# Examples!

**May 3, 2024**

# Contents

**I Problem Environment** **3**

Problem Environment . . . 3

  Potential Pitfalls and Problems . . . 6

  Lack of Conditionals . . . 6

Alternate 'problem' Environments . . . 7

Alternative Problem Environments . . . 7

**II Answer Types** **9**

answerBox . . . 9

multipleChoice . . . 12

selectAll . . . 14

wordChoice . . . 16

**III Supplemental Features** **18**

Feedback . . . 18

Hints . . . 20

Hiding Answers . . . 22

Expandable Content . . . 25

Footnotes . . . 27

YouTube Embeddings . . . 28

Desmos Embeddings . . . 29

**IV Advanced Content** **30**

**14 Randomized Content** . . . 30

  14.1 Using Sage . . . 31

What is Sage? . . . 31

Using Sage Locally . . . 31

  Where to get Sage? . . . 31

  How to use Sage . . . 31

  How to Compile LaTeX with Sage . . . 31

Using Sage Non-Locally . . . 31

  Testing Sage Code . . . 31

If you want to publish/edit locally . . . 32

  14.2 Sage and Ximera . . . 33

How to Write Sage in LaTeX . . . 33

  Phase One . . . 33

  Phase Two . . . 33

Example . . . 33
Potential Pitfalls and Problems . . . 34
Composition is a little weird . . . 34
Ximera processes all the Sage code fully before being populated . . . 34
Strings are tricky to get Ximera to accept... . . . 35
14.3 Randomizing with Sage . . . 37
Details when using Randomized Content with Sage . . . 37
How the 'Try Another' button works . . . 37
Useful Sage Reference Links . . . 37
Common Functions to Use . . . 37
RandInt . . . 38
NonZeroInt . . . 38
Example . . . 38
14.4 Randomizing Problem Order . . . 40
The problem... . . . 40
Example . . . 40
The Code . . . 40
How the code works . . . 40
The Code Itself . . . 41
14.5 Randomized Graphing . . . 43
**15 Complex Validation** . . . 44
Methods of Validation . . . 44
15.1 Validation Types . . . 45

**V Existing Custom Validators** **46**
15.2 isPositive . . . 46
15.3 sameParity . . . 47
15.4 caseInsensitive . . . 48
15.5 sameDerivative . . . 49
The Problem . . . 49
Potential Pitfalls and Problems . . . 49
Letters like Arbitrary Constants are treated like Variables . . . 49
Example . . . 49
The Code . . . 50
15.6 factorCheck . . . 52
15.7 samePlane . . . 60
15.8 Numeric Targeted Feedback . . . 62

**VI Question Design** **64**
15.9 Interval Answer Formats . . . 64
15.10 Interval Answer Formats . . . 65

15.11 Randomized Multiple Choice . . . . . . . . 67
15.12 Equalities with Radicals . . . . . . . . 69
The Finished Product . . . . . . . . 69
The magic . . . . . . . . 69
15.13 Cubic with Two Factorable Derivatives . . . . . . . . 72
The Problem . . . . . . . . 72
The Result . . . . . . . . 72
The Magic . . . . . . . . 72

# Part I

# Problem Environment

## Problem Environment

*A brief demonstration of the problem environment and nesting interactions for problems with multiple parts.*

## The problem environment

### How to Use the Problem Environment(s)

The most basic part of a Ximera document is the problem environment. Each individual activity will need to be lead by a `\begin{problem}` and followed by a `\end{problem}`. This is what will handle the layout of the page and the numbering of each of the individual problems. It is entirely possible to have text and page content outside of an environment (E.G. this very paragraph), however life will rapidly become strange if there is an answer box outside of this environment. An example of this would look something like

**Problem 1** *An example problem. The answer is, of course, 42.*

$\boxed{42}$

*A bit of text after the answer*

Which is generated by:

```
\begin{problem}
An example problem. The answer is, of course, 42.

$\answer{42}$

A bit of text after the answer
\end{problem}
```

Note that the code that generates the problem doesn't actually give it a number. This is simply the first `\begin{problem}` and so it is called Problem 1. Each problem environment will cause the page to assign it a number. This behavior will be continued for the rest of the tile.

### Nesting

It is a relatively common thing to want to have problems that are related somehow. In such a circumstance you might want to give some thought to how you present that information. Perhaps you want to write a problem but think that the jump from start to end might need a bit of guidance along the way and so you want to ask a relatively simple first question followed by another intermediary step and so on. Perhaps you want to have several preamble problems before having a second stage that combines those pieces of information together. Perhaps you have a problem that has some starting point and then branches out into several different independent subproblems. The answer to each of these concerns is (*clearly*) nesting. By choosing to start a new problem environment before we end one, we create a problem that is only revealed upon completion of the previous. See examples for how this problem chaining works.

### Optional Arguments

The problem environment does not, at the time of this writing, have any optional arguments.

## Examples

(

Example 1) First we will look at a problem environment where the nesting has been used to break down each step of the activity. Each answer will reveal the subsequent step in completing our plea for aid.

**Problem 2** *In this problem we will spell "help". The first character is* $\boxed{h}$

**Problem 2.1** *The second character is* $\boxed{e}$

**Problem 2.1.1** *The third character is* $\boxed{l}$

**Problem 2.1.1.1** *The final character is* $\boxed{p}$

**Problem 2.1.1.1.1** *And now we can escape from this madness.*

***Multiple Choice:***

(a) *FREEDOM!* ✓

This problem is generated by:

```
\begin{problem}
In this problem we will spell ‘‘help".
The first character is $\answer{h}$
\begin{problem}
The second character is $\answer{e}$
\begin{problem}
The third character is $\answer{l}$
\begin{problem}
The final character is $\answer{p}$
\begin{problem}
And now we can escape from this madness.
\begin{multipleChoice}
\choice[correct]{FREEDOM!}
\end{multipleChoice}
\end{problem}
\end{problem}
\end{problem}
\end{problem}
\end{problem}
```

We have simply started a new "problem" before ending the previous one. This will cause the next problem to be concealed until the correct answer has been put in. Notice how the numbering works for each of the revealed, nested problem environments as we travel through the process. The overall problem is referred to as 2, while each layer appends a .1 to whatever was before it. We can contrast this with something like the following. More information on the "answer" and "multipleChoice" calls can be found later in this document.

**Example 2**

Here we will help Dark Helmet enter the air shield combination so that we can reveal the punchline.

**Problem 3** *The combination is:*

*1* $\boxed{1}$ *2* $\boxed{2}$ *3* $\boxed{3}$ *4* $\boxed{4}$ *5* $\boxed{5}$

**Problem 3.1** *Which, as everyone knows, is the kind of password that an idiot would put on his luggage.*

***Multiple Choice:***

(a) *....consider changing the password for your luggage....* ✓

**Problem 3.2** *Comedy gold.*

***Multiple Choice:***

(a) *The works of Mel Brooks are timeless.* ✓

The code that generates that problem is:

```
        \begin{problem}
        The combination is:

        1 $\answer{1}$
        2 $\answer{2}$
        3 $\answer{3}$
        4 $\answer{4}$
        5 $\answer{5}$
    \begin{problem}
        Which, as everyone knows, is the kind of password that an idiot would put on his luggage.
        \begin{multipleChoice}
            \choice[correct]{....consider changing the password for your luggage....}
        \end{multipleChoice}
    \end{problem}
    \begin{problem}
        Comedy gold.
        \begin{multipleChoice}
            \choice[correct]{The works of Mel Brooks are timeless.}
        \end{multipleChoice}
    \end{problem}
    \end{problem}
```

Note that we don't get the punchline from the fearsome Dark Helmet until after we have entered the correct answer for each of the prompts on that level. Further, the punchline and this author's commentary on that punchline are both different "problems". Each of these subsequent problems could have further nested problems or whole problem sets, but then we might be tempted to return to Problem 2. In particular, the punchline itself is 3.1 and the commentary is 3.2.

This automatic numbering for problems can continue *ad nauseam*. We could go any number of steps deep in this process although it is likely that more than say three or four will cause a revolt among students. You have been warned.

## Potential Pitfalls and Problems

### Lack of Conditionals

Like all content written at the LaTeX level, this cannot currently be randomized. While you probably don't want to have problems randomly appearing or disappearing at the top level, the catch is that there is no way to make it hinge on a conditional. For example, it might be desirable for a nested question to show up after a certain number of incorrect responses on the part of the student, but at this time no such functionality exists.

# Alternate 'problem' Environments

*Alternative problem environments. These work identically in every way, but have different 'titles' and can be used to signal to students different types of problems.*

## Alternative Problem Environments

### How they work

There are a number of alternative variants of the problem environment. The key thing to know is that they are *literally* the same as the problem environment in every meaningful way (they use the same commands and formatting macros under the hood as the problem environment), the only way they differ is the displayed name. Currently there are the following problem-style environments:

- problem
- question
- exercise
- exploration

There are ways to create new forms of these problem-style environments with different names, although it requires some minor alterations of the ximera.4ht file, so if you want to do this you would need to contact the Ximera admin on further steps to take.

### Numbering

Notice the numbering of the following problem-style environments:

**Problem 4** *This is a problem, and the answer is 1!* $\boxed{1}$

**Question 5** *This is a problem, and the answer is 2!* $\boxed{2}$

**Exercise 6** *This is a problem, and the answer is 3!* $\boxed{6}$

**Exploration 7** *This is a problem, and the answer is 4!* $\boxed{24}$

The problem-style environments share a counter by default in order to minimize student confusion when they ask about "problem 7", to avoid the situation where they really meant the seventh problem on the page, and not actually "problem 7", as this case there is only one problem-style environment with that number, making it easy to determine which problem the student is (theoretically at least) referring to.

### So why have different ones?

Typically authors use the different types of problem environments to represent different styles of problems. For example, one might make any problem with sub-questions an "exploration" problem to tip students off that there are subproblems involved even before they see them. Likewise, a "problem" might be a difficult problem, an "exercise" might be a mechanical-only skillset problem, and an "exercise" might be an easier level problem. Again this is up to the content author, the option is there if you are interested, but it need not be used.

## Optional Arguments

The problem environment does not, at the time of this writing, have any optional arguments.

## Examples

TBD

## Best Practices

TBD

## Potential Pitfalls and Problems

TBD

# Part II

# Answer Types

## answerBox

*A description of how the answer box works, and it's optional arguments.*

## The answer box

### How to generate/use the answer box.

The answer box is the result of the `\answer` command, and is the bread and butter of the math aspect of Ximera. It has a number of optional arguments (covered below) and one required arguemnt - the content author's answer. Thus if you want to provide an open-answer input for students and the correct answer for that spot is $x^2 + 3x + 1$ you would type `\answer{x^2 + 3x + 1}`. Note that the answer box must be put in mathmode. The previous problem would then look like:

**Explanation.** *The correct answer is* $x^2 + 3x + 1$*:* $\boxed{x^2 + 3x + 1}$.

### How the answer box validates an answer

It uses more than a dozen algorithms to check if the student's provided answer is mathematically equivalent to the content author's provided answer. Most of these are used as shortcuts to allow faster authentication - for example, if the two answers are exact string matches it will immediately be marked as correct. But if the validation fails (returns false) it proceeds to the next validation method. If every validation method returns false/incorrect, *then* the answer is marked wrong. If *any* of the validators returns "correct", then the answer is marked correct. So the multiple validation techniques are more of a sieve than different validation aspects.

Most noteworthy is that the *final* validator in this sieve is computationally intensive (hence we want to try and do 'quick' checks first) but it is exceptionally powerful. It uses the identity theorem from complex analysis - which essentially says that, if the two functions agree on a subset of $\mathbb{C}$ with an accumulation point, then they are the same function.

In this case, Ximera picks a few hundred points just off the real axis in the complex plane that mimic a converging sequence in some sense (a sequence whose distance between points is strictly decreasing), and tests the student given answer and the author-given answer. If they agree (up to machine precision) on more than 98% of those points, it declares them equal. The nature of machine precision requires the 98%, but essentially, if Ximera marks it correct due to this algorithm, it is *exceptionally* unlikely that the answer isn't a correct answer - especially at the student level we use Ximera (i.e. calc sequence and below).

### Optional Arguments

There are also a number of optional arguments/key-values that can be provided to the `\answer` command, for a variety of effects. Most of these are "key-value" pairs, meaning that they have the form "key=value". I outline the key names below and will give examples of values one might use for the given key.

**tolerance:** The Tolerance key-value sets a $\pm$ value on the author's (numeric) answer that will be accepted from a student. For example, the answer box `\answer[tolerance=0.1]{\pi}` would accept anything in the range of $[\pi - 0.1, \pi + 0.1]$ as correct. This was primarily intended to be used for graphical applications, to allow for "fuzzy answers" on a graph.

**validator:** This key-value allows you to load a validator in-line from a prior javascript environment. It's important to note here that **this specific validator call** actually uses the Ximera validation system, and not the generic javascript/python call. As a result, it is *far* superior to the validator environment, but it also only works from a single answer box, which makes it difficult to use for more complex problems.

In practice, you would use a javascript environment to define a js function that returns some kind of true/false type value first. Once you have that 'validateFunc' function, you then call it in the answer box as `\answer[validator=validateFunc` and the validation of the student answer will be entirely decided by the validateFunc you wrote earlier - which may or may not use the 'answer-here' content-author's answer.

**id:** This is used to set an id for the student's answer, which can be called by a validator or some other code. So something like `\answer[id=x]{}` would take whatever the student input into that answer box and assign it as the value of the python variable "x", which could be called in a validator environment or some other code somewhere. I haven't played around with this much *other* than in custom validator environments.

