Table of Contents

SketchResponse

SketchResponse is a Javascript/Python(2.5-2.7) tool for drawing and evaluating mathematical functions. It was designed for use with the edX online courseware platform. However, it is a self-contained application that can be integrated into other web platforms.

Demo

Before we get into the technical details, try out a SketchResponse Demo to get a sense of what SketchResponse can do.

Motivation

TODO Martin said he would write a paragraph.

Features

- Sketch Tool - configurable Javascript front-end with plugin components to provide different function drawing and annotation capabilities. See Sketch-Tool Plugin Configuration for a description of the available plugins.
- Grader Library - python back-end that provides an API of function grading methods that can be composed to construct custom grading scripts. See Create a Simple Grading Script for a tutorial on building a simple grading script.

Read More

- Sketch Tool
- Preliminary results using SketchResponse on edX

Getting started from zero

Prerequisites

To work with the SketchResponse codebase, there are a few technology prerequisites that you will need installed on your machine.

- In order to clone the repository you will need to install Git on your computer. You may also want to create a Github account, though you should not need one to simply clone the SketchResponse repository from the command line. You will need to be logged into a Github account if you want to clone the repository through your browser. You will also need a Github account if you want to contribute changes back to the project.
- The Sketch Tool front-end is built using Node.js and npm (which is now packaged with Node). You will need to have both of these installed on your computer.
- The Grader Library back-end is built using Python and relies on a couple python packages that you will need to install. We recommend you use the pip python package installer. If you are using an up-to-date python installation you probably already have it installed. If you already have another python package manager, we only use commonly available packages so it should not be a problem.

..* Note for Windows Users: When installing Python make sure to enable the option to add Python to the path. This is disabled by default for some reason.

- You will also need to be comfortable using the command-line to execute commands.

.. *Note for Windows Users: You should use the Git Bash shell to execute all commands in this document. You will need to run it as Administrator to execute the* npm *and* pip* installation commands below. To do this, right-click Git Bash from the Start Menu and select Run as Administrator.

If you have statisfied all of the above prerequisites, then we can start working on the codebase.

Getting the codebase

- Clone this repository to the directory of your choice:
  - By using HTTPS:

```
$ git clone https://github.com/SketchResponse/sketchresponse.git
```

  - By using SSH:

```
$ git clone git@github.com:SketchResponse/sketchresponse.git
```

Running either of the above commands will create a new directory called SketchResponse. This SketchResponse directory is your copy of the repository. The rest of this document will refer to it as the *repo* directory.

- Run this command to change directories to the *repo* of the SketchResponse repository.

```
$ cd SketchResponse
```

Building the Sketch Tool front-end

- Change to the *sketch_tool* directory:

```
$ cd sketch_tool
```

- Install dependencies listed in *package.json* using `npm`:

```
$ npm install
```

  JSPM will pull in additional dependencies automatically on a post-install script.

- Build a distribution of the Sketch Tool. The distribution directory will be found in the `/static/sketch_tool_dist/` directory.

```
$ npm run build
```

Now you have a built distribution of the Sketch Tool front-end, we will use it later to let us test grading scripts. The front-end distribution is also what you will need to have hosted on a public server when deploying SketchResponse for your application.

Grader Library back-end dependencies

The SketchResponse grader library back-end has only two third party package requirements, which are listed in the requirements.txt file.

- flask
- numpy

You will need to be in the *repo* directory of the codebase. If you are still in the sketch_tool directory, run the following command to change back to the *repo* directory.

```
$ cd ..
```

Now that you are in the *repo* directory, you can run the following command to install the required python packages.

```
$ pip install -r requirements.txt
```

Numpy is used to support the mathematical computations of the grader library.

Flask is required to run a local server for convenience when implementing and testing new grading scripts. This lets you test your scripts without having to deploy them to another server, giving you more immediate feedback and simplifying the debugging process. See the Tutorial Test a Grading Script on a Local Server for details.

What's next

Now that you have the codebase installed and ready to use, lets start looking at how to configure the Sketch Tool and implement a couple grading scripts. Click here.

Using SketchResponse

Using SketchResponse means writing python scripts to provide a configuration for the Sketch Tool front-end and an implementation of the grader method using the Grader Library back-end API. We call these *grading scripts*.

What is a grading script

Before we get started writing scripts, we want to familiarize ourselves with the structure of a grading script. Read a description of the grading script template here

Writing our first grading script

Now that we know the pieces that must be implemented in the grading script, lets get started by implementing an extremely simple script. All this script will do is using the Grader Library to evalutate whether or not drawn function represents a straight line, or line segment. Click here to follow this tutorial.

