WeBWork Problem Creation - Tutorial

February 18, 2021

Contents

1 Introduction

WeBWork is an online homework and assessment open source delivery system used for Mathematics and Science. It is supported by the Mathematical Association of America and it provides flexible and instant feedback to students. WeBWork has access to a Open Problem Library with more than 20.000 problems across several fields in Mathematics. Many of these problems have been creating under the philosophy of replicating other's work; however, due to the huge amount of problems to browse, it is usually hard to find the appropriate problem to use as a template in case that a custom or new problem wants to be included in a professor course.

The aim of this tutorial is to provide a basic overview on the WeBWork problem creating art accessible to as many enthusiastic as possible. Since the WeBWork problem files are based in a language derived from PERL, we recommend that the author familiarizes with the basic syntax of this language; it is also preferred that the reader has some programming experience in any language, however this is not mandatory. This tutorial is not intended to replace or extend what is already found in the WeBWork official wiki nor include advanced techniques that the most proficient PERL and WeBWork programmers provide, also it does not provides information about how to use the WeBWork interface, so it is also assumed that the reader is familiar on how to navigate through WeBWork and access the source code of the problem files.

The first part of this tutorial is about the structure of a WeBWork problem file, the most basic types of problem that are used in WeBWork, and how to use answer graders and evaluators. We also discuss one of WeBWork's greatest strengths: randomization. The second part of this tutorial covers more advanced techniques and the integration of WeBWork with other open source software such as GeoGebra. The contents of this documents are compilations from the WeBWork wiki, the WeBWork Forum and the Author's work as a Teaching Fellow at the University of Western Ontario. There are companion files to this document that can be openly used by the reader.

2   WeBWork Problems Layout

Any WeBWork problem is rendered using a backend based in PERL. If you want to immerse into the WeBWork problem creation career, I strongly recommend to read the first 6 Chapters of the PERL camel book [?] if you are not familiar with this programming language syntax. Knowing basic pseudo-language programing is useful but not ncessary to create WeBWork questions. WeBWork problem files use the extension `.pg`.

Let us first discuss the layout of any WeBWork problem. It mostly look like the following file:

```
1
2  ##############################################
3  #Initialize the problem
4
5  DOCUMENT();
6
7  #load WeBWork macros
8
9  loadMacros(
10 #type the macros name here
11 );
12
13 #To display the marks of the problems
14 TEXT(beginproblem());
15
16 #To display a title in the problem
17 Title("Write your title here");
18 ##############################################
19
20 #Do you Perl/Webwork computations here
21
22 #Block that is rendered and displayed into the webwork frontend
23 BEGIN_PGML
24
25 # Type the question and answer here
26
27 END_PGML
28
29 #answer evaluators if any
30
31 # Block that renders the solution/explanation of the problem,
32 BEGIN_PGML_SOLUTION
33
34 # Type the explanation here
35
36 END_PGML_SOLUTION
37
38 ENDDOCUMENT();
39 #Finalizes the problem
```

File 1: Initial

Key notes:

- Every WeBWork problem must start with the line `BEGIINDOCUMENT();` and finish with `ENDDOCUMENT();`.

3

- Load the macros inside the enviroment `loadMacros(macro1.pl,macro2.pl,...);`. Macros are commmonly standard and will be provided in the templates acompaning this document. You can also load custom macros placed inside the local folder `macros`. A description of the most used WeBWork macros is discussed in this resource.

- The `BEGIN_PGML - END_PGML` block contains what is going to be displayed to students, as well as the way of recording students answer.

- Outside the PGML blocks, PERL-WeBwork code is used to set up the problem's, variables, functions, numbers, etc,

- The `BEGIN_PGML_SOLUTION` block contains the explanation to the correct solution when is displayed to students (usually after they attempt the problem on their own and past a due date). This section is optional, include it only if you want to display an explanation to the solution.

3 WeBWork Syntax

In this tutorial we use the PGML syntax to create WeBWork problems instead of the traditional/classic syntax `BEGIN_TEXT` that several problems from the Open Problem Library use. When writing WeBWork problems, there are two main formats to consider:

Perl Syntax: It is used in the main body of the problem, outside the PGML blocks, it is mostly similar to the PERL syntax.

PGML Syntax: It is used to render and display the problem in the WeBWork interface. See the PGML cheat sheet for a nice summary on how to type using PGML. We highlight that LaTeXsyntax is enclosed between [` `], and to record and evaluate student's input we use the syntax `[_]{$answer}` to display an answer box, and compare it with the variable `$answer`.

It is also possible to just record the answer and evaluate the input outside the PGML block. It uses the syntax `[_]` to create and answer box and `ANS($ans->cmp());` outside the PGML block to evaluate student answer. This syntax is important to keep in mind as it is useful when parsing options to the evaluator or using a custom one.

Now we are going to walk you through the very basic steps to create your first WeBWork problem. It is a very simple problem but it illustrate the basic syntax that the source file must have. The problem that we are going to write from scratch is the following.

Figure 1: Very basic WeBWork problem

To write this problem, you can use the template file 1, and a full code to compare with your own can be found in the companion file `Problems/Introduction/myfirstproblem.pg`. Follow these steps to create this problem:

1. Initialize your file with `BEGINDOCUMENT();`

2. Load the macros `"PGstandard.pl"`, `"PGML.pl"`, `"PGunion.pl"`.

3. Type the line `TEXT(beginproblem());`. This line displays the marks worth for this problem in students homework.

4. We will use PERL variables to denote the numbers in our problem; namely, we will store the numbers -3 and 2 in the variables a and b respectively. And we will store the answers $a + b$ and $a * b$ in the variables $ans1$ and $ans2$.

5. Initialize the numbers using the syntax `$a = -3;` and `$b = 2;`. Notice that PERL variables start with the dollar sign `$`, and all lines end with semi-colon `;`.

6. Store the answers in the variables `$ans1 = $a + $b;` and `$ans2 = $a*$b;`.

7. Start a PGML block with the syntax `BEGIN_PGML` and `END_PGML`.

8. Type the display text inside the PGML block. To display $-3 + 2$ use the syntax `` [`[$a] + [$b] `] ``. Recall that `` [` `] `` encloses TeX syntax, and to display a PERL variable, say `$a`, use `[$a]`.

9. Create a student input box with the syntax `[_]`. The number of underscores used determines the length of the input box.

10. To evaluate the input associated to the box, and compare it with the variable `$ans1`, type `{num_cmp($ans1)}` besides the input box. Therefore, the syntax to record and compare student's input with the correct answer $a + b$ is `[_]{num_cmp($ans1)}`. Notice that we are using the numeric comparison evaluator `num_cmp`. A detailed discussion about WeBWork common evaluators is found in upcoming section.

