that works like a Real, but keeps the numerator and denominator separate. Provides methods for reducing the fractions, and allowing fractions with whole numbers preceding them.
    - "Fraction": Fractions can be intermixed with real numbers. E.g. 1/2 + .5
    - "Fraction-NoDecimals": Decimal numbers cannot be entered, but they can still be produced from function calls and named constants. E.g. 1/sqrt(2) is allowed.
    - "LimitedFraction": Cannot type decimal numbers, perform operations other than division and negation, or call functions. Must enter a whole number, fraction, or a whole number with a fraction.
    - "LimitedProperFraction": Must enter proper fractions.
- **contextInequalities.pl**: Implements Contexts that provide for inequalities that produce the corresponding Interval, Set, or Union MathObjects.
    - "Inequalities": Both intervals and inequalities are defined.
    - "Inequalities-Only": Only allows inequalities as a means of producing intervals.

- contextInequalitySetBuilder.pl: Implements Contexts that provide for sets described using set-builder notation with inequalities.
- contextIntegerFunctions.pl: Parser Context that adds integer-related functions C(n, r) and P(n, r).
- contextLeadingZero.pl: Enforces whether decimals require a number before the decimal point.
- contextLimitedComplex.pl: Implements a Context in which Complex numbers can be entered, but no Complex operations are permitted. However, operations can still be performed within the real and imaginary parts of Complex numbers.
- contextLimitedFactor.pl: Context file to check that the student's answer agrees in form with a factored polynomial.
- **contextLimitedNumeric.pl**: Implements a Context in which numbers can be entered, but no operations are permitted between them.
    - "LimitedNumeric": Can enter numbers, but can't perform operations between them.
    - "LimitedNumeric-List": Can enter lists of numbers, but can't perform operations between them.
- **contextLimitedPoint.pl**: Implements a Context in which points can be entered, but no operations are permitted between points. However, operations can still be performed within the coordinates of the points.
- **contextLimitedPolynomial.pl**: Implements a Context in which students can only enter expanded polynomials.
- **contextLimitedPowers.pl**: Implements subclasses of the '^' operator that provide various restrictions. These can be:
    - LimitedPowers::NoBaseE(); e cannot be raised to a power.
    - LimitedPowers::OnlyIntegers(); Can only raise to integer powers.
    - LimitedPowers::OnlyPositiveIntegers(); Can only use positive integer powers.
    - LimitedPowers::OnlyNonNegativeIntegers(); Can only use positive integer powers and 0.
- contextLimitedRadical.pl: Defines a root(n, x) function for the n-th root of x, and allows for specification of forms of radical answers, like simplified radicals or with rational denominators.
- **contextLimitedVector.pl**: Allows vectors to be entered, but no vector operations to be performed. Operations can still be performed within the coordinates of vectors.
    - "LimitedVector": Vectors can be entered in either ijk or coordinate format.
    - "LimitedVector-ijk": Vectors can only be entered in ijk format.
    - "LimitedVector-coordinate": Vectors can only be entered in coordinate format.
- contextMatrixExtras.pl: Adds transpose, trace, and determinant to the Matrix Context.
- contextOrdering.pl: Provides a structured way to parse and check answers that are ordered lists of letters, where the letters are separated by greater-than or equal signs.
    - "Ordering": Allows orderings to be defined.
    - "Ordering-List": Allows lists of orderings.
- contextPartition.pl: Allows entry of a partition of an integer as a sum of positive integers.
- contextPercent.pl: Allows for the entry of a percentage values.
- contextPermutation.pl: Allows permutations to be entered using cycle notation. Entries are separated by spaces and enclosed in parentheses.
- contextPiecewiseFunction.pl: Allows the usage of piecewise functions.