**format:** The Format key-value can change what format the answer box expects to receive (and thus how to validate). For example, using the command `\answer[format=string]{test}` let's the answer box know that it is evaluating the student answer as a direct string, rather than an equation. As a result, it would only accept the *string* " test ". Without the "format=string" key-value, any permutation of the letters t, e, s, and t would be accepted as they would be interpreted as variables - thus "$et^2s$" would have been accepted.

Currently, the only value I remember for this key is the "string" value, but I know there are others in theory.

**given:** This is largely ignored now, but this dictates whether the answer for the answer box is provided in the corresponding pdf that is generated from the ximera file. This can be used to make a "teachers edition" version of the notes, rather than a "student" version. For example, the command `\answer[given=true]{\pi}` would display $\pi$ in the pdf, whereas `\answer[given=false]{\pi` would just show a question mark in the box. Note that given=false is the default behavior.

## Examples

Here is an example using a number of different optional arguments.

**Problem 8** *Type* 2. $\boxed{2}$, $\underset{\text{given}}{\boxed{2}}$, $\boxed{\frac{1}{2}}$, $\int_a^b f(x)\,dx$,

$$-\frac{1}{\frac{12}{1}}\sum_{n=1}^{\boxed{\infty}} n$$

## Best Practices

TBD

## Potential Pitfalls and Problems

### Ximera Errs on the side of the student

Generally, answer validation sides with the student. In most cases this philosophy isn't relevant, but it is noteworthy here because this means that if the `\answer` command can't understand your provided answer, then it will default to mark *any* answer from the student correct. In practice, if you find that you have an answer box that is marking anything/everything correct, this is probably because there is some kind of typo or character in the answer box that it cannot understand.

### Answer Box can be a little too good...

Although the validation technique is impressively comprehensive in practice, that can be problematic when you want the answer in a specific *form*. For example, if Ximera will take any function that is mathematically equivalent to the form the author provided, it becomes very difficult to validate whether or not the student's answer is, say, factored. You can often work around this problem with clever question design, or by writing a custom validator to capture the specific property or aspect that you are trying to assess (as it is less important to only validate the equality, and more important to validate the form - often in addition to the equality, of the answer).

## Ximera is still running on a computer

Also remember that Ximera is still running on a computer, which means it doesn't *truly* understand real numbers. Indeed, it only really understands computable numbers (which we won't delve into here) but the short version, is that anything that is beyond machine-precision can't really be assessed accurately by a machine. For example, the following answer box is expecting the answer $2x$, but try inputting $2x + 2^{-40}$ and see what happens.

**Explanation.** $\boxed{2x}$.

Ximera accepts it as correct, even though we obviously know that's not true. But that's because $2^{-40}$ is a number that is too small for machine precision - so to the computer *running* Ximera, it is effectively indistinguishable from zero. This is just an inherent restriction based on the machine running Ximera, rather than Ximera itself - but it can be important to be aware of, since it can crop up in other ways. For example, if you have something like $e^{-200|x|^{100}}$, then almost regardless of what number Ximera picks in the complex plane, chances are excellent that the result would be too small for machine precision to detect - meaning that the entire *function* is indistinguishable from the constant zero. This can crop up as a result of integrating or differentiating weird functions that end up having extra terms that it is important for students to realize exist (e.g. you want to verify they used the chain rule and not just the base derivative rule) but the "extra part" doesn't seem to matter to Ximera - this is typically an issue of machine precision.

## Answers are exposed via right-click

For accessibility reasons (like screen readers and the like) the library that powers the answer box has a bunch of features bundled in, one of which is the ability to right-click and get different formatting of its contents. Unfortunately, if you right-click and go to "show math as" and then "tex commands" you can very easily see the contents of the answer command that was used in the answer file. This information isn't necessary for accessibility (after all, the answer box is suppose to contain the actual content supplied by the student, and doesn't have any content the student needs to know about) but it's tied to the behavior of the underlying library that supports *all* the rendered math on the page, so we can't remove it without killing *all* accessibility.

However, it is easily countered in the specific case of the answer box, which we detail in the how to hide answers section of the documentation, for those that are interested.

# multipleChoice

*Multiple Choice Answer style syntax, usage, and known issues*

## Multiple Choice

### How multiple choice works

#### How to generate multiple choice answers

The multiple choice answer set is generated by an environment, cleverly named 'multipleChoice' along with the choice command to enumerate choices. Indeed, the multiple choice environment is secretly a kind of retooled itemize/enumerate environment and the choice command is secretly a retooled item command. So the following:

```
\begin{multipleChoice}
    \choice{First Choice is Wrong.}
    \choice{Second Choice is Wrong.}
    \choice{Third Choice is Wrong.}
    \choice[correct]{Fourth Choice is Right!}
    \choice{Fifth Choice is Wrong.}
\end{multipleChoice}
```

Results in:

**Multiple Choice:**

(a) First Choice is Wrong.

(b) Second Choice is Wrong.

(c) Third Choice is Wrong.

(d) Fourth Choice is Right! ✓

(e) Fifth Choice is Wrong.

Notice that the correct answer has the optional argument [correct], more on that below.

#### How multiple choice validates an answer

Simply. Multiple choice allows the student to click on (only) one answer choice, and then hit the "check work" button to check their answer. If that answer is marked as correct (has the "[correct]" optional argument), then they get credit for the problem, otherwise it is marked wrong, which greys out the answer slightly and asks the student to try again (thus the student can see which options they have already tried).

Note that, this means if you have multiple choices flagged as correct, then if a student selects *any* answer that is marked correct and checks it, they will get the answer marked correct - which also locks the problem, so they won't be able to check other options for correctness.

### Optional Arguments

Multiple Choice itself (currently) has no optional arguments. The choice command (currently) has only one key-value pair, which is the flag for whether the choice is correct or not. Technically the key is 'correct' and the value is 'correct' or 'incorrect', but the key defaults to 'incorrect' and simply using the key flips it to correct. In other words using `\choice[correct]{Correct!}` is the same as using `\choice[correct=true]{Correct!}`, and both of those will mark the given choice as the correct answer. Likewise `\choice{False!}` and `\choice[correct=false]{False!}` are equivalent, and both result in the answer being incorrect.

## Examples

TBD

## Best Practices

TBD

## Potential Pitfalls and Problems

### Only one answer can be selected

Since only one answer can be selected (selecting another one automatically deselects any previous one) then having multiple 'correct' answers in the list means a student can only select one of them. Since Ximera locks the answer once it is correct, the only way to test other answers for correctness is to "erase work" and start over with a different selection. Since student's won't do this in practice, and since they expect that only one answer being selectable means there is only one correct answer - this can lead to the assumption that since they got "the" correct answer, then all other answers **must be** incorrect, which may result in questions as to why the other (actually correct) answers aren't correct - or even worse, the blanket assumption that they must be incorrect without any questions or opportunities to correct that misunderstanding.

### There are a finite number of options

Since Ximera allows unlimited attempts at answering (this is a feature we intend to make controllable in some way in the future, but currently all answers get unlimited attempts), a clever-enough-to-be-dangerous student will realize they can just spam the answer until they find the right one and move on without even reading the options. There isn't much to do about this, but it is important to keep in mind when designing problems and deciding on answer-type variety.

### Randomization is 'kind of' pointless

Although there are ways to randomize the content of the problems and answers (see the randomization section), it is important to note that LaTeX level content *cannot* be randomized currently (this is going to be changed in the future edition) which means, although the content of the choice command might be randomized, *which* multiple choice option is correct *cannot* be randomized - something that the students will pick up on mighty quick. For review this isn't *as bad* since students are at least aware enough that they are trying to understand how the correct answer is the correct answer - so knowing "the right answer is the third choice" is *less* important... but it's still important.

It is worth noting that there *is* a workaround for this that we have developed, which is addressed in the randomization section, so if you want to make a randomized multiple choice problem, you should followup there.

# selectAll

*Select All Answer style syntax, usage, and known issues*

## SelectAll

### How Select All works

Note that, basically everything about the select all environment is identical to the multiple choice environment, except that multiple choices can be selected by the student to account for needing multiple correct answers being selected simultaneously.

### How to generate Select All answers

The select all answer set is generated by an environment, cleverly named 'selectAll' along with the choice command to enumerate choices. Indeed, the select all environment is secretly a kind of retooled itemize/enumerate environment and the choice command is secretly a retooled item command. So the following:

```
\begin{selectAll}
    \choice{First Choice is Wrong.}
    \choice[correct]{Second Choice is Right.}
    \choice{Third Choice is Wrong.}
    \choice[correct]{Fourth Choice is Right!}
    \choice{Fifth Choice is Wrong.}
\end{selectAll}
```

Results in:

**Select All Correct Answers:**

(a) First Choice is Wrong.

(b) Second Choice is Right. ✓

(c) Third Choice is Wrong.

(d) Fourth Choice is Right! ✓

(e) Fifth Choice is Wrong.

Notice that the correct answer has the optional argument [correct], more on that below.

### How select all validates an answer

Simply. Select all allows the student to click on as many answer choices as they want, and then hit the "check work" button to check their answer. If they have selected *exactly every* answer choice that is marked as correct (has the "[correct]" optional argument), then they get credit for the problem, otherwise it is marked wrong.

Notice this differs slightly from the multiple choice environment, in that it requires the student to select every "correct" answer and none of the incorrect answers before any credit is given, and doesn't grey out answers (since it is not clear from an unsuccessful attempt which selected answers were right or wrong).

## Optional Arguments

Select All itself (currently) has no optional arguments. The choice command (currently) has only one key-value pair, which is the flag for whether the choice is correct or not. Technically the key is 'correct' and the value is 'true' or 'false', but the key defaults to 'false' and simply using the key flips it to true. In other words using `\choice[correct]{Correct!}` is the same as using `\choice[correct=true]{Correct!}`, and both of those will mark the given choice as the correct answer. Likewise `\choice{False!}` and `\choice[correct=false]{False!}` are equivalent, and both result in the answer being incorrect.

## Examples

TBD

## Best Practices

TBD

## Potential Pitfalls and Problems

### Every correct answer must be selected

Since there is no clear feedback as to which answer is correct or incorrect when students attempt a selection and check it, it can be frustrating for students that think they have "tried every possible combination" and still haven't gotten their answer accepted. That being said, this is an inherent issue with this type of answer in general, and not really a technological barrier/issue, it's just something to keep in mind.

### There are a finite number of options/combinations

Since Ximera allows unlimited attempts at answering (this is a feature we intend to make controllable in some way in the future, but currently all answers get unlimited attempts), a clever-enough-to-be-dangerous student will realize they can just spam the answer possibilities/combinations until they find the right one and move on without even reading the options. With select all, this obviously involves *significantly* more effort than with just a multiple choice, but I have definitely had students *tell me* they have done this on a bunch of these problems - so... chalk another one up to "if you had just spent that effort figuring out the answer instead of 'cheating the system' it would have been less work." There isn't much to do about this, but it is important to keep in mind when designing problems and deciding on answer-type variety.

### Randomization is 'kind of' pointless

Although there are ways to randomize the content of the problems and answers (see the randomization section), it is important to note that the LaTeX level content *cannot* be randomized currently (this is going to be changed in the future edition) which means, although the content of the choice command might be randomized, *which* select all option(s) is/are correct *cannot* be randomized - something that the students will pick up on mighty quick. For review this isn't *as bad* since students are at least aware enough that they are trying to understand how the correct answer is the correct answer - so knowing "the right answer are the first and third choices" is *less* important... but it's still important.

It is worth noting that there *is* a workaround for this that we have developed, which is addressed in the randomization section, so if you want to make a randomized select all problem, you should followup there.

# wordChoice

*In-line multiple Choice dropdown box.*

## wordChoice

### How word choice works

The intended point of word choice is to make an "in-line" version of multiple-choice, although in practice it can be a little more flexible and useful. Note that it generates a dropdown menu instead of a list of choices, to allow for in-line formatting.

### How to generate word choice answers

To generate word choice answers, you use the wordchoice command:

```
\wordChoice{
    \choice{$($}
    \choice[correct]{$[$}
    \choice{$)$}
    \choice{$]$}
    }
```

Which generates: (( / [ ✓/ ) / ])

Notice that this is formatted almost identically to the multipleChoice environment - the main difference being that it is a command rather than an environment to ensure that you don't force a line break (since wordchoice is intended to be an in-line version of the multiple choice answer type). Also note that, as long as you don't use the line break latex command: \\or include a double line break yourself (enter twice) in the code, the wordChoice will command itself won't force a line break, allowing you to chain together a number of these to create a sequence of dropdown boxes that allow you to make more elaborate answers. See the tile on letting students input intervals as answers in this documentation for an example.

### How word choice validates an answer

Simply. Word choice allows the student to click on (only) one answer choice, and if that answer is marked as correct (has the "[correct]" optional argument), then they get credit for the problem, otherwise it is marked wrong.

Note that, this means if you have multiple choices flagged as correct, then if a student selects *any* answer that is marked correct, they will get the answer marked correct - which also locks the problem, so they won't be able to check other options for correctness.

### Optional Arguments

Word Choice itself (currently) has no optional arguments. The choice command (currently) has only one key-value pair, which is the flag for whether the choice is correct or not. Technically the key is 'correct' and the value is 'correct' or 'incorrect', but the key defaults to 'incorrect' and simply using the key flips it to correct. In other words using `\choice[correct]{Correct!}` is the same as using `\choice[correct=true]{Correct!}`, and both of those will mark the given choice as the correct answer. Likewise `\choice{False!}` and `\choice[correct=false]{False!}` are equivalent, and both result in the answer being incorrect.

### Examples

TBD

## Best Practices

TBD

## Potential Pitfalls and Problems

### Only one answer can be selected

Since only one answer can be selected then having multiple 'correct' answers in the list means a student can only select one of them. Since Ximera locks the answer once it is correct, the only way to test other answers for correctness is to "erase work" and start over with a different selection. Since student's won't do this in practice, and since they expect that only one answer being selectable means there is only one correct answer - this can lead to the assumption that since they got "the" correct answer, then all other answers **must be** incorrect, which may result in questions as to why the other (actually correct) answers aren't correct - or even worse, the blanket assumption that they must be incorrect without any questions or opportunities to correct that misunderstanding.