Writing a more complicated script

Okay that was pretty easy, lets try building something a little more realistic. This grader script will be for a relatively simple polynomial function. Click here to follow this tutorial.

Testing your grading scripts

As you probably saw at the end of each of our grading script examples, there was a brief description of testing that example script using a local server. More details on how to run that process can be found here.

Where to go from here

As you might expect, there are a lot of different types of function sketches that can be grading using this tool, however the current grading library is focused on introductory calculus type problems. We are working to expand the scope of the functions that can be sketched and graded, but it is a work in progress.

We have only demonstrated the use of a handful of the grading functions provided by the Grader Library Application Programming Interface(API). Explore the API to see what other grading options are available.

Grading script template

A grader script template can be see below. There are two components of the grader script:

1. Sketch Tool configuration
2. grader method implementation

The details of each of these components are examined in greater detail in the simple and complex(complex_grader.md) grading script examples. The template is provided here for reference.

You can also find a copy in */grader_scripts/grader_template.py*.

```
import sketchinput
from draft_code import GradeableFunction, Asymptote

problemconfig = sketchinput.config({
    'width': 750,  # set the pixel width of the front-end interface
    'height': 420,  # set the pixel height of the front-end interface
    'xrange': [-2.35, 2.35],  # set the x-axis displayed range
    'yrange': [-1.15, 1.15],  # set the y-axis displayed range
    'xscale': 'linear',  # only linear is currently implemented
    'yscale': 'linear',
    'plugins': [
        # all instances will use at minimum these two plugins
        {'name': 'axes'},
        {'name': 'freeform', 'id': 'f', 'label': 'Function f(x)',
         'color':'blue'},
    ]
})


@sketchinput.grader
def grader(f):  # arguments is a list of the 'id' values for each plugin

    return True, 'Good Job'
```

A simple grader script

This document will walk through the implementation of a grader script for a simple problem. All this grader will do is test whether the input function is of a straight line.

Each grader script at its base is composed of two components

```
1. The problem configuration
2. The grader function
```

Imports

There are two SketchResponse python modules that must be imported for this simple example. All grader scripts must import the `sketchresponse` module. There are two other modules that provide different grading helper functions. In this case, we only need to input the `GradeableFunction` module from `grader_lib`.

```
import sketchresponse
from grader_lib import GradeableFunction
```

Problem configuration

The problem configuration is passed to the javascript front end to define the size and scale of the drawing space and to define which drawing tools are available for the problem. The `sketchresponse.config()` function takes a dict of configuration options.

In the example configuration below, the first six key/value pairs are required:

- `'width': 750` sets the pixel width of the drawing space as 750 pixels
- `'height': 420` sets the pixel height of the drawing space to 420 pixels
- `'xrange': [-2.35, 2.35]` sets the numerical range of the x axis
- `'yrange': [-1.15, 1.15]` sets the numerical range of the y axis
- `'xscale': 'linear'` sets the scale of the x axis to linear (only option currently implemented)
- `'yscale': 'linear'` sets the scale of the y axis to linear (only option currently implemented)

The last entry `'plugins'` takes a list of dicts that enable the specific javascript plugins that are available to the user. All plugins are declared by 'name'.

The 'axes' plugin entry is the simplest plugin to enable. It has no configuration options so all that must be set is the 'name'. This plugin enables the axes in the drawing space. *It should probably be on by default no?*