- **contextPolynomialFactors.pl**: Provides additional Contexts for dealing with factored polynomials.
- contextPolynomialFunction.pl: Allows only the entry of polynomials, their products, and powers.
- contextRationalFunction.pl: Only allows rational functions and their products/powers.
- contextReaction.pl: Allows for the specification and comparison of chemical reactions. Reactions can be composed of sums of integer multiples of elements, separated by a right arrow.
- contextScientificNotation.pl: Allows entry of answers in scientific notation. Does not allow any operations other than the ones needed in scientific notation.
- contextString.pl: Implements Contexts for string-valued answers.
- contextTF.pl: Implements Contexts for string-valued answers especially for matching problems using True and False.
- contextTrigDegrees.pl: Redefines existing trigonometric functions from radians to degrees.
- extraAnswerEvaluators.pl: Answer evaluators for Interval objects and List objects containing numbers or Point objects.
- **niceTables.pl**: Subroutines for creating tables that conform to accessibility standards, allow a lot of CSS flexibility, and allow some LaTeX flexibility.
- **parserAssignment.pl**: Implements an assignment operator that allows only a single variable reference on the left and any value on the right. Can be used to require students to enter things like “y = 3x + 1”.
- parserAutoStrings.pl: Forces String() to accept any string as a legal value.
- parserCustomization.pl: Placeholder for site/course-local customization.
- **parserDifferenceQuotient.pl**: Implements an answer checker for difference quotients as a subclass of the Formula class.
- **parserFormulaAnyVar.pl**: Formulas can be declared using any letter as their variable. This also goes for student answers, meaning their variable name does not have to match the one in the question.
- **parserFormulaUpToConstant.pl**: Implements formulas “plus a constant”.
- **parserFormulaWithUnits.pl**: Allows formulas to include units of measure in them.
- **parserFunction.pl**: Simplifies the creation of new functions to add to the current Parser Context.
- parserFunctionPrime.pl: Implements prime notation for derivatives for functions added to the Context via a parserFunction() call.
- **parserImplicitEquation.pl**: Implements an answer checker for implicitly-defined equations. Looks for zeroes of the equation and tests that the student and professor equations both have the same solutions.
- parserImplicityPlane.pl: Implements implicit planes.
- parserLinearInequality.pl: Implements implicit open or closed half planes.
- **parserMultiAnswer.pl**: Ties several blanks to a single answer checker so the answer in one blank can influence the the answer in another.
- **parserNumberWithUnits.pl**: Implements a number with units of measure.
- parserOneOf.pl: Allows students to answer any one of several correct answers. The correct answer will list all the possibilities when shown.
- **parserParametricLine.pl**: Implements formulas that represent parametric lines.
- parserParametricPlane.pl: Implements parametric planes in 3D.

- **parserPopUp.pl**: Implements a pop-up menu object that is compatible with MathObjects, the MultiAnswer object, and PGML.
- parserPrime.pl: Defines a prime operator (‘) to perform differentiation.
- **parserRadioButtons.pl**: Implements a radio button group object that is compatible with MathObjects, MultiAnswer objects, and PGML.
- parserRoot.pl: Allows a C<root(n, x)> function to be added to any Context to perform the n-th root of x.
- **parserSolutionFor.pl**: Answer checker that checks if the student’s answer satisfies an implicit equation.
- **parserVectorUtils.pl**: Utility routines that can be useful in vector problems.
- parserWordCompletion.pl: Provides free response, fill in the blank questions with interactive help. As students type their answer, an auto-complete drop-down list of allowable answers will be generated. Also generates a warning message if an answer not in the list is submitted.
- problemPanic.pl: Allows for “panic buttons” that can be used to get additional hints, at the cost of a portion of the score for the question.
- problemPreserveAnswer.pl: Allows sticky answers to preserve their special characters.
- problemRandomize.pl: Reseeds a problem so that a new random version is generated for a particular student.
- sage.pl: Provides functionality for calling a Sage cell server.
- **scaffold.pl**: Provides the ability to make a single problem file contain multiple parts, where the later parts aren’t visible until the earlier ones are completed. The author has control over which parts are allowed to be opened, and which are showing, but does not have to keep track of what answer blanks go with which sections (as they did with the early compoundProblem macros).
- text2PG.pl: Sanitizes a text string for use with TEXT and EV3 so that non-math text is properly displayed in HTML mode and TeX mode.
- unionLists.pl: Functions used to make unordered and ordered lists in HTML files and corresponding lists in TeX mode.
- **unionTables.pl**: Functions for creating tables of various kinds.