### There are a finite number of options

Since Ximera allows unlimited attempts at answering (this is a feature we intend to make controllable in some way in the future, but currently all answers get unlimited attempts), a clever-enough-to-be-dangerous student will realize they can just spam the answer until they find the right one and move on without even reading the options. There isn't much to do about this, but it is important to keep in mind when designing problems and deciding on answer-type variety.

### Randomization is 'kind of' pointless

Although there are ways to randomize the content of the problems and answers (see the randomization section), it is important to note that LaTeX level content *cannot* be randomized currently (this is going to be changed in the future edition) which means, although the content of the choice command might be randomized, *which* multiple choice option is correct *cannot* be randomized - something that the students will pick up on mighty quick. For review this isn't *as bad* since students are at least aware enough that they are trying to understand how the correct answer is the correct answer - so knowing "the right answer is the third choice" is *less* important... but it's still important.

# Part III
# Supplemental Features
## Feedback

*Demonstration of the 'feedback' environment*

## Feedback

### How to generate/use the feedback environment.

Feedback can be provided to students using the feedback environment within a problem-style environment. The feedback environment takes some optional arguments (see below) to determine when it is displayed, but by default it is displayed after any attempt at the problem (regardless of whether that attempt was correct or incorrect).

For example, consider the following:

**Problem 9** *Guess my number!* $\boxed{\pi}$

***Feedback(attempt):*** $\pi$. *The answer is* $\pi$.

The previous problem is generated by the code:

```
\begin{problem}
    Guess a number! $\answer{\pi}$
    \begin{feedback}
        $\pi$. The answer is $\pi$.
    \end{feedback}
\end{problem}
```

It is also worth noting that you can stack feedback environments in the same question. For example, try the following:

**Problem 10** *Ok, ok, that was unfair. Guess an integer this time.* $\boxed{31415}$

***Feedback(attempt):*** *The answer is 31415. Obviously.*

***Feedback(correct):*** *And this is only displayed because you've entered the correct answer. Good work!*

### Optional Arguments

Currently feedback only has two optional arguments, which serve as triggers for when the feedback is displayed:

**correct** The optional argument "correct" will ensure that the feedback is only displayed once the question has been flagged as "correct" and locked. This can be used for followup explanation of the problem once it has been correctly answered - in case you want to include information that would be too much information for a student still trying to answer it.

**attempt** The optional argument "attempt" will display the feedback after any attempt, regardless of whether that attempt is correct or incorrect. This is the default behavior of feedback if no optional argument is provided, and note that it is the only behavior that gives feedback when a student gives a wrong answer. However, it also doesn't go away once the student gets the correct answer.

## Examples

TBD

## Best Practices

TBD

## Potential Pitfalls and Problems

### Limited triggering

The feedback system only has the two current methods of triggering, either a correct answer, or an attempt. There is a (very) limited custom validator written to give targeted feedback based on specific numeric answers, but it uses javascript validation methods, which are ... not awesome.

### Content is only hidden, but it exists

Technically the content within a feedback environment exists, but is just hidden until triggered. This means, if you want to do something like put another problem within the feedback environment, and you customize the trigger so that a student completes the page without triggering the feedback, then the problem they don't see still counts for "page credit" meaning that the student is now stuck without being able to get full credit.

Under the current system, the only way a problem locks is if they get it right, and correctly answered problems *always* display the feedback under either of the default trigger behaviors - meaning that the above concern isn't possible under the existing system. But if you want to play with the system and do custom work, be aware that this is an easy pitfall to land in by accident.

# Hints

*Demonstration of the 'hints' feature.*

## Hints

### How to generate Hints

Hints can be added via the hint environment. This provides an unfoldable hint that students can click to get a hint for their problem. Consider the following:

**Problem 11** ***Hint:*** *The answer is 42*

*What number am I thinking of?* $\boxed{42}$

Notice that the hint requires a certain amount of time (30 seconds) on the page before a student can open it - to stop students from just spamming the "hint" button to get enough hints to make the problem trivial.

You can also add multiple hints to a problem, consider:

**Problem 12** ***Hint:*** *The answer is prime...*

***Hint:*** *The answer is not even.*

***Hint:*** *The answer is less than 5.*

*What number am I thinking of?* $\boxed{3}$

The above is generated by:

```
\begin{problem}
    \begin{hint}
        The answer is prime...
    \end{hint}
    \begin{hint}
        The answer is not even.
    \end{hint}
    \begin{hint}
        The answer is less than 5.
    \end{hint}
    What number am I thinking of? $\answer{3}$
\end{problem}
```

Notice that, if you want to have multiple hints, it's best to not make them nested, that way the hint counter will correctly know how many hints there are when displaying the button to click.

### Optional Arguments

There are currently no optional arguments for the hint environment.

## Examples

TBD

## Best Practices

TBD

## Potential Pitfalls and Problems

None known, but this hasn't been used much... or at all really.

# Hiding Answers

*How to hide answers from the right-click menu.*

## Hiding Answers from Students

### The Problem, right-clicking

In order to make Ximera accessible (as in DRC accessibility) we use a well supported and widely used library to render the mathematics in a way that is compatible with a whole lot of different accessibility options. One of these features is the automatic generating/providing of the LaTeX code that generates mathematics on the page (interestingly, this is the primary way blind people do mathematics online, they learn and use LaTeX syntax). So, if the original mathemode code is written in LaTeX, then that code is exposed to the student (via a right-click menu in some cases) as-is.

Unfortunately, this applies to *everything* in mathemode, which includes the `\answer` command (which must be in mathemode). Moreover, since the answer command has, as it's argument, the actual *answer* that Ximera is looking for, that means the content of the answer command (a.k.a. the actual answer for the problem) is directly available to students. By way of example:

**Exercise 13** *Right click the following answer box and go to 'show-math-as' and then 'TeX Commands' to figure out the desired answer!* $\boxed{4}$.

Obviously this isn't ideal, as it allows students to just find and copy the answer code into the answer box without doing the problem. There are reasons to actually want this to be the case,[1] but most instructors would prefer to disable this feature.

Unfortunately we can't *literally* disable it as a feature without killing all the accessibility features of the page - which is obviously not something we want to do. Fortunately however, there are ways to disable it in practice.

### First Solution using just LaTeX

The key to understanding how to fix this, is to realize exactly what is being exposed to the student. Specifically, the *content of the answer command* is what gets displayed to the student. As a result, it is as simple as changing what is in the actual answer command. The easiest way to manage this, is to declare a new command that contains the answer and using that instead of the answer itself. For example:

**Problem 14** *Right-click the following answer box to find out the answer!* $\boxed{3!}$

***Feedback(attempt):*** *Ha, got you! The answer is actually 6.*

The code for the above problem is:

```
\begin{problem}
    Right-click the following answer box to find out the answer! $\answer{\ansOne}$
    \begin{feedback}
        Ha, got you! The answer is actually 6.
    \end{feedback}
\end{problem}
```

[1]It gives a known trap to funnel cheaters into and detect them for example

However, the key that makes it work is the hidden previous line, which is `\newcommand{\ansOne}{3!}`. Notice that I could actually leave the contents unsimplified (indeed, it has 3! and Ximera still took 6 as the answer). This can be used to obfuscate even more if you feel so inclined, but there isn't really any reason to do so.

Also note that you can (and should) define the answer command in the *body* of the document (after the begin document command) since the preamble is largely destroyed during the compiling of the assignment.

**Examples**

TBD

**Potential Pitfalls and Problems (With this method)**

It is worth a note that you should have unique answer commands/macros for each answer box to avoid unexpected behavior (technically it *should* expand answers as they occur, but in practice you can run into some interesting expansion scope and timing quirks of Ximera here, so it's best to avoid it).

Also, ideally you want to avoid defining commands that depend on other custom commands. For example you want to avoid something like: `\newcommand{\ansTwo}{3 + \ansOne}`, writing instead `\newcommand{\ansTwo}{3 + 3!}`. Technically one can use custom commands within an answer command, but it almost certainly needs to use an immediate expansion macro definition instead, which would require the use of something like `\edef` or `\let` commands. If you don't recognize the difference between something like `\edef` and `\def` for LaTeX it is *highly* recommended you just avoid using custom commands inside answer commands in general.

## Second Solution using Sage

Another option, which arguably has a number of other benefits, is to use sage to generate the problem and the answer. For example, consider the following problem:

**Problem 15** *Let* $f(x) = ??$. *Then* $f(??) = \boxed{??}$

***Feedback(attempt):*** *Right click the answer box and see what it shows...*

The above code is generated by:

```
\begin{sagesilent}
    p1c1 = 3
    p1f1 = 2*x + 4

    p1ans = p1f1(x=p1c1)
\end{sagesilent}
\begin{problem}
    Let $f(x) = \sage{p1f1}$. Then $f(\sage{p1c1}) = \answer{\sage{p1ans}}$

    \begin{feedback}
        Right click the answer box and see what it shows...
    \end{feedback}
\end{problem}
```

The obvious upside here is that you can then use sage to generate randomized and dynamic content in a nice way. If you are interested, you can check the advanced features section for more information on using sage, and best-practices and associated potential pitfalls and problems with sage.

#### Examples

TBD

#### Potential Pitfalls and Problems (With this method)

TBD

## Best Practices

TBD

# Expandable Content

*Demo of collapsible/expandable content.*

## Expandable and/or Foldable Content

### How to make expandable/foldable content

You can include content that expands out, or collapses back in (i.e. expandable/collapsible boxes, which starts expanded or collapsed). To do this you want to use one of two environments, either the foldable environment or expandable environment.

Thus to have collapsible content that starts compressed, we would type something like:

```
\begin{expandable}
    This text will be hidden inside an expandable bar with an arrow prompt to the right.
\end{expandable}
```

Which gives:

This text will be hidden inside an expandable bar with arrow prompt to the right.

And to have collapsible content that starts expanded, we would type something like this:

```
\begin{foldable}
    This text will be visible, but inside a collapsible box with an arrow prompt to the right.
\end{foldable}
```

Which gives:

This text will be visible, but inside a collapsible box with an arrow prompt to the right.

In theory you could use this to subdivide your content a little more cleanly and provide optional commentary and such that students can collapse/expand as they want.

### Optional Arguments

There are currently no optional arguments for these environments.

### Examples

TBD

### Best Practices

TBD

### Potential Pitfalls and Problems

#### The Collapsible Arrow is Subtle

As you may notice, the arrow that lets you collapse/expand the included content can be pretty small and far to the right, depending on the device and/or window size/resolution the student has, so you may consider prompting them to let them know that some content is expandable so they see the arrow to expand and read info if they are interested. You could also include the environment within other environments to shrink it's size (e.g. inside an explanation environment or something similar) which may help.

# Footnotes

*Show that footnotes work without anything special.*

## How Footnotes Work Online

Footnotes are used frequently in math text, but it isn't clear how they should be dealt with in an online format. Eventually I aim to build out footnotes to work more like the xkcd what-if footnotes. For now though, footnotes *do* work, and they work by expanding in-line in a different color font.

So, you can use something like:

```
We love to have footnotes in math texts%
\footnote{Or, at least, they seem to keep getting used...}
even though they make you go look elsewhere to see what is happening.
```

Becomes:

We love to have footnotes in math texts[2] even though they make you go look elsewhere to see what is happening.

Thus there is nothing special about the syntax or usage to let you use footnotes. It is worth a note here though that you may need to update your ximera.cls, ximera.4ht, and possibly a few other files to make this work - but if you are interested in doing so it's a simple matter of copy/pasting the files, just contact the Ximera dev and they will send you updated files.

## Optional Arguments

There are currently no optional arguments for these environments.

## Examples

TBD

## Best Practices

TBD

## Potential Pitfalls and Problems

[2]Or, at least, they seem to keep getting used...

# YouTube Embeddings

*How to embed YouTube videos, and how they are validated.*

## How Embedding YouTube Works

YouTube content can be embedded (when embedding that video is allowed by the video's owner) easily in Ximera by using the `\youtube` command. The command has most of the embedding and url information already built in, so the only thing you need to include is the part of the url that comes after the "v=" in the url.

For example, in order to embed the video found at the url: `https://www.youtube.com/watch?v=FvgF95i0_lw` you would use the command `\youtube{FvgF95i0_lw}` which would embed the video into the page, like this:

YouTube link: `https://www.youtube.com/watch?v=FvgF95i0_lw`

## How is it validated?

YouTube videos count toward the progress of a tile, but it only checks to see what the saved progress of watching the video is. So if a student watches the video all the way to the end, then they would get full credit for that segment of their grade (see the previous tile on how credit broken up and assigned/earned).

## Optional arguments

There are no optional arguments for the youtube command at this point.

## Potential Pitfalls and Problems

### Progress is tied to time stamp of last watched

It is noteworthy that the credit gained for a video is tied to the percentage last completed of the video. This has two main points to consider.

(a) If the student skips to the end of the video, they can immediately get credit for watching the video, even though they didn't watch it. This information is captured - so we plan on allowing or disallowing this ability in the future, and possibly having a notification for when students skip ahead for credit instead of watching video.

(b) Sometimes YouTube will stop the video a second or two before the actual end of the video. We implemented a "good enough" feature to video watching to account for this - basically if they watch more than 95% or so of the video it counts as full credit, but depending on the length of the video, this can sometimes still have a video registered as not completed by enough to mark the tile as not fully completed. This, in turn, can cause students to lose their minds as they try to assure their instructor that they watched everything fully and did everyone on the page, but can't get full credit.