11. Now create a solution PGML block with the syntax `BEGIN_PGML_SOLUTION` - `END_PGML-SOLUTION` and use the same syntax as in a PGML blocl.

12. For example, to display the first solution, type `` [`[$a]+[$b] = [$ans1]`] `` or any explanation of your choice.

13. Finish the problem with the line `ENDDOCUMENT();`.

4 Basic Question Types

In this section we introduce the basic question creation in WeBWork. We cover basic numeric and text input, and closed answer questions (i.e. those type of questions where students needs to select the correct answer from a given list of possible answers). Since we are starting from scratch, we won't cover advanced techniques or discuss answer inputs in detail until later in the upcoming sections.

These templates can be used in questions with layout of Multiple choice, True-False, Matching Lists and others. We discuss the layout that WeBWork offers when creating these type of problems. It is important to remark that you can feature several of this types in the same problem, in case that a problem with several subquestions wants to be implemented

4.1 Numeric Input and num_cmp() evaluator

These questions require student to enter the numeric answer to a computation. For example, The companion file is `Problems/Tutorial/BasicTypes/numeric.pg`.

Key notes:

- The answer evaluator used is `num_cmp`. It compares student input with the correct answer as PERL numbers.

Numeric Entries

Evaluate the function $f(x) = x^2 + x - 1$ at $x = 3$.

- Answer: ☐

Solution:

Figure 2: Template problem to use numeric input

- Options can be parsed to the evaluator. See the Wiki for the possible options added. For example, `num_cmp($answer, mode=>'strict')` requires students to enter the exact number, and no functions or arithmetic symbols can be entered.

4.2 String input

String based questions require student to type their answers inside an answer box. These type of questions are commonly used when students face True/False questions, and they are asked to enter `T` or `F`. An example exhibiting this situation is the following: The companion file is `Problems/Tutorial/BasicTypes/true-false.pg`.

True/False Questions

Mark the following statements as either True or False by entering T or F respectively. T

- ☐ Every differentiable function is continuous.

- ☐ The function $f(x) = x^{1/3}$ is everywhere differentiable.

Solution:

Figure 3: Template problem to use true/false question

Key notes:

- The answer evaluator used is `str_cmp`. It compares student input with the correct answer as string, capitalization ignored; that is, `t` and `f` are also valid inputs. Any other input will be marked as incorrect.

- Options can be parsed to the evaluator. See the Wiki for the possible options added.

4.3 Radio Buttons

Radio Buttons are useful when a multiple choice question with unique answer wants to be displayed. The student must select and option from a list of possible answers followed by a clickable button. When the answer is selected by clicking on it the radio button will be filled to display the selected choice. For example:

The source code for this problem is `templates/Problems/Tutorial/BasicTypes/multiple-radio.pg`.

Key notes:

Multiple choice - Unique answer

To complete a problem with multiple choice and a unique correct answer, select the answer by clicking and selecting the corresponding radio button.

For example, Which of the following numbers is even.

○ A. 1

○ B. 4

○ C. $x^2 + x + 1 = \cos(x)$

○ D. 7

◉ E. None of the above

Figure 4: Template problem to use radio buttons

- It uses the macro `parserRadioButtons.pl` to load the radio buttons environment `RadioButtons`.

- The environment `RadioButtons` has the following arguments: `RadioButtons([choices,...],correct,options);`.

- Choices (First Argument): Listed inside right brackets [,]. A randomizing list can be included by listing the choices using a second right-bracketed list.
  For example, in `["Choice 1", ["Choice 2", "Choice 3", "Choice 4"], "Choice 5"]` , Choice 1 will always appear first and Choice 5 last when rendered the problem, whereas Choice 2,3,4 will appear in a random order.

- The choices must be enclosed in single or double quotes. If TEX input is being used, enclose using `\(    \)` or `` `    ` ``.

- Correct option (Second Argument): Denote the index of the correct choice in the main choices list argument, starting at 0. For example, if this value is set to `3`, the correct option will be `Choice 4` (randomization is ignored). The correct option can also be expressed using the exact same string used for the choice; that means that also entering `"Choice 4"` in the second argument is a valid way of setting up this choice as the correct one.

- Options (Third Argument - Optional). The arguments follow the layout `argument => value`, and they are:

  - `labels` : Determine what labels are displayed besides each radio button (default: no label is displayed), and also the label is displayed when the answer is previewed or submitted (default: it is displayed `Choice X` where X is the number of the position of that choice in the student list) The values are : `"ABC"` (capital letters are displayed), `"123"` (numbers are displayed), `"text"` (the actual choice is displayed), `["custom1", "custom2",...]` (custom labels are displayed).

  - `displayLabels` : If `1` (default), the labels set up above appear besides the radio button (and in the answer preview). If `0`, the labels do not appear besides the radio buttons (but they do appear in the answer preview).

  - `separator` : Specifies what text is put between two radio buttons. Default: `"$BR"` (single line break). It is suggested to use this option with value `"$BR $BR"` (double line break) when displaying math formulas in the choices.

  - `checked` : Specifies the index (or string) of the choice that is automatically selected when the problem is rendered. If this option is not used, no choice is selected when initializing.

  - `uncheckable` : If 1, the selected option can be unchecked when clicked again (so no option is selected). If 0 (default), the selected button can not be unchecked. *Do not confuse with locking up an option. Students are able to select and change the answer before submiting, just controls if the state of "no button selected" can be displayed*

4.4 Dropdown List

Drop down menu list are also used when a multiple choice problem with a unique answer wants to be implemented. An example of this is as follows: The companion file is `Problems/Tutorial/BasicTypes/dropdown.pg`.

Figure 5: Template problem to use a drop down menu

Key notes:

- It uses the macro `parserPopUp.pl` to load the radio buttons environment `PopUp`.

- The environment `PopUp` has the following arguments: `PopUp([choices,...],correct);`.

- Choices and the correct answer works in a similar fashion as in RadioButtons

- TEX input can not be displayed into the menu items (due to HTML restrictions), so avoid to use this format when the options require to display mathematical expressions.

4.5 Checkboxes

Checkboxes are mostly used when a multiple choice question admits several correct answers. Students are asked to select the correct answers by selecting the checkbox besides each choice. For example,

Figure 6: Template problem to use checkboxes

The companion file is `Problems/Tutorial/BasicTypes/checkbox.pg`

Key notes:

- It uses the macro `PGchoicemacros.pl` to load the checkbox environment `new_checkbox_multiple_choice()`.