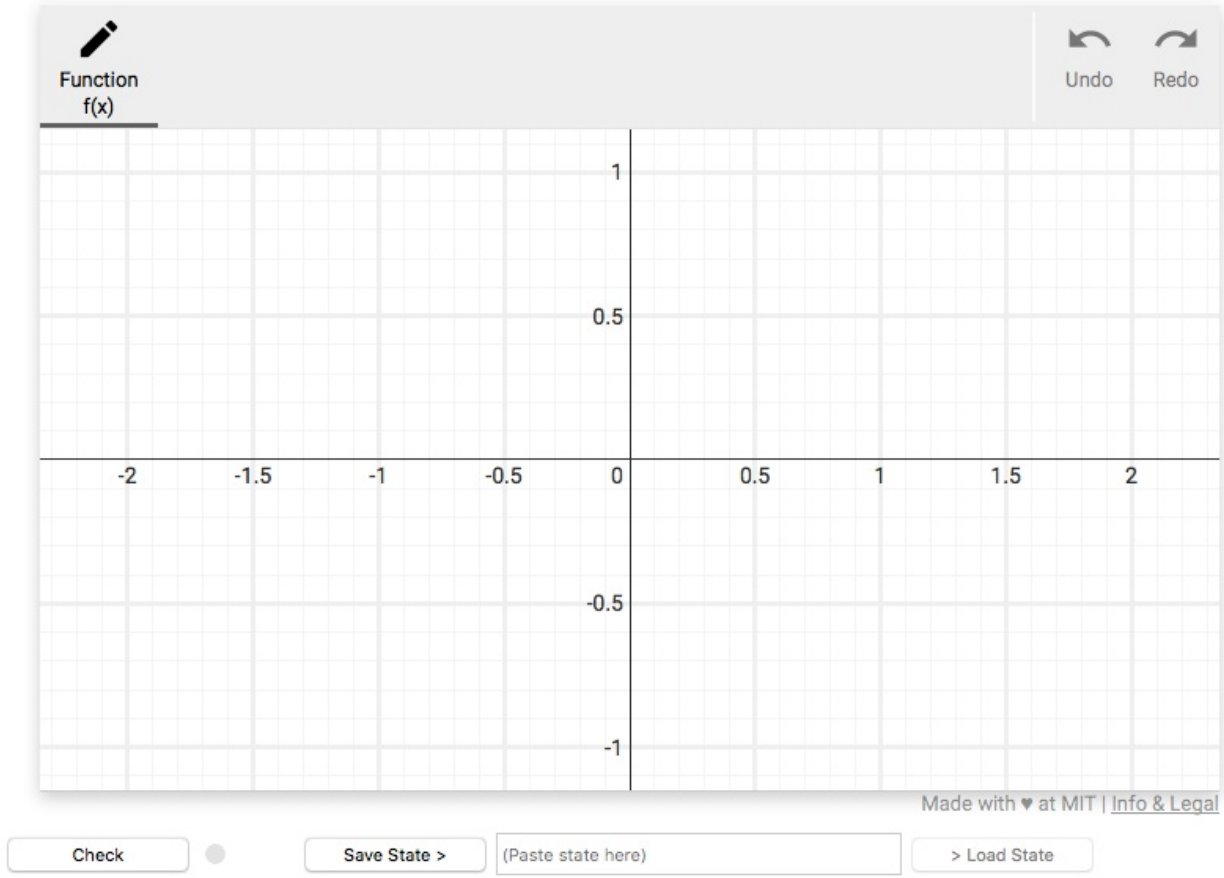
The 'freeform' plugin entry enables the freeform drawing tool. It has three configuration options to set:

- `'id'` sets the name of the argument of the grader callback function (described in the next section) to which the data generated by this plugin is passed.
- `'label'` is the name of the tool displayed to the user.
- `'color'` is the color used to render the drawn function.

A listing of all the built-in plugins can be found at SketchResponse Plugins.

```
problemconfig = sketchresponse.config({
    'width': 750,
    'height': 420,
    'xrange': [-2.35, 2.35],
    'yrange': [-1.15, 1.15],
    'xscale': 'linear',
    'yscale': 'linear',
    'plugins': [
        {'name': 'axes'},
        {'name': 'freeform', 'id': 'f', 'label': 'Function f(x)', 'color':'blue'},
    ]
})
```

The above problem configuration settings will create a javascript tool that looks something like the image below.



Define the grader callback function

```
@sketchresponse.grader
def grader(f):
    gf = GradeableFunction.GradeableFunction(f)

    if not gf.is_straight():
        return False, 'Not straight'

    return True, 'Good Job'
```

The grader callback function implements the function passed to the sketchinput grader to evaluate the data sent from the javascript tool.

The arguments of the grader function are the `'id'` values as defined in the problem configuration above. E.g. in our problem configuration we enabled the freeform drawing tool with id 'f' and we have a corresponding argument f in the signature of the function that will be automatically unpacked.

Before we can execute any grading helper functions on the data, we must instantiate the data as a `GradeableFunction`.

In this simple example all we are checking is that the submitted function defines a straight line over its entire domain. We are not checking for slope of the line. To do this we call the grader helper function `gf.is_straight()`. `is_straight()` returns a boolean value. The full API documentation for the grader helper functions can be found at SketchResponse API.

And that's it! Those two simple blocks of code complete our first grader script. Admittedly this particular script doesn't do much. Check out the Complex Grader example for a more realistic grader tutorial on an example math problem.

Testing the script

Once the script is written, you can run the script in the local testing server. See the Test a Grading Script on a Local Server tutorial for details on installing and running the testing server.

There is already a copy of this grader script in the `grader_scripts` directory so all you need to do is start the server and point your browser of choice at the url:

```
http://localhost:5000/simple_grader
```

You should see the configured Sketch Tool. If you draw a straight(ish) line and press the check button you will get accept message. If the line is not straight enough, you will get a reject message.