## Examples

TBD

## Best Practices

TBD

# Desmos Embeddings

*How to embed Desmos Lessons*

## How's it work?

You can embed Desmos lessons into Ximera easily, much like you can with YouTube, using the `\desmos` command. Similarly to the youtube command, Desmos will take in a portion of the url (once the Desmos lesson is marked as embeddable) and configure everything else for you. By way of example, I have a lesson I give student to be able to play around with and understand translations and transformations here:

Thus if you wanted to share a Desmos lesson at https://www.desmos.com/calculator/ryvd3xeltm. To embed this, all we need is the string after the last slash, so we would use the command:
`\desmos{ryvd3xeltm}{}{500}`[3]. Which gives us:

Desmos link: https://www.desmos.com/calculator/ryvd3xeltm

## Validation

Playing around with Desmos lessons has no natural "completion" metric, so it is currently ungraded - meaning that student interaction doesn't count toward progress on the page in any way.

## Optional Arguments

There are currently no optional arguments for Desmos.

## Examples

TBD

## Best Practices

TBD

## Potential Pitfalls and Problems

### No grade potential

Since students can't earn points from it, it can sometimes be difficult to motivate them to interact with a Desmos lesson. One option is to make followup problems that require some kind of manipulation within your Desmos lesson in order to be able to answer them, but finding a motivational method is up to the creativity of the author.

[3]the second argument is the width - which defaults to the width of the window. The third is the height, which usually defaults to something too small, so I typically use 500 or so.

# Part IV

# Advanced Content

# 14 Randomized Content

*High level aspects of adding randomized content to your assignments*

## Methods of Randomization

There are two main routes one could take to implement randomization, LaTeX and Sage.

### LaTeX randomization - works in theory, but not in practice.

LaTeX is a Turing-Complete language, which means it certainly can provide randomization. Indeed, I have actually written some useful randomization packages for LaTeX myself.

Unfortunately, although one could implement this randomization into the LaTeX code for Ximera, the problem comes in how Ximera is currently published[4].

Currently, Ximera is published using the Xake tool, which essentially translates the TeX files into html and javascript (among other things), and then pushes those files to the server. But this means that all the LaTeX content is run and translated into the webpage *when the page is published.* In other words, the only time LaTeX is run (and thus the only time any randomization at the LaTeX level that occurs) happens when an update is published to the web.

So, in order to get any kind of LaTeX-level randomization, you'd need to republish the assignment each time you wanted to randomize it. Moreover, this would only randomize the content served to the server, not to the students - in other words, every student would necessarily get the *same* randomized content, it wouldn't be different for each student in any way.

### Sage randomization

The other option is to use the sagetex package for LaTeX and sage code to implement randomization. This has the benefit of having a much more powerful programming syntax and tools to enable much more complex mathematical manipulation. In order to use sage however, you would likely want to get a local installation of sage and sagetex in order to be able to debug content locally before publishing it to the web. You can read more about getting a local installation of sage and sagetex here - the details are beyond the scope of this document.

In the case of sage, the seed for all randomization is set behind the scenes at the server level. This is important for three reasons.

(a) The randomization needs to be stable so that each time a given student loads a page, they get the same problem - rather than getting new problems every time they see the page (this would be especially confusing when looking back at problems they got "right", since the displayed question and the provided answer wouldn't match up once the displayed problem randomized to something new).

(b) A design decision was made to allow students to get the same problems in the same order for a given randomization for a given semester. This allows them to work together on problems, since they would all get the same variation of a problem together, then they could all hit "try another" and end up with the same problem on the next iteration as well.

(c) Instructors can load a page and see the same problems in the same order, that students got, since they are also loading the same content with the same randomization seed.

[4]As an important note, this will change in the next version of Ximera - which will compile LaTeX directly in the browser, which opens up all kinds of interesting possibilities.

## 14.1 Using Sage

*We discuss aspects of Sage we use, and provide some core sage functions we use.*

### What is Sage?

Sage - officially "sagemath" now - is an offshoot of python - it is basically python for mathematics. This means that, if you are at all familiar with python, writing sage is basically the same - with some additional commands for math specific operations and context.

### Using Sage Locally

#### Where to get Sage?

You can read how to get and install sage here.

#### How to use Sage

In order to use sage locally you would want to download it (see above) and then install the sagetex LaTeX package, which links the two systems together. This is relatively straight forward - the sage installation comes with the relevant LaTeX package automatically, so it's largely a process of copy/pasting the .sty file to the right location and making it available to your LaTeX installation. There are detailed instructions here.

#### How to Compile LaTeX with Sage

Once you have LaTeX and Sage installed locally, you'll want to setup your LaTeX editor to compile using sage as well. If your LaTeX file is named something like "foo.tex" then you'll want to run the following commands to compile:

```
pdflatex foo.tex
sage foo.sagetex.sage
pdflatex foo.tex
```

Depending on your LaTeX editor, there is usually a way to automate this as a command, but that is specific to your editor of choice.

### Using Sage Non-Locally

If you don't want to use sage locally, you can't get sagetex working in any of the typical online editor (for example, overleaf does not support sagetex). Nonetheless you can still write the sage code and the LaTeX code and largely test them independently - since sagetex really does run sage on a generated file that is, more or less, equivalent to the sage content you included in your file.

#### Testing Sage Code

You can test sage code in any public sage compiler - usually referred to as "sage cells". For example, you can use the official sage cell server and copy/paste your code in with a little debug code to see if it compiles and what it yields.

## If you want to publish/edit locally

If you are publishing/editing content locally, you almost certainly want to have sage installed locally so you can properly and fully debug the sage code before trying to publish it. That being said, technically you don't need to be able to compile the sage code locally in order to publish sage code online via Ximera, since the sage is handled in another way - although it might be difficult to get the various sage LaTeX commands to compile during the "bake" process.

## 14.2 Sage and Ximera

*How sage interacts with Ximera.*

## How to Write Sage in LaTeX

There are two phases to writing sage code using sagetex. The first phase is writing the actual code, which generates all the content you want to utilize, and the second phase is calling parts of the code from the first phase to display or use.

### Phase One

The first phase should almost certainly be written in the "sagesilent" block, which is where you write pure sage code that contains everything from generating the function you want to display, to the answer you want the student to enter.

For example, we would write something like the following:

```
\begin{sagesilent}
p1f1 = 3*x+1
p1ans = 10
\end{sagesilent}
```

This generates the sage variable named `p1f1` and `p1ans`. This may seem like an odd answer scheme but there is a good reason for it (see the "potential pitfalls section below". There is also a "sageblock" environment, but this tries to display the running sage code as if it was being done in a sagecell prompt as you go, which is almost always a bad idea.

### Phase Two

The second phase is when you want to display the results of something from the sage code you executed to define all the variables/functions/etc you needed to populate the problem. In the current Ximera system, the only sage command that loads sage content for the student to see, is the `\sage` command. Most importantly, the other typical sage commands, like `\sagestr` and `\sageplot` do not work at all, and should be avoided. So you will want to only use the `\sage` command to call content, even though other commands may work locally - they won't work on a Ximera page[5].

### Example

Consider the following problem:

**Problem 16** *Let* $f(x) = ??$. *Then* $f(4) = \boxed{??}$

The above problem is generated with the following code:

```
\begin{sagesilent}
p2f1 = 3*x+1
p2ans = p2f1(x=4)
\end{sagesilent}
\begin{problem}
Let $f(x) = \sage{p2f1}$. Then $f(4) = \answer{\sage{p2ans}}$
\end{problem}
```

The composition bit may look a little odd, but you can see why this is necessary in the "Pitfalls and Problems" section below.

[5] at least currently, but we are trying to fix that for the next iteration of the platform

## Potential Pitfalls and Problems

### Composition is a little weird

When doing composition in sage, it (now) requires you to use "x = ()" style notation. For example, if you use something like:

```
\begin{sagesilent}
## Function for first problem:
p3c1 = 4
p3c2 = 5
p3f1 = const1*x+const2
# Answer is just f(3)
p3ans = p3f1(3)
\end{sagesilent}
```

Sage will error and you will get a "spinning wheel of death" that kills everything after it on the page. Instead, you need to write:

```
\begin{sagesilent}
## Function for first problem:
p3c1 = 4
p3c2 = 5
p3f1 = const1*x+const2
# Answer is just f(3)
p3ans = p3f1(x=3)# Could also use p3f1(x=(3))
\end{sagesilent}
```

If the substitution is just one variable, you can use "x=var" but if you need something more complex, like substituting in $3x + 1$ you would need to use extra parentheses, like `p3f1( x=(3x+1) )`.

### Ximera processes all the Sage code fully before being populated

Unlike when you compile locally (for most editors) all of the sage code is processed independently before it is populated into the page in Ximera. This means that if you accidentally use the same variable for two different problems, the second definition of that variable will overwrite the value for that variable for *both* problems. In other words, if you had something like:

```
\begin{sagesilent}
## Function for first problem:
const1 = 4
const2 = 5
func1 = const1*x+const2

## Function for second problem:
const1 = pi
func2 = sin(x=(x - const1))

\end{sagesilent}
```

Then the first function will come out as $\pi \cdot x + 5$ not $4x + 5$ because the `const1` sage variable was overwritten in the second problem's code and overwrote it for *every* problem where that variable is used - including the first problem. Annoyingly, you may compile it locally, and depending on your editor, it may have shown the first function as $4x + 5$ leading to more confusion.

## Strings are tricky to get Ximera to accept...

Since sagetex usually uses the `\sagestr` command to generate text, and since the `\sage` command requires mathemode, it makes strings a little... interesting.

In particular, trying to actually display strings, you get some weird latex code. Consider the following code:

```
\begin{sagesilent}
p4str = 'this is a string'
p4ans = 'yes'
\end{sagesilent}
\begin{problem}
Is the following text?
\[
\sage{p4str}
\]
Type ``yes'' or ``no'' (without the quotes!)
$\answer{\sage{p4ans}}$
\end{problem}
```

You get:

**Problem 17** *Is the following text?*

??

*Type "yes" or "no" (without the quotes!)* $\boxed{??}$

The text shows up correctly as a string of text, but it is surrounded by brackets and the LaTeX command. Moreover, if you type in yes (or even 'yes') it won't work. Indeed, if you were to check the code to see what answer it was expecting, you would discover that it is expecting `\textit{yes}` - just like the displayed string.

But, there is a way to get a string to display correctly, both in actual text and in the validator - by using the "LatexExpr(r'string')" command. So if we make some minor modifications to our previous code:

```
\begin{sagesilent}
p5str = LatexExpr(r'\text{this is a string}')
p5ans = LatexExpr(r'yes')
\end{sagesilent}
\begin{problem}
Is the following text?
\[
\sage{p5str}
\]
Type ``yes'' or ``no'' (without the quotes!)
$\answer{\sage{p5ans}}$
\end{problem}
```

Then we get:

**Problem 18** *Is the following text?*

??

*Type "yes" or "no" (without the quotes!)* $\boxed{??}$

Which both displays the problem, and accepts the answer, correctly. Notice that we need to use the `\text` command when we want to display text correctly to the student - otherwise mathmode will display the provided text as if it were in mathmode (i.e. without any spaces, and as if all letters were variables). Also note that we didn't use "format=string" in the answer command - indeed, doing so makes the validation fail because of how LaTeX handles fonts in mathmode (this is deep LaTeX magic, so you probably don't want details - suffice it to say it isn't something that can be easily fixed or worked around with the current system). This means that the validator is really thinking that the 'y', 'e', and 's' are variables, so any permutation of those variables would be accepted. This doesn't typically impact student answers however, but it might be worth remembering just in case.

## 14.3 Randomizing with Sage

*Common Sage Functions and Details*

## Details when using Randomized Content with Sage

Despite having used sage in some of the previous pages, you might notice that none of it was actually randomized. Indeed, if you look at the top right corner of this page, you will see a green “Try Another” button, but in the previous pages, the button doesn’t appear. This is because the “Try Another” button only appears if there is sage content that is being randomized.

### How the ’Try Another’ button works

When you use randomized values as part of the sage code, the green “Try Another” button appears. This button essentially re-rolls all the randomized values and repopulates the *entire* page. There is no way to only randomize part of the page’s randomized code, it always rebuilds the entire page (there are weird pedagogical reasons to do things this way and this just happens to be how the Ximera team decided to go - but it does purposefully work this way).

If you click the “Try Another” button it may (it doesn’t always) pop up a warning saying that all your work will be deleted and progress will be lost. It is noteworthy that, Canvas won’t lower grades because the progress is reset on the Ximera page. However, in order to get full credit, the student needs to have everything completed *at the same time*. It is not sufficient to, say, do problems one through nine, then “Try Another” and then do problem 10 out of 10 to get full credit - they would have to redo all of one through nine as well to get 100% completion.

Also, as a deliberate pedagogical choice, all problems given to students use the same randomized seed - meaning all students get the *same* randomized problems in the same order. This is to allow students to work together on content, as well as for a bunch of technical reasons.

## Useful Sage Reference Links

There are a couple useful links that have a bunch of built-in esoteric mathematical functions/ideas for sage. I figured I’d put them here, and add them as I find them.

- Sage Quickref on Linear Algebra
- Sage Quickref on Number Theory
- A comprehensive introductory course on Sage - course notes.

## Common Functions to Use

Generally we include the following block at the top of the first sagesilent environment of a file that intends to use sage:

```
#####Define default Sage variables.
#Default function variables
var('x,y,z,X,Y,Z,r,R')
#Default function names
var('f,g,h,dx,dy,dz,dh,df')

def RandInt(a,b):
    """ Returns a random integer in [`a`,`b`]. Note that `a` and `b` should be integers themselves to avo
    """
    return QQ(randint(int(a),int(b)))
    # return choice(range(a,b+1))
```

```
def NonZeroInt(b,c, avoid = [0]):
    """ Returns a random integer in [‘b‘,‘c‘] which is not in ‘av‘.
        If ‘av‘ is not specified, defaults to a non-zero integer.
    """
    while True:
        a = RandInt(b,c)
        if a not in avoid:
            return a
```

This defines the default (mathematical) variables for sage (more can be added if needed) and default (mathematical) function names, along with the RandInt and NonZeroInt functions.