- Enter the question statement and their correct answers using the PERL assignment

  `qa(Question, CorrectAnswer1, CorrectAnswer2,...)`.

- Enter the incrrect options using the assignment `extra(Incorrect1,Incorrect2,...)`.

- WeBWork will display all the correct and incorrect options in a random order. Display the statement using `[@ $mc -> print_q() @]*` and the possible answers `[@ $mc -> print_a() @]*`.

- If you want to display an incorrect option at the bottom ignoring the randomization, use the assignment

  `makeLast(...,almostLastIncorrect, LastIncorrect)`.

- TEX input can be displayed anywhere by using the enclosings \(\).

- At the end of each option, use `$BR` for line break or `$BR $BR` for a double line break (creates an empty line between choices).

- This type of questions does not grant partial marks to students. They must check ALL correct answers to be granted full marks, otherwise they will not be granted any credit. Partial marks for checkboxes are discussed in Section 7.5

5 Mathobjects and Contexts

WeBWork uses two main inputs when recording and evaluating students answer depending on how the problem is coded. These two main type are Perl scalars and Perl objects; the latter in WeBWork are called "MathObjects". In this section we cover features of some Mathobjects (functions, vectors, matrices, fractions among others) and the evaluators that need to be loaded alongside.

If you use Perl scalars (defined as variables using the dollar symbol $) in your WeBWork problem, you need to use the `num_cmp()` or the `str_cmp()` evaluators to compare student's answer depending on whether a number or string is being evaluated. The `cmp()` evaluator is used for Mathobjects, and it includes more features as the possibility to add custom feedback messages or weights to student's answer as discussed in Section 5.

Perl scalars can be made into MathObjects by using the syntax `Compute("")`;. For example, `Compute("1")` or `Real("1")` makes the number 1 as a Mathobject. In this case the answer must be parsed using `cmp()` and not `num_cmp()`,

Contexts are WeBWork libraries for specific MathObjects, they include constants (numeric or strings), functions and variables. These context can be loaded in specific problems situation as they have particular features and limited computer algebra abilities. For example, you can take derivatives in WeBWork, or compute the inverse of a Matrix. Contexts are loaded by adding the line `Context("")`; where the context name is typed inside the quotes.

Reference for introduction to contexts: here.

The most common contexts in WeBWork are the following:

- Numeric: It is the basic context, besides constants and functions, it includes Formulas (mathematical functions that can be evaluated, differentiated, etc.), Lists (useful when asking students several correct answers in the same input box).

- Complex: It is the numeric context plus the ability to manipulate complex numbers.

- Point: Numeric context plus the manipulation of coordinate points in $\mathbf{R}^n$.

- Vector: It is the Point context but also includes features for vectors. Useful in linear algebra problems.

- Matrix: It is an extension of the Vector context that also contains matrix support.

- Complex-Point, Complex-Vector, Complex-Matrix: Extension of the complex context by the appropriate objects.

- Intervals: Allows the manipulation of intervals (and finite sets). Useful when asking inequality solutions or domain/range of real valued functions. "Infinity" is supported.

- Full: includes all previous contexts features.

There are more specialized contexts and it is also possible to create new ones.

5.1 Functions available in WeBWork

When using contexts, WebWork allows both problem authors and students to use several mathematical symbols and functions when writing problems and entering answers. A comprehensive list of the available tools are found in the WeBWork wiki. In summary, they include the basic arithmetic operators, factorials, mathematical constants, square roots and absolute values, trignometric, natural exponential and logarithmic functions.

If you want to disable or undefine operators or functions for student's input, visit this resource or this one.

Key Notes:

- If you want to disable all functions but just a couple, it is easier to disable `All` and enable the ones that you want to allow.

- You need to parse your answers as MathObjects, as the functions are enabled/disabled for these type of objects. Perl variables and answers can be still allowed to use these functions even though you disable them in your source code.

5.2 Adding Constants and Strings to the Answers

Resource

6 Introduction to Question Randomization

One of the most powerful features of using WeBWork is the ability to randomize questions: the order that choices are displayed, the internal variables or even the complete statement that leads to different outcome. In this tutorial we will focus on the randomization that can be achieved internally from the source code; randomization that uses the WeBWork frontend is going to be (superficially) covered in another section.

6.1 Randomization of constants - Variables

Let us consider the following question.

Find the intercept of the line $3x - 4y = -4$ with the x-axis.

This question has a straightforward solution, set $y = 0$ and obtain the answer $x = -4/3$. To add randomization to this question, we might start by working with a general line equation $ax + by = c$ and choose randomly the values for a, b, c; there is also an important requirement, that $a \neq 0$. To randomize these constants, we are going to use two WebWork functions in our code: `random(min,max,step)` and `non_zero_random(min,max,step)`. The description of these two functions are clear: select randomly a number between the given range [min,max] following the entered step; the second function is the same plus the condition that the number selected must be non-zero (if it can be possibly selected from the given range). The step argument is optional: if empty the default step is 1. Any number is allowed as an argument.

For example, we can set up our variables in the problem above:

```
1  $a = non_zero_random(-5,5);
2  $b = random(-3,3);
3  $c = random(-6,6,2);
```

And the answer is also systematic: answer $= c/a$. So we can then create our answer variable `$ans = $c/$a`. Since we are possibly dealing with fractions, students can enter `-4/3` as a valid answer, or can enter a decimal. If you use just `num_cmp($ans)`, students would need quite some decimals in the description of the fraction $-\frac{4}{3}$. You can set up a tolerance of a given number of decimals by parsing the option `tol => t` where t is the desired tolerance. For example, to accept correct answers up to two decimal places of approximation in our example, modify the evaluator as `num_cmp($ans, tol => 0.01)`.

Task: Write the Perl code for this question featuring randomization of the coefficients.

The basic randomization for Perl variables are:

1. `random(min,max,step)`.

2. `non_zero_random(min,max,step)`

3. `list_random(element1,element2,....)`

Now suppose that you want to modify the problem, so students can get either the intersection with the x-axis or the intersection with the y-axis (and thus the approach for the solution will be different). Let us discuss the code to create the following (possible) version of our question.

Find the intercept of the line $4x - 2y = 6$ with the y-axis.