## RandInt

This generates a random integer between a and b (inclusive). So `p1c1 = RandInt(-5,5)` creates, and defines, the sage variables “p1c1” as a random integer between -5 and 5. This is as straight forward as it sounds.

## NonZeroInt

NonZeroInt started as a version of RandInt to avoid zero, but even while writing the initial definition it was obvious that zero may not be the only number you may want to avoid - and indeed, you may want to avoid more than just one value. So NonZeroInt(-5,5) gives a non-zero integer between -5 and 5, but you can add a third (optional) argument which is a list of values to avoid. So `p1c2 = NonZeroInt(-5,5,[-p1c1,0,p1c1])` would create the sage variable “p1c2” and assign it a random integer between -5 and 5, but not whatever -p1c1 or p1c1 are, and not 0 as well. Note that `p1c3 = NonZeroInt(-5,5,[p1c1])` would create the sage variable p1c3, and assign it a random number between -5 and 5 - except for p1c1... and importantly, if p1c1 is not zero, then p1c3 *could* be assigned zero unless you include it in the list of things to avoid.

## Example

Consider the following problem:

**Exercise 19** *Is the following polynomial factorable?*

**??**

***Multiple Choice:***

(a) *Yes* ✓

(b) *No*

The above problem is generated by the code:

```
    \begin{sagesilent}
        #####Define default Sage variables.
        #Default function variables
        var(’x,y,z,X,Y,Z,r,R’)
        #Default function names
        var(’f,g,h,dx,dy,dz,dh,df’)

        def RandInt(a,b):
            """ Returns a random integer in [‘a‘,‘b‘]. Note that ‘a‘ and ‘b‘ should be integers themselve
            """
            return QQ(randint(int(a),int(b)))
            # return choice(range(a,b+1))

        def NonZeroInt(b,c, avoid = [0]):
            """ Returns a random integer in [‘b‘,‘c‘] which is not in ‘av‘.
                If ‘av‘ is not specified, defaults to a non-zero integer.
            """
            while True:
                a = RandInt(b,c)
                if a not in avoid:
                    return a

        p1c1 = NonZeroInt(-10,10)
        p1c2 = RandInt(-5,5)
        p1c3 = RandInt(-5,5)
        p1f1 = expand( (p1c1*x-p1c2)*(x-p1c3) )
    \end{sagesilent}
    \begin{exercise}
        Is the following polynomial factorable?
        \[
            \sage{p1f1}
        \]
        \begin{multipleChoice}
            \choice[correct]{Yes}
            \choice{No}
        \end{multipleChoice}
    \end{exercise}
```

You can hit the “Try Another” button above and see how the problem randomizes.

## 14.4 Randomizing Problem Order

*Common Sage Functions and Details*

This isn't what I would necessarily call convenient - it is more of a proof of concept and useful for when it is definitely necessary. Hopefully a more convenient/natural method will be available in future releases.

### The problem...

No matter how well randomized a specific problem is[6], if the student does enough on a page, they already know what mechanics/techniques are involved for each problem - since the order of the problems don't change, just the way it was generated.

Luckily, since we can make strings in Sage, we can technically randomize the prompt along with the answer box. This allows us to randomize question order, along with the questions themselves.

### Example

Consider the problems below - note how the text matches correctly to the actual style of problem. Hit the "Try Another" button and see how they randomize, but move together properly. (Note, only 3 items, so you may need to hit the button a few times to get an actual different order).

**Problem 20** ??

$$?? = \boxed{??}$$

**Problem 21** ??

$$?? = \boxed{??}$$

**Problem 22** ??

$$?? = \boxed{??}$$

### The Code

#### How the code works

It's a bit tricky because we can't call sage code within the latex string formation. Nonetheless we can build a list of prompt/display/answer components, and then randomize the elements of that list. It's important to notice that calling "shuffle" on the list shuffles the list in place (it doesn't work to assign a new variable to be the shuffled result) but it only shuffles it on the outermost layer, so the elements stay in the [prompt,disp,answer] format, which makes calling them easier.

[6]Ok, if it's randomized well *enough* this isn't quite true, but that is usually pretty tough

## The Code Itself

```
\begin{sagesilent}
    #####Define default Sage variables.
    #Default function variables
    var('x,y,z,X,Y,Z,r,R')
    #Default function names
    var('f,g,h,dx,dy,dz,dh,df')

    def RandInt(a,b):
        """ Returns a random integer in [`a`,`b`]. Note that `a` and `b` should be integers themselves to avo
        """
        return QQ(randint(int(a),int(b)))
        # return choice(range(a,b+1))

    def NonZeroInt(b,c, avoid = [0]):
        """ Returns a random integer in [`b`,`c`] which is not in `av`.
            If `av` is not specified, defaults to a non-zero integer.
        """
        while True:
            a = RandInt(b,c)
            if a not in avoid:
                return a

    #### Problem p1
    p1ans = (x-RandInt(-5,5))*(x-RandInt(-5,5))
    p1disp = expand(p1ans)

    #### Problem p2
    p2ans = (NonZeroInt(-5,5)*x-RandInt(-5,5))*(NonZeroInt(-5,5)*x-RandInt(-5,5))
    p2disp = expand(p2ans)

    #### Problem p3
    p3c1 = NonZeroInt(-5,5)
    p3c2 = NonZeroInt(-5,5)
    p3c3 = NonZeroInt(-5,5)
    p3ans = (p3c1*x-p3c2)*(x-p3c3)*(x+p3c3)
    p3disp = expand(p3ans)

    p1prompt = LatexExpr(r'\text{Factor the following polynomial. } \\
    \text{Note that its leading coefficient is one, so you should use the coefficient method. }')

    p2prompt = LatexExpr(r'\text{Factor the following polynomial with the AC-method. }')
    p3prompt = LatexExpr(r'\text{Factor the following polynomial via grouping. }\\
    \text{Remember you must factor it completely. }')

    problemVec = [[p1prompt,p1disp,p1ans],[p2prompt,p2disp,p2ans],[p3prompt,p3disp,p3ans]]
    shuffle(problemVec)

\end{sagesilent}

\begin{problem}
        $\sage{problemVec[0][0]}$ \\
        \[
            \sage{problemVec[0][1]} = \answer{\sage{problemVec[0][2]}}
        \]
\end{problem}
\begin{problem}
        $\sage{problemVec[1][0]}$ \\
        \[
            \sage{problemVec[1][1]} = \answer{\sage{problemVec[1][2]}}
        \]
\end{problem}
\begin{problem}
        $\sage{problemVec[2][0]}$ \\
        \[
            \sage{problemVec[2][1]} = \answer{\sage{problemVec[2][2]}}
        \]
\end{problem}
```

## 14.5 Randomized Graphing

*Attempts at randomizing graphing.*

### Ways to generate randomized graphs (Still being worked on)

#### Using sageOutput environment

SAGE-Output

```
def RandInt(a,b):
    """ Returns a random integer in [‘a‘,‘b‘]. Note that ‘a‘ and ‘b‘ should be integers themselves to avoid u
    """
    return QQ(randint(int(a),int(b)))
    # return choice(range(a,b+1))

def NonZeroInt(b,c, avoid = [0]):
    """ Returns a random integer in [‘b‘,‘c‘] which is not in ‘av‘.
        If ‘av‘ is not specified, defaults to a non-zero integer.
    """
    while True:
        a = RandInt(b,c)
        if a not in avoid:
            return a

p1temp1 = ’temp’

p1temp2 = x^2 + NonZeroInt(-2,2)
plot(p1temp2,(x,-3,3))
```

Note that everything inside a sageOutput environment is essentially locked into it’s own scope - no using variables from in there to populate environments elsewhere on the page (e.g. no using the same random number to define a function in a problem, and display a function on the page - that I’ve found).

Above is (probably) NOT a graph of **??** for example.

$\boxed{\textbf{??}}$

**??**

**??**

#### Using Desmos

# 15 Complex Validation

## Methods of Validation

Stuff about the validator keyword in answer box, versus the validator environment.

## Optional Arguments

There are currently no optional arguments for these environments.

## Examples

TBD

## Best Practices

TBD

## Potential Pitfalls and Problems

## 15.1 Validation Types

There are two main ways to implement validators, the validator optional key-value pair in the `\answer` command, and the validator environment. They have surprisingly different properties and capabilities.[7]

### Validator Environment

#### How to customize the Validator Environment

TBD

#### Optional Arguments

There are currently no optional arguments for these environments.

#### Examples

TBD

#### Best Practices

TBD

#### Potential Pitfalls and Problems

### Validator Key-value in the Answer Command.

#### How to customize the Validator optional argument in the answer box

TBD

#### Optional Arguments

There are currently no optional arguments for these environments.

#### Examples

TBD

#### Best Practices

TBD

#### Potential Pitfalls and Problems

[7] At least in the current iteration - this will hopefully change in the new system.

# Part V

# Existing Custom Validators

## 15.2 isPositive

*This demonstrates a very basic custom validator, which returns correct if the submitted answer is positive.*

JavaScript
```
// NOTE: The below are intended to be used inside an \answer optional argument with the validator key, NOT in

var x
// Check to see if input is positive.
  function isPositive(number) {
    return number > 0;
  };

```

**Problem 23** *This shows a basic "isPositive" validator, that simply checks to see if the input in positive. Notice that the validator completely overrides the expected answer, as the correctness of the answer is determined by the validator returning true/false, and this validator doesn't actually use the provided answer in any way.*

*Enter a number!* $\boxed{22}$

## 15.3 sameParity

*This demonstrates a slightly more complex validator that uses both user input and supplied answer.*

JavaScript
```
// NOTE: The below are intended to be used inside an \answer optional argument with the validator key, NOT in

// Check to see if two inputs are the same parity in terms of even/odd.
  function sameParity(a,b) {
    return (a-b)%2 == 0;
  };



```

**Problem 24** *This validator checks to see if the provided answer has the same even/odd parity as the author-provided answer. This first one expects an even number (content author provided answer is 22).*

*Enter an even number!* $\boxed{22}$

**Problem 25** *Now try an odd number! Supplied answer is 27*

*Enter an even number!* $\boxed{27}$

## 15.4 caseInsensitive

*Demonstrates a way to check student string against provided string, but without case sensitivity*

JavaScript

```javascript
// NOTE: The below are intended to be used inside an \answer optional argument with the validator key, NOT in

// Check to see if two strings match in a case-insensitive way.
  caseInsensitive = function(a,b) {
    return a.toLowerCase() == b.toLowerCase();
  };

```

**Problem 26** *This validator checks the student supplied string against an author provided string, but disables case sensitivity. Try entering in the string:*

*rawrness*

*with whatever capital lettering you want. Author provided answer is RaWrNeSS* $\boxed{RaWrNeSS}$

## 15.5 sameDerivative

*Checks student submission and supplied problem to see if they have the same derivative*

## The Problem

There are lots of reasons we would find it much easier to test derivatives, rather than functions themselves. For example, if you want to see if a student supplied a correct antiderivative can be tricky, since there are infinitely many options for an antiderivative. So this validator checks that the derivatives of the student's answer and the author's answer line up.

## Potential Pitfalls and Problems

### Letters like Arbitrary Constants are treated like Variables

In order to build a validator that is, somehow, indifferent to variable, this validator actually checks all lower case letters as if they are variables to make sure that it matches the provided answer. This can also be nice to make sure that students don't randomly change variables (e.g. default to "x" rather than keep it in terms of the variable given). It also checks the "variable" $C$, to make the classic "$+C$" testable. Note however, that this checks the derivatives as if the $C$ was a variable, so if the student enters something like "$+2C$" and the instructor provided $+C$, then the derivatives won't match ($2 \neq 1$) and thus the answer will be rejected. This may be desirable behaviors in some classes, and not in others.