The main changes in the source code are the following:

```
1  $a = non_zero_random(-3,3);
2  $b = non_zero_random(-10,10);
3  $c = random(-6,6,2);
4
5  @axis = ('x','y');
6  @ans = ($c/$a, $c/$b);
7
8  $k  = list_random(0,1);
9
10 BEGIN_PGML
11
12 Find the point of intersection of the line [`[$a]x+[$b]y = [$c] `] with the [`[$axis[$k]]`]-axis.
13
14 - Answer: [_]{num_cmp($ans[$k], tol => 0.01)}
15
16 END_PGML
```

Key notes:

- We make sure that both a and b are non-zero.

- We create the Perl array `@axis` containing the strings `x` and `y`.

- We create the Perl array `@ans` containing the numbers c/a and c/b.

- We use the variable `$k` to select an index at random from our arrays: If $k = 0$, the x-axis intersection is going to be asked. If $k = 1$, the y-axis intersection is asked.

- We use the string that contains the axis in the PGML block by using `$axis[$k]`. Similarly, the entered answer is compared with the correct answer `$ans[$k]`.

6.2   Randomization of Variables with Conditions

You can use control structures (if, while, do...) to randomize constants under a condition. A common technique is to use the `do - until` control structure. We will exemplify it in the next situation: *We want to define in WeBWork an invertible random 2×2 matrix with real entries* $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$. This can be achieved with the following syntax

```
do {
$a = random(-5,5);
$b = random(-5,5);
$c = random(-5,5);
$d = random(-5,5);
} until ( $a*$d - $b*$c != 0 );

Context("Matrix");

$M = Matrix([$a,$b],[$c,$d]);
```

Several conditions can be added using `AND, OR` for example in the `until` structure.

6.3   Randomization of Question and Answers - Array Randomization

Suppose that you have a list of n-questions with their respective answers. In this section we discuss an approach to write a question where k-questions are displayed in a random order. The key is to use the function `NchooseK(n,k);` this function returns a subarray of length k with entries between 0 and $n-1$ (in a given order). Requires to load the macro `PGchoicemacros.pl`.

We assume that there are two arrays `@questions = (question_1,questiion_2,...,question_n)` and `@answers = (answer1,answer2, ..., answer_n)`. We start by selecting a slice of the questions and answers to use:

```
$n = $#questions  # total number of questions
$k = k # number of questions to use


@slice = NchooseK($n,$k);
@chosen_q = @questions[@slice];
@chosen_a = @answers[@slice];
```

We have created two arrays of chosen questions and answers. We can use a `foreach` statement to write the questions:

```
foreach $i (0..1) {

BEGIN_PGML
[$i+1]. [$chosen_q[$i]]**
```

```
5
6  [_]{$chosen_a[$i]}
7
8  END_PGML
9  }
```

Let us see this in action in the following example, two possible questions generated by the same source code (File: `Problems/Tutorial/randomization/random-question.pg`).

For each of the following statements, determine whether it is **True** or **False** by selecting the appropiate option.

1. Any differentiable function is continuous

   ○ True

   ○ False

2. The domain of the $\ln(x)$ is $\mathbb{R}$

   ○ True

   ○ False

Figure 7: Randomization of Question: Variant A

For each of the following statements, determine whether it is **True** or **False** by selecting the appropiate option.

1. The function b^x is increasing for any value of $b > 0$

   ○ True

   ○ False

2. The domain of the function $\sqrt{x}$ is $[0, \infty)$

   ○ True

   ○ False

Figure 8: Randomization of Question-Answers: Variant B

6.4   Randomization of Problems

Another technique to include randomization on student's assignments or assessments is to select a random problem from a list of similar problems. For example, if there are 5 problems about "integration by parts" that you have created, and each problem features internal randomization, you can ask WeBWork to select randomly one of those problems when generates a version for each student.

WeBWork already has a method to do this for gateway-quizes in the official Wiki here. However, we will provide a different technique to achieve this that works for both regular homework and gateway quizzes. It also has the advantage that does not require to create a WeBWork homework set with the problems that you want to use. We need to use the file `macros/selectproblem.pl` that needs to be added to your local macro folder.

The use of this macro is the following:

- Place the all the problem files that you want to use as your pool in the same folder. For example: `/by-parts/problem01.pg,`

- In that folder, create a `pg` file with the following contents:

```
1   DOCUMENT();
2   loadMacros("selectproblem.pl",);
3   includeRandom("problem01.pg", "problem02.pg", "problem03.pg",...);
4   ENDDOCUMENT();
5
```

  Name the file something convenient in each situation; for example, `random-by-parts.pg`.

- Add as many problems as you want in the `includeRandom` function, as long as they are inside the same folder.

- When creating a homework/assessment, add the problem `random-by-parts.pg` (or the name that you used) to it so each student will have a randomly selected problem from the one listed.

- If this randomizing file is added more than once in a homework, different problems are going to be selected. If you add more this question more times than the number of available problems, an error will be raised on student's end.

7 Answer Graders and Checkers

7.1 Answer Graders

Assume that in our WeBWork problem there are two or more questions (or sub-questions). By default, WeB-Work assigns each sub-question the same weight; namely,(1/total_number_of_questions). However, depending on the particular situation of the question and/or assignment, you have three main options when grading student submissions:

- **Standard Grader:** No partial credit is awarded; student is granted full marks if ALL their answers are correct. It is used by adding the line `install_problem_grader(~~&std_problem_grader)` to the WeBWork problem initialization.

- **Average Grader (Default):** Partial credit is awarded, and each sub-question is equally weighted (1/to-tal_number_of_questions). It is used by adding the line `install_problem_grader(~~&avg_partial_grader)` to the WeBWork problem initialization.

- **Full-Partial Grader:** Partial credit is awarded; each sub-question is equally weighed (1/total_number_of_questions). Also full credit is awarded if the LAST answer is correct no matter what the previous sub-answers were. It is used by adding the line `install_problem_grader(~~&full_partial_grader)` to the WeBWork problem initialization.

7.2 Weighted Partial Grader

This grader awards partial credit to sub-questions but the weight is custom defined. Its use is a bit different than the previously discussed. Start by adding the macro `weightedGrader.pl`. and the line `install_weighted_grader()` to the WeBWork problem body. After getting student's answer, parse the weighted grader by using the syntax `WEIGHTED_ANS( ($answer)->cmp, X );` after the PGML block. Here `$answer` denotes the variable with the correct answer and `X` is the weight as a percentage (integer number between 0 and 100). Notice that all you weights must add up to one. For example, if we want to assign weights of 25%, 25% and 50% to the sub-questions a),b) and c) respectively in our example. We use the `num_cmp` parser to evaluate the answer.

```
1  WEIGHTED_ANS( num_cmp($ans1), 25 );
2  WEIGHTED_ANS( num_cmp($ans2), 25 );
3  WEIGHTED_ANS( num_cmp($ans4), 50 );
```

The weighted answer checker can also be used inside a PGML block by using the syntax `[_]{weighted_ans($ans->cmp, X)}`.

7.3 Labelled Answer Rules and Other Techniques

Labelled answer rules are used when you want to assign a label or a name to a particular student answer. This can be combined with Weighted sub-questions to create a different grading techniques For example; creating a weighted full-partial graded problem.

A label is created by using the rule `NAMED_ANS_RULE()` when collecting student answer. The syntax inside the PGML block is the following: `Answer = [@ NAMED_ANS_RULE("label", t) @]*` where `"label"` is the custom name that you want to label to that answer, and t is an integer number that denotes the length of the answer box displayed. To evaluate and grade student input, we use the following syntax (outside the PGML block) depending on the situation:

- Use `NAMED_ANS("label",$answer->cmp())` to compare the answer labelled `label` with the answer `$answer`, and when the standard, average or full-partial graders are loaded.

- Use `NAMED_WEIGHTED_ANS( "label" => $answer->cmp(), X );` to use the weighted grader; recall that `X` denotes the weight for that answer.

- Use `CREDIT_ANS( $answer->cmp(), ['label1','label2'], X );` to grant

  `X + marks_for(label1) + marks_for(label2)` marks

  if student gets correctly this answer, no matter if answers for `label1` and `label2` are correct or incorrect. This can be used to create a full-partial weighted answer problem. This checker can also be used inside a PGML block by using the syntax `[_]{weighted_ans($ans->cmp, X, ['label1','label2'] )}`.

7.4 Unordered Checker

This is an alternative to using Lists in MathObject contexts, but it can be useful when students need to input several answers and the order they enter their answers do not matter. For example, let us code the following question:

Determine the irreducible monic factors of the polynomial $x^3 - 2x^2 - 2x - 3$

Our example will look in WeBWork like this:

Determine the monic irreducible factors of the polynomial $f(x) = x^3 - 2x^2 - 2x - 3$.

Answers: ☐, ☐, ☐.

Note: If there are less than three factors, leave the remaining answer boxes type NONE.

Solution:

Figure 9: Unordered students answers

In our example, the factors are $x^2 + x + 1$ and $x - 3$. Students need to enter `x^2+x+1, x-3, NONE` in any order.

Refer to the source code `Tutorial/answers-checkers/unordered.pg` and to the key notes below in the usage of the unordered answer checker.

Key Notes:

1. Load the macro `unorderedAnswer` to load the answer checker `UNORDERED_ANS()`.

2. The answer checker has format `UNORDERED_ANS($answer1->cmp(), $ans2->cmp(), ...)` to compare the correct answers with student's input.

3. Weights can be added by using the `weighted_ans()` syntax.

4. All answers must be parsed as **MathObjects** if using the `cmp()` checker: Need to use `Compute(" ")` for strings and numbers if they are also used in the answers. (Note: Strings must be defined in your context).

5. If all answers are either numbers or strings, you can use the syntax `str_cmp($ans1), str_cmp($ans2),...` or `num_cmp($ans1), num_cmp($ans2),....`

6. Do NOT mix answers checkers (`cmp, num_cmp, str_cmp`) inside `UNORDERED_ANS` or an error will be raised.

7.5   Partial Credit on Checkboxes

The checkbox environment does not support partial credit: Students need to select all correct answers in order to get full marks, otherwise they will get no credit at all. This might be a bit unfair depending on the situation, so here we discuss a work around to this issue. First, it is needed to download the custom macro `checkbox-partial.pl` and load it into your checkbox problem (along the usual macros when using checkboxes). Then you need to define two variables into the body of your problem `$pointsForCorrectAndMarked` and `pointsForIncorrectAndUnmarked`. Each variable represents the score (between 0 and 1) that students get when a checkbox with a correct answer is selected by the student and when an incorrect answer is not selected respectively. Let us look at the following example:

Let $a, b \in \mathbb{R}$. Select all expressions below which are equivalent to $a^2 + b^2$.

☐ **A.** Incorrect 2 $(a - b)^2$
☐ **B.** Incorrect 5 $a + b$
☐ **C.** Incorrect 1 $(a + b)^2$
☐ **D.** Correct 3 $(a - b)^2 + 2ab$
☐ **E.** Correct 2 $(a + b)^2 - 2ab$
☐ **F.** Incorrect 4 $(a - b)^2 - 2ab$
☐ **G.** Correct 1 $b^2 + a^2$
☐ **H.** Incorrect 3 $(a + b)^2 + 2ab$

Partial credit is given, except when no boxes or all of them are marked.

Solution:

Figure 10: Checkboxes with partial answers

The file is found at `Tutorial/Other/checkbox-partial.pg`.

Key Notes:

- There are three correct answers and 5 incorrect answers.

- We have set `$pointsForCorrectAndMarked = 0.3` and `pointsForIncorrectAndUnmarked = 0.02` as $3(0.3)+ 5(0.02) = 1$.

- If students select all boxes or none, they won't get any credit.

- The macro was created by Nathan Wallach, and shared in the WeBWork forum community.

7.6   Custom Answer Checker and Multiple Correct Answers

Let us consider the following question:

Find a solution to the equation $2\sin(x) = 1$ *in the interval* $[\pi, 3\pi]$.

The answers are $\frac{13}{6}\pi$ and $\frac{17}{6}\pi$. We can use a custom answer checker to consider correct any of the two answers.

First, let us set up a variable with any correct answer: `$ans = 13/6*pi`. And we are going to parse a custom checker by defining the variable `$cmp = $ans->cmp( checker => $checker )`. We are using the subroutine `$checker` (not defined yet) inside the MathObjects evaluator `cmp`.

The key is to define the subroutine `$checker`. The code for our example is the following:

```
1 $checker = sub {
2 my ($correct, $student, $self) = @_;
3 my $stAns = $student->value;
4 return 0 if ( ($stAns < pi) || ($stAns > 3*pi) );
5 return (sin($correct) == sin($stAns));
6 };
```

The source code is `Tutorial/answers-checkers/one-of-multiple-correct.pg`.

Key Notes:

- The checker subroutine has three arguments: The correct answer, the student input and a self reference variable.

- The subroutine must return either 1 (for a correct answer) or 0 (for an incorrect answer

The body to code the problem above is given as follows:

```
1 $ans = (2+1/6)*pi;
2 $cmp = $ans->cmp( checker => $checker );
3
4 BEGIN_PGML
5
6 Find a solution to the equation [`2\sin(x) = 1 `] in the interval [`[\pi, 3\pi]`].
7
8 Answer = [_]{$cmp}
9
10 END_PGML
```

7.7 Feedback and Display Errors

When a student previews or submits their answers, a custom message can be displayed to control their input or provide feedback. This can also be accomplished using a custom checker, that can also help to control student's input.