## Example

JavaScript

```
// NOTE: The below are intended to be used inside an \answer optional argument with the validator key, NOT in

// sameDerivative checks to see if the derivative with respect to x and c are equal.
// Because of how they are loaded, I need to currently manually check each variable letter, so we do a full c
// Currently not checking e, f, g, h, or i as variables because they are special reserve letters in math.
  sameDerivative = function(a,b) {
    return (
    a.derivative('a').equals( b.derivative('a') ) &&
    a.derivative('b').equals( b.derivative('b') ) &&
    a.derivative('c').equals( b.derivative('c') ) &&
    a.derivative('d').equals( b.derivative('d') ) &&
    a.derivative('j').equals( b.derivative('j') ) &&
    a.derivative('k').equals( b.derivative('k') ) &&
    a.derivative('l').equals( b.derivative('l') ) &&
    a.derivative('m').equals( b.derivative('m') ) &&
    a.derivative('n').equals( b.derivative('n') ) &&
    a.derivative('o').equals( b.derivative('o') ) &&
    a.derivative('p').equals( b.derivative('p') ) &&
    a.derivative('q').equals( b.derivative('q') ) &&
    a.derivative('r').equals( b.derivative('r') ) &&
    a.derivative('s').equals( b.derivative('s') ) &&
    a.derivative('t').equals( b.derivative('t') ) &&
    a.derivative('u').equals( b.derivative('u') ) &&
    a.derivative('v').equals( b.derivative('v') ) &&
    a.derivative('w').equals( b.derivative('w') ) &&
    a.derivative('x').equals( b.derivative('x') ) &&
    a.derivative('y').equals( b.derivative('y') ) &&
    a.derivative('z').equals( b.derivative('z') ) &&
    a.derivative(’C’).equals( b.derivative(’C’) )) ;
  };
```

---

**Problem 27** *This problem shows the ’same derivative’ validator. It’s intended to be used to test the result of an indefinite integral, so it requires the “+C” at the end, and (should) mark any answer wrong that doesn’t have it. Note that the $C$ is case-sensitive.*

*Enter in any answer whose derivative is $x^2 + \sin(x) - 3$ (and don’t forget the $+C$, but notice you can also add random constants to it too).*

$$\int x^2 + \sin(x) - 3dx = \boxed{\frac{1}{3}x^3 - \cos(x) - 3x + C}$$

*Sidenote: You can avoid the problem of the $+C$ if you don’t put it in the expected answer box, which will then (correctly) mark an answer wrong that* does *include a $+C$. So in a sense you can force the student to include it, or not, with this validator. Although only checking the derivatives to match has obvious issues for non indefinite integrals.*

## The Code

```
\begin{javascript}
// NOTE: The below are intended to be used inside an \answer optional argument with the validator key, NOT in

// sameDerivative checks to see if the derivative with respect to x and c are equal.
// Because of how they are loaded, I need to currently manually check each variable letter, so we do a full c
// Currently not checking e, f, g, h, or i as variables because they are special reserve letters in math.
  sameDerivative = function(a,b) {
    return (
    a.derivative(’a’).equals( b.derivative(’a’) ) &&
    a.derivative(’b’).equals( b.derivative(’b’) ) &&
    a.derivative(’c’).equals( b.derivative(’c’) ) &&
    a.derivative(’d’).equals( b.derivative(’d’) ) &&
    a.derivative(’j’).equals( b.derivative(’j’) ) &&
    a.derivative(’k’).equals( b.derivative(’k’) ) &&
    a.derivative(’l’).equals( b.derivative(’l’) ) &&
    a.derivative(’m’).equals( b.derivative(’m’) ) &&
    a.derivative(’n’).equals( b.derivative(’n’) ) &&
    a.derivative(’o’).equals( b.derivative(’o’) ) &&
    a.derivative(’p’).equals( b.derivative(’p’) ) &&
    a.derivative(’q’).equals( b.derivative(’q’) ) &&
    a.derivative(’r’).equals( b.derivative(’r’) ) &&
    a.derivative(’s’).equals( b.derivative(’s’) ) &&
    a.derivative(’t’).equals( b.derivative(’t’) ) &&
    a.derivative(’u’).equals( b.derivative(’u’) ) &&
    a.derivative(’v’).equals( b.derivative(’v’) ) &&
    a.derivative(’w’).equals( b.derivative(’w’) ) &&
    a.derivative(’x’).equals( b.derivative(’x’) ) &&
    a.derivative(’y’).equals( b.derivative(’y’) ) &&
    a.derivative(’z’).equals( b.derivative(’z’) ) &&
    a.derivative(’C’).equals( b.derivative(’C’) )) ;
  };
```

```
\end{javascript}

%%%%
\begin{problem}
    This problem shows the 'same derivative' validator. It's intended to be used to test the result of an ind

    Enter in any answer whose derivative is $x^2 + \sin(x) - 3$ (and don't forget the $+C$, but notice you ca
    \[
        \int x^2 + \sin(x) - 3 dx = \answer[validator=sameDerivative]{\frac{1}{3}x^3 - \cos(x) - 3x + C}
    \]

    Sidenote: You can avoid the problem of the $+C$ if you don't put it in the expected answer box, which wil
\end{problem}
```

## 15.6 factorCheck

*Checks provided factoring to see if it matches author's factoring.*

JavaScript

```

var debugOn=true;

/*
HOW THIS SHOULD WORK:
    Initially check to make sure the submitted answer (and proposed answer) are at least in some kind of theo

    Next duplicate the raw trees so we can mess with them without worrying about changing the original.

    Fold up any exponents on both trees so that our root node is of the form: ['*',factor1,factor2,factor3,..
        This includes killing off any leading negative signs (again we'll compare for equality using the orig

    Now call a recursive function to deep-dive into each factor to find what degree that factor actually is.
        Will also identify if the factor isn't even a factor, in which case we will return a negative value t

    Once we have the degree for each factor, now we can compare the instructor degree list and student degree
*/


// Subfunction to identify if something is a number:

function isNum(numb) {
    if ((typeof numb === 'number')||(numb=='e')||(numb=='pi'))
    {return true} else {return false}
}

// Subfunction to identify if something is a non-negative integer:

function isPosInt(numb) {
    if ((isNum(numb))&&((numb>=0)&&(numb%1==0)))
    {return true} else {return false}
}


// This does a recursion through a factor to eventually find it's degree - assuming it's a polynomial.

function degreeHunt(tree,position,curDeg) {
    //(Re)set curDeg just in case:
    var curDeg=0;

    // First, let's figure out what to do about negative signs, since they can be annoying.
    //  I think there's three possibilities, it's a negative array, a negative x, or a negative number.

    if (tree[position][0]=='-') {
        debugText('Processing a minus sign.');
        if (tree[position][1]=='x') {
            // We found a ``-x'' term within our factor, so that's degree 1 I guess!

            debugText('Found a -x term!');
            curDeg = Math.max(curDeg,1);

        } else if (Array.isArray(tree[position][1])) {
            // Else if there's a negative outside of an array, just bypass the negative and keep digging for
```

```

                debugText('Found a negative Array term!');
                let tempVal = degreeHunt(tree[position],1)
                if (tempVal<0) { return (-1)} else {
                    curDeg = Math.max(curDeg,tempVal);
                }


            }// Note that, if it's a negative number, I don't care about it, so no need for an ''else''.
        } else if (tree[position][0]=='apply') {
            debugText('Processing an apply symbol.');
            // any 'apply' is inevitably a function that isn't a polynomial.
            //  Although, only if it actually has an 'x' in there - so I need to fix/update that at some point.

            // To find out if the apply is actually just a number or not, we need to recurs through every sub-nod
            //  If the degree ends up positive, then we have a variable inside the apply function it's not a poly
            //  If the degree ends up zero, then it's ultimately some bizarre number formation and we're fine.

            var tempDeg = 0;

            for (var j = 1; j < tree[position].length; ++j) {
                // Walk the array to find any powers of x.
                if (tree[position][j]=='x') {
                    // If the entry is just x, then we have pos deg and we are done.
                    debugText('Found an x inside an apply function that suggests the factor is not actually a pol
                    return (-1);
                } else if (Array.isArray(tree[position][j])) {
                    // If the term is an array, then we need to recurs to find the degree.
                    let tempVal=degreeHunt(tree[position],j,0);
                    if (tempVal!==0) {
                        debugText('Found an issue inside an apply function that suggests the factor is not actual
                        return (-1)
                    }
                }// Any other option is degree 0, so no need for an else.
            }// End of for loop and end up 'apply' function
        } else if (tree[position][0]=='^') {
            // Now we process the exponent sign case, but be careful cause students do crazy shit,
            //  So it might be a x^N situation, but it might be some other shenanigans.

            debugText('Processing an exponential sign.');

            if ((tree[position][1]=='x')&&(isPosInt(tree[position][2]))) {
                // We have something like x^N

                debugText('We have x^N');

                curDeg = Math.max(curDeg,tree[position][2]);

            } else if (tree[position][1]=='x') {
                // If the base is x but it's not being raised to an integer power, then it's not a proper monomia
                //  Note that we are assuming here that students won't put shit like ''1+1'' as the power, if so,

                debugText('We have x^(g(x)) but g(x) is not a positive integer.');

                return -1;

            } else if ((isNum(tree[position][1]))&&(isNum(tree[position][2]))) {
                // This means we have a^b which is still just a number, so it's fine... but doesn't give a degree

                debugText('We have a^b');

            } else if ((Array.isArray(tree[position][1]))&&(isPosInt(tree[position][2]))) {
                // We have something of the form (f(x))^N which might be a part of a factor.
                //  We recurse on the array, and multiply the result by the N.

                debugText('We have f(x)^N');
                let tempVal = tree[position][2]*degreeHunt(tree[position],1,curDeg);
                if (tempVal<0) { return (-1)} else {
                    curDeg = Math.max(curDeg,tempVal);
                }


            } else {
                // All other situations are inevitably not polynomials.

                return (-1);
            }
        } else if (tree[position][0]=='/') {
            // This is problematic, because the only way we can allow a division is if the denominator is just a
            //  But if the denominator has an 'x' anywhere in it, then we have a problem...

            debugText('Processing a division sign.');

            if (tree[position][2]=='x') {
                // We're dividing by x, which is bad.

                debugText('Dividing by x, naughty naughty!');

                return (-1);
            } else if (Array.isArray(tree[position][2])) {
                //  Let's try doing a recurs, and if we get a result that isn't 0, then that means there's an 'x'
                if (degreeHunt(tree[position],2,0)!==0) {
                    // If it's not 0, then we found an 'x' or something that invalidates the polynomial.
                    return (-1)
                } // otherwise we have some kind of 'dividing by a number' situation, so it's fine, and doesn't i

            }

            // Now that we've dealt with the possibilities that cause a non-polynomial function, we can proceed a

            debugText('The denominator is just a number, so we need to check the numerator.');

            if (tree[position][1]=='x') {
                // If we've made it past the first two hurtles, then the bottom is just a number of some form.
                //  So if the top is 'x', then we have something like ``x/a'' which is still a degree 1 factor.
                var curDeg = Math.max(curDeg,1);
                debugText('We found a fraction with just x in the top.');
            } else if (Array.isArray(tree[position][1])) {
                // If the top of the fraction is an array - but we've already reduced to case where denominator i
                //  So we need to figure out if there is a degree in the top to count.

                debugText('We found a fraction with an array for the numerator.')
                var tempVal = degreeHunt(tree[position],1,0)
                debugText('I think the numerator degree from the array is: '+tempVal);
                if (tempVal<0) { return (-1)} else {
                     var curDeg = Math.max(curDeg,tempVal);
                }

            }// Note that if none of the above, then it's just a/b, which is fine and doesn't impact degree.

        } else if (tree[position][0]=='*') {
            // If we are multiplying, we could be multiplying a bunch of terms - maybe a bunch of x terms.

            debugText('Processing a product sign.');

            var tempDeg = 0;
            for (var j = 0; j < tree[position].length; ++j) {
                // Add the degree of each thing being multiplied, even though most are probably zero.
                if (tree[position][j]=='x') {
                    // If the entry is just x, then we have deg 1.
                    debugText('Found a solo x term inside the product sign, degree is at least 1.')
                    tempDeg = Math.max(tempDeg,1);
                } else if (Array.isArray(tree[position][j])) {
                    // If the term is an array, then we need to recurs to find the degree.
                    // let tempVal=degreeHunt(tree[position],j,0);
                    if (degreeHunt(tree[position],j,0)<0) {
                        return (-1)
                    } else if (degreeHunt(tree[position],j,0)==0){
                        debugText('No degree term found inside product sign, so we stay at degree: '+tempDeg);
                    } else {
                        debugText('found a higher degree term inside the product sign, degree is at least: '+temp
                        tempDeg = Math.max(tempDeg,degreeHunt(tree[position],j,0));
                    }
                }// Any other option is degree 0, so no need for an else.
            }
            // Once the for loop finishes, we convert the temp deg to the current deg.
            var curDeg = Math.max(curDeg,tempDeg);
            debugText('Inside the product sign we found the degree is: '+tempDeg);
            debugText('So we set the curDeg to: '+curDeg);
        } else {
            // If it's none of the above, then we should just recurs on any arrays we find.

            debugText('Processing an unknown sign? Specifically: '+tree[position][0]);

            for (var j = 0; j < tree[position].length; ++j) {
                // Walk the array to find any powers of x.
                if (tree[position][j]=='x') {
                    // If the entry is just x, then we have deg 1.
                    curDeg = Math.max(curDeg,1);
                } else if (Array.isArray(tree[position][j])) {
                    // If the term is an array, then we need to recurs to find the degree.
                    let tempVal=degreeHunt(tree[position],j,0);
                    if (tempVal<0) { return (-1)} else {
                        curDeg = Math.max(curDeg,tempVal);
                    }
                }// Any other option is degree 0, so no need for an else.
            }

        }
        debugText('Made it to the end of the degreeHunt function, which means we need to return a curDeg variable
        return curDeg
    }

    // Subfunction just to make sure that the submitted function is in a legitimately factored form.

    function JNFisFactored(factorTree) {

        // First we check to see if we have a negative factored out, which messes everything up in the tree.
        if ((factorTree[0]=='-')||(factorTree[0]=='*')||((factorTree[0]=='/')&&(isNum(JNFoperation[1])))
        ) {return true} else {return false}
    }

    // Subfunction to make debug easier.

    function debugText(text) {
        if (debugOn) {
        console.log('DEBUG INFO::' + text)
        }
    }



    function factorCheck(f,g) {
        // This validator is designed to check that a student is submitting a factored polynomial. It works by:
        //  Checking that the degree of each factor matches between student submitted and instructor submitted an
        //  Checking that the submitted answer and the expected answer are the same via real Ximera evaluation,
        //  Checking that the outer most (last to be computed when following order of operations) operation is mu
        //  It ignores degree 0 terms for degree check, and now can ignore factored out negative signs.


        console.log(f.tree);
        console.log(g.tree);

        if (JNFisFactored(g.tree)==false) {
            console.log('Answer rejected, instructor answer not in a factored form. Bad instructor, no donut.');
            return false
        }

        // First we check to make sure it is in *a* factored form:
        if (JNFisFactored(f.tree)==true) {
            console.log('The student answer is at least in *a* factored form.');


        } else {
            console.log('Answer rejected, student answer not in a factored form.');
            return false
        }

        // Let's duplicate the trees to manipulate, so we keep the original correctly.
        var studentAns=f.tree
        var instructorAns=g.tree

        // Also, if there is a factored out negative, let's just kill that, since we aren't doing a funtion compa
        while (studentAns[0]=='-'){studentAns = studentAns[1];};
        while (instructorAns[0]=='-'){instructorAns = instructorAns[1];};

        // Now we want to fold up any root-level exponents into duplicate children of the master tree,
        //  This lets us assume the top-level node has 1 child per factor.
        debugText('folding up external exponents of studentTree  so factors do not have exponents')
        for (var i = 0; i < studentAns.length; ++i) {
            if ((studentAns[i][0] == '^')&&(isPosInt(studentAns[i][2]))) {
                studentAns=studentAns.concat(Array(studentAns[i][2]).fill(studentAns[i][1]));
                studentAns.splice(i,1);// This should theoretically remove the original term now that we've dupli
                i=i-1;// since we shortened our array by 1, we should move the iteration value down 1 too.
                debugText('Ok, I folded up a term, so hopefully our student vector still makes sense. It is now:
            } else if (studentAns[i][0] == '^') {
                //if we have a power, but not a positive integer power, then we have a non-polynomial factor, so
                console.log('I think I found a non-polynomial term, specifically some root term has a non natural
                return false;
            }
        }
        debugText('After all preprocessing my studentAns vector is:');
        console.log(studentAns);

        // Now re repeat with instructor tree:
        debugText('folding up external exponents of instructorTree so factors do not have exponents')
        for (var i = 0; i < instructorAns.length; ++i) {
            if ((instructorAns[i][0] == '^')&&(isPosInt(instructorAns[i][2]))) {
                instructorAns=instructorAns.concat(Array(instructorAns[i][2]).fill(instructorAns[i][1]));
                instructorAns.splice(i,1);// This should theoretically remove the original term now that we've du
                i=i-1;// since we shortened our array by 1, we should move the iteration value down 1 too.
                debugText('Ok, I folded up a term, so hopefully our student vector still makes sense. It is now:
            } else if (instructorAns[i][0] == '^') {
                //if we have a power, but not a positive integer power, then we have a non-polynomial factor, so
                return false;
                console.log('Found a non-polynomial term in the instructor answer... huh? Check the code!');
            }
        }
        debugText('After all preprocessing my instructorAns vector is: '+instructorAns);

        /*
            ::NOW LETS PROCESS THE STUDENT ANSWER::
        */

        var studentDegList=[0]
        for (var i = 0; i < studentAns.length; ++i) {
            if (studentAns[i] == 'x') {
                // If the factor is simply 'x', then it's a degree 1 factor... yay?
                studentDegList.push(1);
                debugText('Found another factors degree, so now studentDegList is: '+studentDegList);
            } else if (Array.isArray(studentAns[i])) {
                // Otherwise, if it is an array, we have something to go hunting in.
                studentDegList.push(degreeHunt(studentAns,i,0));
                debugText('Found another factors degree, so now studentDegList is: '+studentDegList);
            }// Note that the only other possibility is it being a number, which we don't care about.
        }
        studentDegList = studentDegList.filter(x => x!==0);// Remove all zeros from the array to avoid stupid pad
        studentDegList.sort();// Sort the result so that we can later compare it to the instructor version.
        debugText('The final List of Factor Degrees given by the student is: ' + studentDegList);

        if (studentDegList.some(elem => elem<0)) {
            console.log('I think one of the student factors is NOT a polynomial. So I am rejecting the answer.');
            return false
            }


        /*
            ::NOW LETS PROCESS THE INSTRUCTOR ANSWER::
        */

        var instructorDegList=[0]
        for (var i = 0; i < instructorAns.length; ++i) {
            if (instructorAns[i] == 'x') {
                // If the factor is simply 'x', then it's a degree 1 factor... yay?
                instructorDegList.push(1);
                debugText('Found another factors degree, so now instructorDegList is: '+instructorDegList);
            } else if (Array.isArray(instructorAns[i])) {
                // Otherwise, if it is an array, we have something to go hunting in.
                instructorDegList.push(degreeHunt(instructorAns,i,0));
                debugText('Found another factors degree, so now instructorDegList is: '+instructorDegList);
            }// Note that the only other possibility is it being a number, which we don't care about.
        }
        instructorDegList = instructorDegList.filter(x => x!==0);// Remove all zeros from the array to avoid stup
        instructorDegList.sort();// Sort the result so that we can later compare it to the instructor version.
        debugText('The final List of Factor Degrees given by the instructor is: ' + instructorDegList);

        if (instructorDegList.some(elem => elem<0)) {
            console.log('I think one of the instructor factors is NOT a polynomial. So I am rejecting the answer.
            return false
            }

       /*
            ::NOW WE COMPARE::
       */

        if (studentDegList.length!=instructorDegList.length){
            console.log('Ans Rejected: Wrong number of factors.');
            return false;
            }

        for (var i = 0; i < studentDegList.length; ++i) {
            if (studentDegList[i] !== instructorDegList[i]) {
            console.log('Ans Rejected: At least one factor is the wrong degree.');
            return false;
            }
        }

        if (f.equals(g)){
            } else {
            console.log('Ans Rejected: Factors do not expand to original Polynomial.');
            }

        return (f.equals(g));
    }