Continuing with the example from the previous section, what if you want to warn students that their input is not inside the desired interval; or want to display an error message when they submit an incorrect answer to provide immediate feedback. This can be accomplished by adding the following two subroutines two the body of the problem.

```
1 # Report student's input. This is used in answer checkers for messages that should
2 # be shown when previewing, checking, or submitting the answer.
3 sub InputError($) {
4 my ($msg) = @_;
5 Value->Error($msg);
6 }
7
8 # Provide feedback about an incorrect answer. This is used in answer checkers for messages
9 # that should be shown when checking or submitting the answer, but not when previewing it.
10 sub Feedback ($) {
11 my ($msg) = @_;
12 unless (defined $inputs_ref->{previewAnswers}) {
13 Value->Error($msg);
14 }}
```

And modify the custom checker:

```
1  $checker = sub {
2  my ($correct, $student, $self) = @_;
3  my $stAns = $student->value;
4  my $error_msg = "Your answer does not satisfy the equation";
5  my $display_msg = "The number entered is not in the interval!";
6  InputError($display_msg) if (($stAns < pi) || ($stAns > 3*pi));
7  Feedback($error_msg) if (sin($correct) != sin($stAns));
8  return (sin($correct) == sin($stAns));
9  };
```

Key Notes:

- `InputError` displays a message when an answer is previewed or submitted (graded as incorrect) under a certain condition.

- `Feedback` displays a message when an incorrect answer is submitted and also satisfies a given condition.

8   Other Problem Techniques

8.1   Adding Images to Problems

You can add images to your WeBWork problem either as a visual aid or illustration for your problem, or as list of answers in a multiple choice situation. To add an image to the body of the problem, first place the image file in the same folder as your problem. Let us assume that the image is called `image.png`. Next in the problem file create a Perl variable `$image = ("image.png", width=>200,height=>200,tex_size=>400)`. Modify the width and height of the image accordingly to your needs. The `tex_size` attribute is used to set up the image size when creating a PDF copy of the problem. Finally, in the `PGML` block use the syntax `[@ $image @]*`.

You can use several Perl variables to store images and use them inside multiple choice questions (radio, checkboxes). See the companion file `/images/venn.pg` for the source code of the following example.

8.2   Tables

When a problem requires student to fill out values in a table, WeBwork provides a macro that allows to display and record answers in this format. First, it needs to be included the macro `niceTables.pl` into your problem. This technique is useful in Logic when Truth Tables wants to be asked. For example

Figure 11: Tables in WeBWork

The body for this problem is the following:

```
1  @ans=qw(T
2  F
3  F
4  F
5  );
6
7  BEGIN_PGML
8
9  Complete the following truth table by filling in the blanks with T or F as appropriate.
10
11 [@ DataTable([
12 ['\( P \)', '\(Q \)', '\( P \land Q \)'],
13 ['T', 'T', PGML('[_]{str_cmp($ans[0])}')],
14 ['T', 'F', PGML('[_]{str_cmp($ans[1])}')],
15 ['F', 'T', PGML('[_]{str_cmp($ans[2])}')],
16 ['F', 'T', PGML('[_]{str_cmp($ans[3])}')],
17 ], midrules=>1,
18 align =>'|c|c|c|');
19 @]*
20
21 END_PGML
```

Key Notes:

- The syntax for tables is `[@ DataTable( Row1, Row2,---  ), options @]*`.
- Each row is an array enclosed with right brackets. Any text must be added with single quotes `'text'` as an element of this array.
- Any TeX syntax must be added as a text (enclosed with single quotes) and with the operators `\( \)` .
- Student input is recorded as an element of the table with the syntax `PGML('[_]{$answer}')`.
- The options to style the table are `midrules => X`, where horizontal lines to divide cells are displayed if `X=1`, if `X =0` no lines are displayed.
- The `align => ' '` option control the display of the columns and the vertical lines. It follows the TeX syntax.
- The answer array uses the format `@ans = qw ( ans1 ans 2  )` separated by spaces or new lines.
- It can be used any grader or evaluator in the student answer.

In case that there are a lot of answers to check, it is possible to use a `for` control structure to evaluate the answers. See the modified version in Appendix.

8.3   Finite Sets

Sets are useful when creating problems to illustrate Union, Intersection, Subsets of a finite set. Let us look at the following example.

Let $A = \{1, 2, 3, 4, 6, 7\}$ and $B = \{2, 3, 4, 5, 6\}$

Determine the following sets (enter your answers using set notation).

$A \cup B =$ ☐

$A \cap B =$ ☐

$A \setminus B =$ ☐

Figure 12: Basic Set Theory

Students need to enter their answer as a comma separated list, enclosed by curly braces {}. This is intended to be used only when the elements are numbers. There is a work around to include strings, but in that case is easier to use lists. This file is located in `Tutorial/Other/sets.pg`.

The advantage of using this technique, is that we can use WeBWork to compute basic set operations (Union, intersection, difference) as we need to load the context `Interval`. The code for the above problem can be found in Appendix.

Key Notes:

- Load the context by adding the line `Context("Interval");` to your problem.

- Define a set $A = \{1, 2, 3\}$ using the syntax `$A = Compute("{1,2,3}")`.

- The union of two sets $A \cup B$ is achieved by `$A + $B`.

- The difference of two sets $A \setminus B$ is achieved by `$A-$B`.

- The intersection of two sets $A \cap B$ is achieved by `$A->intersect($B);`

- The order in which the elements are listed when entering a set does not matter, and there is no need to enter spaces, but they cause no harm.

- There must be exactly one comma between entries, but no commas before the first entry or after the last entry.

- To enter the empty set as an object, use `Compute("{}");`, and students must type a set with no elements:

- Do not repeat elements in a set, and do not use arithmetic operations or functions in your sets.

- The file for the example problem contains a PERL subroutine to transform a numeric PERL array into a finite set.

8.4 Power Sets

WeBWork does not natively support "Sets of Sets", that can be useful if you want to create a question about that asks students to enter the power set of a finite set. We provide the companion macro `macros/ContextSetsOfSets.pl` and the example file `Tutorial/Other/power-sets.pg.` on how the macro is used.

Key Notes:

- Load the macro `"contextSetOfSets.pl"` and add the line `Context("SetOfSets");` to your problem to initialize this feature.