```

**Problem 28** *This validator checks to see if the provided "factored form" from the student is actually factored in a similar way to the author-provided "factored form".*

*For example, if you have the polynomial $x^3 - 4x^2 - 4x + 16$, you might want the student to just do the factor by grouping step and want them to enter in $(x^2 - 4)(x - 4)$. Try trying in the full polynomial versus a fully factored version, versus the desired version:* $\boxed{(x^2-4)(x-4)}$

**Problem 29** *Now, let's say you want them to fully factor, not just factor by grouping. Try the factor by grouping version versus unfactored vs fully factored here:* $\boxed{(x-2)(x+2)(x-4)}$

**Problem 30** *Adding another problem here that deliberately has some repeated factors to test out if they work. The answer should be $(x-1)^3(x+1)(x^2-1)$ (Also testing the 'not fully factored version is the target to accept').*

$\boxed{(x-1)^3(x+1)(x^2-1)}$

**Problem 31** *Finally, made a sage generated version so that we can make sure nothing about sage syntax messes things up.*
*Actual answer should be* **??**. *[Also stress testing to make sure larger number of factors isn't an issue.]*

$\boxed{\textbf{??}}$

## Potential Problems and Pitfalls

The current generation of Ximera *really* isn't designed to have this level of custom validation check - so the actual validator code is remarkably hacky and intensely exploits how the data was saved in the backend of the renderer *at the time I wrote the validator*. The current generation of this validator is fairly robust, but future patches to underlying systems may break it. Currently, as long as numeric exponents are actually simplified (e.g. students write an exponent as 2 instead of 1+1) things seem to work pretty much as expected (including correctly handling negative signs, simplified exponents, and fractions). Some irrational and weird numbers might cause issues, but that is more to do with needing to figure out how numbers in weird formats might be submitted or encoded and I don't have enough data for that yet.

## 15.7 samePlane

*This demonstrates the validator that tests for points to represent the same plane.*

JavaScript

```
// NOTE: The below are intended to be used in a validator environment, rather than the validator argument of

// testPlane function tests list a against list b element-wise to
// make sure that the ratio is the same value for each,
// ie a = k.*b; eg a = [1, 1, 2]
// b could be [3, 3, 6] but not [3, 3, 5].
// This can be tricky if one of the inputs is zero though.
function testPlane(a,b) {
        var ratioVec = [];
        var result = 1;
        for (i = 0; i < a.length; i++) {
                if (b[i] == 0) {
                        if (a[i].evaluate() != 0){
                                result = 0;
                        }
                }
                else {
                        var tempRat = a[i].evaluate()/b[i];
                        ratioVec.push(tempRat);
                        console.log(tempRat);
                }
        }
        ratioVec.sort();
        var endPoint = ratioVec.length-1;
        if (ratioVec[0] == ratioVec[endPoint]) {
                result = result*1;
        }
        else {
                result = result*0;
        }

    // Throw in a default check to make sure student doesn't just use all 0s
    // Going a bit overboard in case we want longer vectors in future.

    var isAllZero = 0;
    for (i = 0; i < a.length; i++) {
      if(a[i].evaluate() !== 0) {
        isAllZero = 1;
        break;
      }
    }

    result = result*isAllZero;
        return result;
};


// End of Validators intended to (only) be used in validator environments.

```

**Problem 32** *This problem shows the 'samePlane' validator. It's intended to detect any equivalent expression that generates the same plane. Note that formatting is tricky for authoring the problem as it requires that you use the validator* *environment to test multiple answer boxes simultaneously, and you must enter the predicted answers and the ids of the supplied answers in as lists using brackets, and in the same order. Thus if your id for the $x$ coefficient has 'id=x', the $y$ coefficient has 'id=y' and the $z$ coefficient has 'id=z' (in their respective* `\answer` *commands optional argument) and the expression you want for the plane is, say, $-3x, 5y + z = 0$, then you would use the following:*

```
Plug in any expression that generates the plane defined by $x + y + 2z = 0$.
\begin{validator}[{testPlane([x,y,z],[1,1,2])}]
    \[
        \answer[id=x]{1}x + \answer[id=y]{1}y + \answer[id=y]{2}z = 0
    \]
\end{validator}
```

*The code is demonstrated below, try various equal and non-equal coefficients and see how it goes. Note that this also works correctly if you feel like making some of the coefficients zero for whatever reason.*

*Plug in any expression that generates the plane defined by $x+y+2z = 0$. testPlane([AnsOne,AnsTwo,AnsThree],[1,1,2])*

$$\boxed{1}x + \boxed{1}y + \boxed{2}z = 0$$

*As a footnote, due to peculiar coding, the value included the the answer command's mandatory argument is never used. So you could have used* `\answer[id=x]{673}` *instead of* `\answer[id=x]{1}` *and as long as you entered the correct values into the validator environment, it would still only take correct answers for the $x$ coefficient.*

## 15.8 Numeric Targeted Feedback

*Generic testing link.*

JavaScript

```
//  A validator that also allows for targeted feedback.
//  Currently only works with numeric answers with some limitations (read comments in the validator).
//
//  Moreover, you need to use the corresponding ‘‘\feedbackOutput" command which builds the html for the outp
//  You need the argument of the \feedbackOutput command to match the ’outputName’ that you enter in the
//      validator function (see the example; using the output name ’outputOne’.
//      Note that the apostrophes are necessary as JS needs it to be a string.
//
//  Finally, the input for the validator needs to be written in javascript code, not LaTeX which is a pain.
//      See the note in the validator; in short you need to use Math.e or Math.pi for the math constants.
//      For a list of javascript usable functions go to:
//          https://www.w3schools.com/js/js_math.asp
//      Keep in mind that the math module (all lowercase, as oppose to Math) is not loaded.
//          Among other things, this means there are no complex numbers or complex valued functions.


function feedbackFunc(studentInput,outputName,ansCheck,ansFeedback,defaultfeedback=’’) {
    var studentAns = studentInput.evaluate();

    // Note that you can use e and pi by using Math.E and Math.PI;
    //      but these are using javascript values, not Ximera equalities.
    // This means testing equality is a lot more fragile;
    //      in particular function equality will be nearly pointless to test.

    // Set the default feedback text, for a student answer that doesn’t match any predicted answer.
    var feedbackText = defaultfeedback;

    // Now loop through the expected list and see if the student entered an expected answer.
    for (var i = 0; i < ansCheck.length; i++) {
        // If the students answer matches the current expected answer, print the corresponding feedback.
        if (studentAns==ansCheck[i]) {
            feedbackText = ’Feedback: ’ + ansFeedback[i];
        }
        // If you have found the correct answer, return correct and we’re done.
        if (studentAns==ansCheck[0]) {
            return 1;
            }
    }

    // Print the feedback to the dummy element we built with \feedbackOutput.
    document.getElementById(outputName).innerHTML = feedbackText;

    // If we have gotten this far, we haven’t found the right answer, so return false.
    return 0;
}
```

**Problem 33** *Testing new feedback mechanism. The correct answer is $\frac{20}{3}$. But try typing in 4 or $5-e$ first for targeted feedback example. This is to demonstrate that you can use some irrational constants, but keep in mind that they are javascript coded, not latex coded. See tex file for more detail. This validator currently only works with numeric values, and even then only kinda.*

*feedbackFunc(inputOne,’outputOne’,[6+2/3,5-Math.E,Math.PI/6,4],[’Correct!’,’Try going up 1+e+2/3’,’You entered the arcsin*
$\boxed{x}$. *¡p id=”outputOne”¿ ¡/p¿*

# Part VI

# Question Design

## 15.9 Interval Answer Formats

*Some Tips on Best Practices for Writing Problems.*

## 15.10 Interval Answer Formats

*How to get students to input an interval as an answer.*

This is an example page to show how to take in intervals as answers to a problem. There are a number of options, each one demonstrated below with the corresponding code printed below the answer box/section that you can copy/paste into your own relevant tex file in order to modify and use.

**Problem 34** *This first is the easiest but least difficult for a student to answer; which is just straight up multiple choice.*

*We want the interval:* $[a, b)$.

***Multiple Choice:***

(a) $(a, b)$

(b) $(a, b]$

(c) $[a, b)$ ✓

(d) $[a, b]$

*The code for the multiple choice answer block is the following:*

```
\begin{multipleChoice}
    \choice{$(a,b)$}
    \choice{$(a,b]$}
    \choice[correct]{$[a,b)$}
    \choice{$[a,b]$}
\end{multipleChoice}
```

*Note that you can add additional* `\choice{AnswerOptions}` *to include more possible answer options; say if you wanted to add other values than* $a$ *or* $b$ *as possible endpoints.*

**Problem 35** *If you want to force students to determine if an endpoint is included or not (for example, if you are introducing the difference between brackets and parentheses, or doing an interval of convergence problem) you can use a mixture of open answer and multiple choice style dropdown option to build a multi-part answer.*

*We want the interval:* $[a, b)$

$(($ / $[$ ✓/ $)$ / $])$ $\boxed{a}$ , $\boxed{b}$ $(($ / $[$/ $)$ ✓/ $])$

*You can reproduce this answer with the following:*

```
\wordChoice{
    \choice{$($}
    \choice[correct]{$[$}
    \choice{$)$}
    \choice{$]$}
    }
$\answer{a}$ , $\answer{b}$
\wordChoice{
    \choice{$($}
    \choice{$[$}
    \choice[correct]{$)$}
    \choice{$]$}
    }
```

*Be careful to note where the mathmode delimiters are being used; the* `\wordChoice{}` *command needs to not be in mathmode, but* `\answer{}` *must be in mathmode.*

**Problem 36** *Finally, you can just give a number of answer blanks for your students:*

*We want the interval* $[a, b)$.