- Create Sets of Sets using a similar syntax as in the regular finite sets. For example, `Compute('{{},{1},{2},{1,2}}');`.

- This context is limited, it does not support sets operations or variable randomization.

8.5 Matrices with Partial Marks

The context "Matrix" does not support partial marks when comparing student's input with the correct answer as a matrix. Recall that in this case, we use the syntax `[_]*{$A}` if you want to use partial marks, weighted marks or labelled answer rules, an example is provided in the file `Tutorial/Other/matrices-single-entry.pg'`.

8.6 Adding Numeric Functions to WeBWork

WeBWork admits several functions that can be used in computations when both creating problems and students entering answers. As we discussed in ???, we can control what functions are allowed in students answers.

It is also possible to create and include new functions to our roster. In this section we discuss the macro that help us to add those new functions. For example, the `floor` and `ceiling` functions, or the combinatorics functions for permutations of k out of n elements $P(n, k)$ and choosing k out n elements $C(n, k)$. This technique is based in the built-in `"contextIntegerFunctions.pl"` macro, and we use a modified version of this macro in the companion file `"myFunctions.pl"`, that you can modify to add your own custom functions and it already contains the functions mentioned above. Notice that you need a bit of knowledge on coding PERL subroutines (functions in PERL) to create these new ones. For now, we describe its use in the following steps

- Copy the file `myFunctions.pl` into the Course `macros` folder.

- Load the macro `"myFunctions.pl"` and the context `Context ("myFunctions")` in the body of the problem.

- The grader `num_cmp` is not compatible with these functions, you need to use the MathObjects comparison `cmp`.

- As a consequence of the above restriction, when evaluating a numeric answer `X`, you need to make it a MathObject by using `Real(X)`.

- For example, `$answer = Real(P($n,$k))` must be evaluated in PGML as `[_]{$answer}`. You can also use the syntax `$answer = Compute(P($n,$k))` when evaluating these functions.

9 Interaction with other Software

9.1 GeoGebra

"GeoGebra is an interactive geometry, algebra, statistics and calculus application, intended for learning and teaching mathematics and science from primary school to university level." [1]

WeBWork has a script that integrates GeoGebra into WeBWork problems. A basic knowledge on using GeoGebra is assumed, and we will discuss three main ways of integrate a GeoGebra activity (or Worksheet) into Web Work It is important to remark that this script requires JavaScript to work (enabled by default in the traditional web browsers).

Ungraded Activity

This is used when only you want to illustrate or help students to figure out their answers, but their input is done in the traditional web work ways. The steps to embed the GeoGebra activity are as follows (after you have finished your GeoGebra activity):

- Export your activity to a `.png` file, as the PDF copy of the WeBWork can not render the GeoGebra script. Save it in the same folder as the WeBWork `.pg` file. For example `picture.png`.

- Create a PERL variable to store the image. Use the syntax:

  `$im = image( "filename.png", width=>400, height=>400, tex_size=>500 );`

- Load the GeoGebra script into the WeBWork problem with the following code

---

[1]Wikipedia/GeoGebra

```
1  HEADER_TEXT('<script type="text/javascript" src="https://cdn.geogebra.org/apps/deployggb.js
       "></script>
2  <script type="text/javascript" src="https://cdn.jsdelivr.net/npm/ww_ggb_applet/lib/
       ww_ggb_applet.js"></script>');
3
4  TEXT( MODES(TeX=>'', HTML=><<END_SCRIPT ) );
5
6  <div id="applet" class="ww-ggb"></div>
7
8  <script>
9
10 encodedApp = YOUR ENCODED APP HERE
11
12 new WwGgbApplet('applet', {width: 500, height: 400, ggbBase64: encodedApp});
13 </script>
14
15 END_SCRIPT
16
17 </script>
```

The label `applet` can be modified by any text in both the `div` environment and the `newnew WwGgbApplet` function. Modify the width and height (in pixels) to set up the size of the applet in the webwork problem.

- This script can be placed before or after a PGML block, depending on the position that you want to have for the GeoGebra applet in the problem.

- Press `ctrl + shift + B` in GeoGebra to copy the Base64 encoding of your activity, this is a very long alphanumeric string that encondes the html geogebra activity. This string must be enclosed in double quotes `""` when defining the variable `encodedApp` in the above script.

- Inside the `PGML` block , type

```
1  >>
2  [@ MODES(HTML=>'', TeX=>$im) @]*
3  <<
```

to ensure that a static view of the applet is printed in the PDF copy of the question.

GeoGebra Evaluator and Unique Answer

GeoGebra is a Computer Algebra System, so it can be used to check internally answers and bring those answers to WeBWork. This method is used when you want to evaluate a unique answer; for example, that the graph made by the students is correct (no partial grading). The steps for using this method is esentially the same as before, but there are important additions to parse the evaluator to WeBWork. These aditional steps are as follows:

- We make that GeoGebra returns `1` if the condition is correct. So we use the custom checker:
```
1    Context("ArbitraryString");
2    $geogebraChecker = sub {
3    my ($correct,$student,$ansHash) = @_;
4    my @values = split(';', $student->value);
5    return $values[-1] == '1' ? 1 : 0;
6    };
7    $ans = String('1')->cmp(checker => $geogebraChecker);
8
```

- In the `PGML` block we create an answer box `[_]{$ans}`. This answer box is not directly used by students (so it must be left empty when submitting an answer) but it is nessary to parse the answer to WeBWork.

- The GeoGebra script also includes two functions:
```
1        var onUpdate = function(obj) {
2    }
3
4    var onLoad = function(applet) {
5    }
6
```

Inside these functions, we use GeoGebra syntax plus script bridges to transfer information between GeoGebra and WeBwork. The `onUpdate` function is used to record student answer when he/she submits it for grading. The `onLoad` function is used to call GeoGebra input directly from the WeBWork problem.

- Refer to the source code `Tutorial/geogebra/geo2.pg` for the following problem as its construction exemplifies the usage of the new functions.

  Move the sliders to adjust the slope and intercept to plot the tangent line of the function $f(x)$ at $x = 0$.
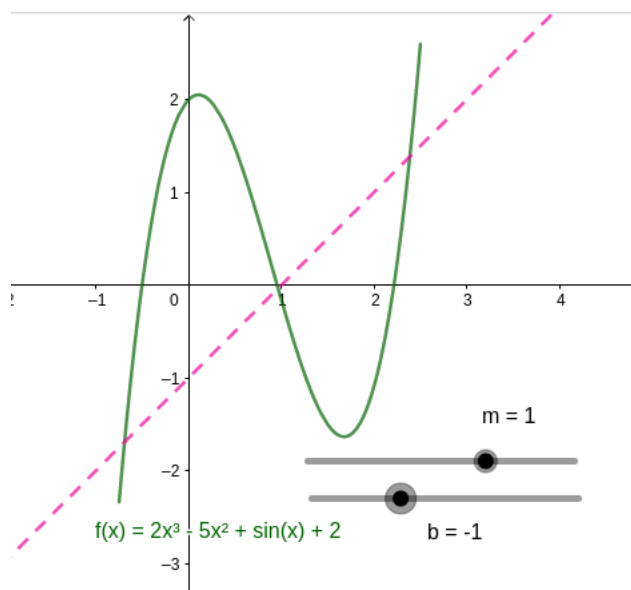

Figure 13: GeoGebra in WeBWork - Graph Evaluation

- In GeoGebra we create the function, line $y = mx + b$ and sliders for m and b.

- In the function `onLoad` we use `this.setCoordinates('AnSwEr0001', { m:'0',b :'0' });` to associate to our WeBWork answer box (name by default `AnSwEr001`) and we initialize the variables m and b that are being evaluated. It is important that you use the same notation as used in GeoGebra. You can also add new GeoGebra objects in this argument, or define your variables directly from this line. However, it is hard to control the visuals directly from the line.

  We use the syntax `applet.evalCommand("")` to create GeoGebra objects in our applet directly from WeBWork. We create the student Line `L = m*x + b` and the correct solution `solution = $m*x + $b` using the correct answers parsed as PERL variables `$m` and `$b`.

  We also use the line `applet.setVisible("L",false);` to hide student's new Line (we are displaying the one created directly from GeoGebra). Another option is that you don't define the line in GeoGebra, and displays this line L instead. However, you need more lines to customize color for example.

  Finally, we compare student's line with the correct line by using the GeoGebra evaluator `AnsCorrect = (L == Solution)` together with the `evalCommand` function.

- In the function `onUpdate` we used the line `this.setCoordinateAnswer('AnSwEr0001', ['m', 'b'], 'AnsCorrect')`. The first argument is the WeBWork default label for the first student input box. The second argument is an array of GeoGebra objects that are being evaluated (m and b), and the last entry is the checker that records from GeoGebra (student answer == correct answer) as defined earlier.

- The line `new WwGgbApplet` in the scripts needs to be modified to include the functions `onLoad` and `onUpdate`.

Evaluate individually GeoGebra objects

This technique is useful when you have several GeoGebra objects that are going to be evaluated, and you can allow students to have partial marks instead of evaluating the single GeoGebra input as a whole. This technique is built

on top of the previous cases, and we exemplify it by using a modified version of 14 where we evaluate separately the input for the slope and the intercept. We don't use the custom checker, but we are using the `NAMED_ANS_RULE` to label the WeBWork answer boxes. It is also nice that when students move the slider, the answer box is automatically filled with the corresponding value (this works only if no input help has been activated in the course configuration).
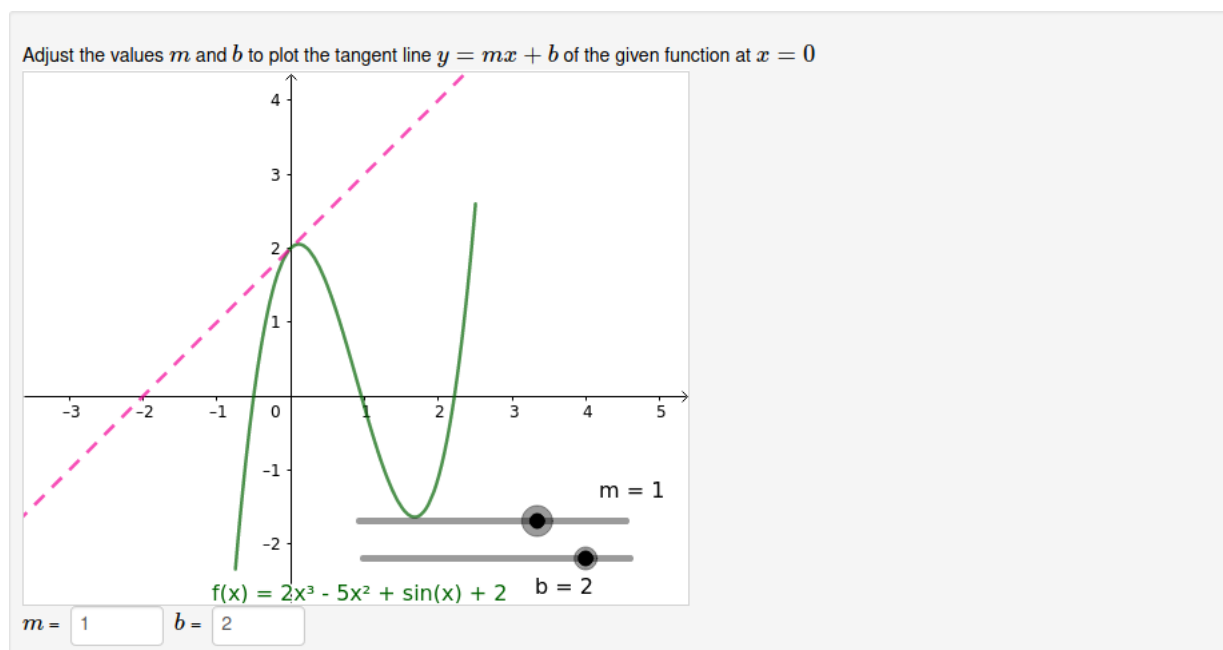

Figure 14: GeoGebra in WeBWork - Graph Evaluation

The source code for this problem is `Tutorial/geogebra/geo1.pg`. The main changes with the previous technique are as follows:

- We are not using the function `onLoad` as all our variables are defined from GeoGebra.

- In the function `onUpdate` we retrieve the student answer by getting the value of m and b when students submit their answers. They are stored in the variables `sm` and `sb` respectively. We use the line `var sm =this.applet.getValue('m`

- We associate these variables with WeBWork answer boxes using the line `this.setAnswer('slope', sm);`. So we need to label our answer box `slope` in WeBWork.

- The answer box is labelled with the line `[`m`]   =   [@ NAMED_ANS_RULE("slope", 5)@]*` inside the PGML block.

- Once associated the GeoGebra value with the answer box, we parse the grader `NAMED_ANS( 'slope' => num_cmp($m) );` where the variable `$m` was defined in the body of the problem and it contains the right answer.