$\boxed{[}\boxed{a}, \boxed{b}\boxed{)}$

*You can reproduce this with:*

```
$\answer[format=string]{[}
\answer{a}$,
$\answer{b}
\answer[format=string]{)}$
```

*Notice the 'format=string' line in the answer boxes that contain the bracket or parentheses. This is necessary because these are otherwise special characters that will ruin the answer box validation process. This is also why it is absolutely necessary to separate any special characters (aka the brackets, braces, and/or parentheses) into their own answer box compared to any other content.*

## 15.11 Randomized Multiple Choice

*Workaround for Multiple Choice randomization*

As was noted in the section on multiple choice answer types, because multiple choice is a LaTeX level environment, it can't really deal with randomization very well. Indeed, you can randomize the content, but which answer is marked as "correct" is inevitably the same because it is flagged at the LaTeX level.

### Randomizing multipleChoice

Instead, we can design the problem by listing the possibilities, and then asking for which numbered entry is the correct one. For example:

**Problem 37** *Which of the following choices is* **??***?*

*1:* **??**

*2:* **??**

*3:* **??**

*4:* **??**

*5:* **??**

$\boxed{\textbf{??}}$

***Feedback(attempt):*** *Now hit the "Try Another" button in the top right corner and see how things move around, but it still works! Remember you can append ".tex" to the end of the url to get the code for the page, which you can use for a template if you want.*

### Randomizing Select All

You can also randomize select all in two ways. The first is to just do the same as above, but use the powers of 2 (or any base larger than 2) for the enumeration and ask for the sum, which would sum to a unique value. For example:

**Problem 38** *Which of the following choices is* ***not*** **??***?*

*1:* **??**

*2:* **??**

*4:* **??**

*8:* **??**

*16:* **??**

*Add up the numbers that correspond to the correct answers, and enter that sum into the box:* $\boxed{\textbf{??}}$

***Feedback(attempt):*** *Now hit the "Try Another" button in the top right corner and see how things move around, but it still works! Remember you can append ".tex" to the end of the url to get the code for the page, which you can use for a template if you want. Although the answer is computed by cheating and using geometric series formulas...*

Perhaps a more natural (and less prone to error) option however, is to use letters as the enumeration and then have students type in the string of those letters as so:

**Problem 39** *Which of the following choices is* **??** *and* **??***?*

*a* **??**

*b* **??**

*c* **??**

*d* **??**

*e* **??**

*Type in the letters that correspond to the answer - remember to just give the letters, no white space or commas or anything of the like.* $\boxed{\textbf{??}}$

***Feedback(attempt):*** *Now hit the "Try Another" button in the top right corner and see how things move around, but it still works! Remember you can append ".tex" to the end of the url to get the code for the page, which you can use for a template if you want. Hopefully the sage code isn't totally opaque as to how this works. Note that there is a very unintuitive requirement here to bypass how sage likes to make strings unreadable when called with the \sage command. In particular, instead of using something like 'a' as the string 'a' when generating the correct answer, you need to use: "LatexExpr(r"a")". Check the end of the sagesilent block by appending .tex to the url and see how it was generated to see how it is done.*

## 15.12 Equalities with Radicals

*A solution to the problem of making radical + value = radical + value style problems procedurally that remain suitably "nice" for precalc students.*

It is surprisingly difficult to random-generate a nice version of the classic $\sqrt{f(x)} = \sqrt{g(x)} + C$ style problems that maintain nice solveability by hand for a student at this level. Austin Jacobs and Jason Nowell coded this version that abuses the fact that we can force two parabola to intersect each other on an integer lattice in predictable places, then generate the radicals by using inverse functions of the parabolas while inferring the solutions by applying the inverse process to the integer lattice geometrically.

Apologies that the code is rather opaque - it requires a picture that no longer exists.

Below is the explanation for the code as per Austin,

**Explanation.** *The core idea here is that square roots with a linear term as their argument can be thought of as the "inverse" of a parabola. So what we are going to do is take two parabolas and then just draw a horizontal line at the y value we want to be our solution. The first parabola will be $F(x) = a1(x-b1)^2+c1$. As long as we choose a1, b1, and c1 to be integers, we know that putting in integers for x will force F(x) to be an integer (eg if $x = b1+2$ then $F(x) = a1*4+c1$). So we choose three integers and then pick a value for x-b1, which we'll call p1w. This will give us the point $(b1+p1w, p1h)$ with a guarantee of both coordinates being integers.*

*Now we draw our horizontal line $y = p1h$ which we know goes through that point (indeed, this will end up being our solution). We pick a horizontal distance to travel along this path, say C, and we specifically choose it to be an integer. This will be our "+C" in the initial problem statement.*

*We now have another integer point $(b1+p1w+C, p1h)$. This is our jumping point for the second parabola, G(x). The only thing we need to do now is pick a2, b2, c2, and d2, however we don't need to pick all of these. Once we make some choices the rest will be fixed. For simplicity, choose a2 and d2. This will let us know that our second parabola will have its vertext at $(b1+p1w+C-d2, p1h-a2*d2^2)$ and we only need to sort out from here what the appropriate b2 and c2 need to be. Since we have the vertex of the parabola, we know that b2 is the x value of that point and c2 is just the y value.*

*Finally, we get the display expressions by taking the function inverses of F and G.*

## The Finished Product

**Problem 40** *What is the **sum** of the answers to the following equality?*

$$?? = ??$$

*The sum of solutions is:* $\boxed{??}$.

***Feedback(attempt):*** *Recall that you need to isolate one of the radicals, then square both sides. Once you expand everything out, you will need to re-isolate the remaining radical and square both sides again, which should leave you with a quadratic. Once you solve the quadratic, you then need to check your solutions to see if they are extraneous or valid.*

## The magic

Here is the sage code that generates the problem... as mentioned it is currently rather opaque.

```
\begin{sagesilent}

def RandInt(a,b):
    """ Returns a random integer in [‘a‘,‘b‘]. Note that ‘a‘ and ‘b‘ should be integers themselves to avoid u
    """
    return QQ(randint(int(a),int(b)))
    # return choice(range(a,b+1))
```

```
def NonZeroInt(b,c, avoid = [0]):
    """ Returns a random integer in [‘b‘,‘c‘] which is not in ‘av‘.
        If ‘av‘ is not specified, defaults to a non-zero integer.
    """
    while True:
        a = RandInt(b,c)
        if a not in avoid:
            return a
```

```
p1ans1 = 100
p1ans2 = 200

while p1ans1>50 or p1ans2>50:
    # Make p(x)
    p1c1 = NonZeroInt(1,5)# a
    p1c2 = RandInt(1,5)# b
    p1c3 = RandInt(1,5)# c

    p1px = p1c1*(x-p1c2)^2 + p1c3

    p1w = RandInt(1,3)

    p1h = p1px(x=(p1c2+p1w))

    p1wprime = RandInt(2,5)

    p1bprime = RandInt(p1c2+1, 10)
    p1aprime = NonZeroInt(1,5,[p1c1])
    p1cprime = p1h - p1aprime*p1wprime^2

    p1gap = p1bprime + p1wprime -p1c2 - p1w#RandInt(p1bprime - p1c2 - p1w + 1, 2*(p1bprime - p1c2 - p1w + 1))


    p1left = sqrt((x - p1c3)/p1c1) + p1c2 + p1gap
    p1right = sqrt((x - p1cprime)/p1aprime) + p1bprime


    p1d = p1c2+p1gap-p1bprime
    p1e = p1c3*p1aprime-p1cprime*p1c1-p1d^2*p1c1*p1aprime

    p1A = (p1c1-p1aprime)^2
    p1B = 2*(p1c1-p1aprime)*p1e - 4*p1d^2*p1aprime^2*p1c1
    p1C = p1e^2 + 4*p1d^2*p1aprime^2*p1c1*p1c3

    p1ans1 = (-p1B + sqrt(p1B^2 - 4*p1A*p1C))/(2*p1A)
    p1ans2 = (-p1B - sqrt(p1B^2 - 4*p1A*p1C))/(2*p1A)

    if p1left(x=p1ans1) == p1right(x=p1ans1):
        if p1left(x=p1ans2)==p1right(x=p1ans2):
            p1ans = p1ans1+p1ans2
        else:
            p1ans=p1ans1
    else:
        p1ans=p1ans2
    if p1ans1==p1ans2:
        if p1left(x=p1ans1)==p1right(x=p1ans1):
            p1ans=p1ans1
        else:
            p1ans=LatexExpr(r"DNE")


\end{sagesilent}
```

## 15.13 Cubic with Two Factorable Derivatives

*How to generate "nice" polynomials that factor well, along with its first and second derivative.*

## The Problem

Often we want to have a polynomial whose first and second derivative are suitably "nice" for students in order to allow them to factor the polynomial itself, as well as its first and second derivatives, in order to find zeros for things like graphing. Unfortunately, it turns out, that creating a polynomial that is factorable, and whose first and second derivative are also factorable, is a *highly* nontrivial problem. Here we provide code that works (and generates *relatively* nice polynomials as well) to generate such a polynomial that is a cubic function (this also helps avoid requiring something like the rational root theorem for factoring the original polynomial - in this case the cubic is always factorable by grouping as well).

## The Result

The original functions is $f(x) =$ **??** which should factor into:

$$\textbf{??}\,(\textbf{??})$$

The first derivative is $f'(x) =$ **??** which should factor into

$$(\textbf{??}) \cdot (\textbf{??})$$

The second derivative is $f''(x) =$ **??** which is linear so it factors trivially (Yes, we kind of cheat here).

## The Magic

Here is the sagecode that generates the above problem:

```
\begin{sagesilent}
def RandInt(a,b):
    """ Returns a random integer in [`a`,`b`]. Note that `a` and `b` should be integers themselves to avoid u
    """
    return QQ(randint(int(a),int(b)))
    # return choice(range(a,b+1))

def NonZeroInt(b,c, avoid = [0]):
    """ Returns a random integer in [`b`,`c`] which is not in `av`.
        If `av` is not specified, defaults to a non-zero integer.
    """
    while True:
        a = RandInt(b,c)
        if a not in avoid:
            return a



## Start with a while loop to make sure the result has reasonable coefficients, at least in general size.
p1f2 = 9999*x^3 + 9999*x^2 + 9999*x + 9999

while ((abs(p1f2.coefficient(x^3))>100) or (abs(p1f2.coefficient(x^2))>100) or (abs(p1f2.coefficient(x))>100)
    ### We start by taking a product of factors to get a factorable first derivative.
    # Make sure the leading coefficient is divisible by 3, which will help ensure the numbers stay nice(ish)
    p1c1 = 1
    p1c3 = 1
    while mod(p1c1*p1c3,3)>0:
        p1c1 = NonZeroInt(-5,5)
        p1c2 = NonZeroInt(-5,5)
        p1c3 = RandInt(1,6)
        p1c4 = -sign(p1c1)*sign(p1c2)*RandInt(1,5)# Rigged sign to make sure we get a difference of squares i


    p1fact1 = p1c1*x-p1c2
    p1fact2 = p1c3*x-p1c4

    p1f1 = expand(p1fact1*p1fact2)



    ### Now we make the original function by integrating, then finding an appropriate ‘‘C’’ to add to make it

    p1f2temp = integral(p1f1,x)

    # Now, we assume we are going to factor by grouping, to be kind, so we extract the necessary constant we
    p1c5 = p1f2temp.coefficient(x^2)*p1f2temp.coefficient(x)/p1f2temp.coefficient(x^3)

    p1f2 = p1f2temp+p1c5

    if p1f2.coefficient(x^3)*p1f2.coefficient(x)<0:
        p1sqrtval = p1f2.coefficient(x)/p1f2.coefficient(x^3)
        p1f2fact1a = x - sqrt(abs(p1sqrtval))
        p1f2fact1b = x + sqrt(abs(p1sqrtval))
        p1f2fact1 = -sign(p1sqrtval)*(p1f2fact1a)*(p1f2fact1b)#(-sqrt(-p1f2.coefficient(x^3))*x + sqrt(p1f2.c
        p1f2fact2 = p1f2.coefficient(x^3)*x + p1f2.coefficient(x^2)
        p1f1zero1 = -p1f2fact1a(x=0)/p1f2fact1a.coefficient(x)
        p1f1zero2 = -p1f2fact1b(x=0)/p1f2fact1b.coefficient(x)
        p1f1zero3 = -p1f2fact2(x=0)/p1f2fact2.coefficient(x)
    else:
        p1f2fact1 = p1f2.coefficient(x^3)*x^2 + p1f2.coefficient(x)
        p1f2fact2 = x + p1f2.coefficient(x^2)/p1f2.coefficient(x^3)
        p1f1zero1 = 0
        p1f1zero2 = 0
        p1f1zero3 = -p1f2fact2(x=0)/p1f2fact2.coefficient(x)

    p1f2check = expand(p1f2fact1*p1f2fact2)




    ### To get the second derivative function we can take the derivative of the original - which must be line

    p1f3 = derivative(p1f1,x)

\end{sagesilent}
```