A more realistic grader script

This document will walk through the implementation of a grader script for a more complicated problem than the Simple Grader Tutorial tutorial. The problem description for this grader script is below.

Sketch the function $f(x) = 2x^2 / x^2 - 1$.

Label vertical and horizontal asymptotes, extrema, and inflection points.

This problem will introduce new javascript drawing plugins to handle labeling asymptotes and points and some new backend API functions that can be used to evaluate these new sources of data.

Imports

There are three SketchResponse python modules that must be imported for this simple example. As you saw in the Simple Grader Tutorial all grader scripts must import the `sketchresponse` module. We again need to input the `GradeableFunction` module from `grader_lib`, but also need to import the `Asymptote` module to support the asymptote labeling task.

```
import sketchresponse
from grader_lib import GradeableFunction, Asymptote
```

Problem configuration

The problem configuration is passed to the javascript front end to define the size and scale of the drawing space and to define which drawing tools are available for the problem. The `sketchresponse.config()` function takes a dict of configuration options.

In the example configuration below, the first six key/value pairs are required. In this configuration we increase the ranges of the X and Y axes compared to the Simple Grader Tutorial:

- `'width': 750` sets the pixel width of the drawing space as 750 pixels
- `'height': 420` sets the pixel height of the drawing space to 420 pixels

- `'xrange': [-3.5, 3.5]` sets the numerical range of the x axis
- `'yrange': [-4.5, 4.5]` sets the numerical range of the y axis
- `'xscale': 'linear'` sets the scale of the x axis to linear (only option currently implemented)
- `'yscale': 'linear'` sets the scale of the y axis to linear (only option currently implemented)

The last entry `'plugins'` takes a list of dicts that enable the specific javascript plugins that are available to the user. All plugins are declared by 'name'.

The 'axes' and 'freeform' plugin usage here is identical to the Simple Grader Tutorial and is explained there as well as on the [Plugin Description Page(probconfig_plugins.md).

There are three new plugins introduced in this grading script: 'vertical-line', 'horizontal-line', and 'point'. The declaration of these plugins is very similar to the declaration of the 'freeform' plugin, however, each type of plugin has one additional parameter that needs to be defined: 'dashStyle' for the lines, and 'size' for the points.

The 'vertical-line' plugin entry enables the a labeling tool for vertical asymptotes. It has four configuration options to set:

- `'id'` sets the name of the argument of the grader callback function (described in the next section) to which the data generated by this plugin is passed.
- `'label'` is the name of the tool displayed to the user.
- `'color'` is the color used to render the drawn function.
- `'dashStyle'` is the style of dashed line to used to draw the asymptote.

The 'horizontal-line' plugin entry enables a labeling tool for horizontal asymptotes. Its configuration options mirror the 'vertical-line'.

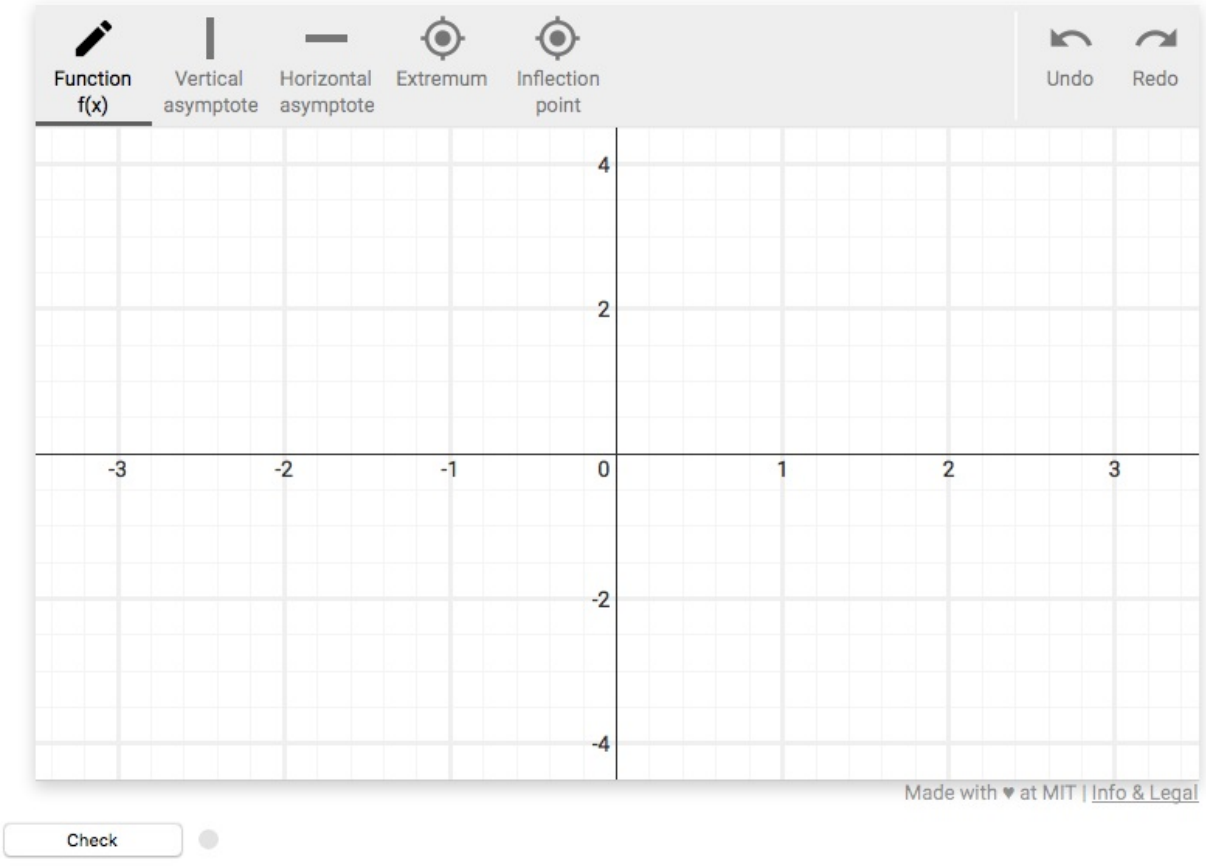
In this problem, we want students to label both extrema points and the inflection point for the function. To do this we can declare two instances of the 'point' plugin with different 'id' and 'label' values. The 'point' plugin has four configuration options to set:

- `'id'` sets the name of the argument of the grader callback function (described in the next section) to which the data generated by this plugin is passed.
- `'label'` is the name of the tool displayed to the user.
- `'color'` is the color used to render the drawn function.
- `'size'` is the pixel width of the point drawn by the plugin.

A listing of all the built-in plugins can be found at SketchResponse Plugins.

```
problemconfig = sketchresponse.config({
    'width': 750,
    'height': 420,
    'xrange': [-3.5, 3.5],
    'yrange': [-4.5, 4.5],
    'xscale': 'linear',
    'yscale': 'linear',
    'plugins': [
        {'name': 'axes'},
        {'name': 'freeform', 'id': 'f', 'label': 'Function f(x)', 'color':'blue'},
        {'name': 'vertical-line', 'id': 'va', 'label': 'Vertical asymptote', 'color': 'gra
y', 'dashStyle': 'dashdotted'},
        {'name': 'horizontal-line', 'id': 'ha', 'label': 'Horizontal asymptote', 'color':
'gray', 'dashStyle': 'dashdotted'},
        {'name': 'point', 'id': 'cp', 'label': 'Extremum', 'color': 'black', 'size': 15},
        {'name': 'point', 'id': 'ip', 'label': 'Inflection point', 'color':'orange','size'
: 15}
    ]
})
```

The above problem configuration settings will create a javascript tool that looks something like the image below.



Define the grader callback function

Handling the input data

```
@sketchresponse.grader
def grader(f,cp,ip,va,ha):

    f = GradeableFunction.GradeableFunction(f)
    cp = GradeableFunction.GradeableFunction(cp)
    va = Asymptote.VerticalAsymptotes(va)
    ha = Asymptote.HorizontalAsymptotes(ha)
    ip = GradeableFunction.GradeableFunction(ip)

    msg=''
```

The first thing that the grader function needs to do is format the input data so that Grader Library API methods of interest can be used to check specific features of the input data.

As explained in the Simple Grader Tutorial the data is passed to the grader function as a dictionary with keys equal to the 'id' values used in the pluginconfig above so the dictionary can be directly unpacked into argument variables with the same names.

The 'freeform' and 'point' data are evaluated with API methods in the `GradeableFunction.GradeableFunction` class. The 'vertical-line' and 'horizontal-line' data are evaluted with API methods in the `Asymptote.VerticalAsymptote` and 'Asymptote.HorizontalAsymptote` classes respectively.

The `msg` variable will be used when evaluating specific checks on the data to supply check specific feedback to the student if errors are found.

Checking the number of extrema points labeled

```
if cp.get_number_of_points() != 1:
   if cp.get_number_of_points() == 3:
      msg += '<font color="blue">Are you sure about the number of extrema? (note that you
should not label the endpoints of your function)</font><br />'
   else:
    msg += '<font color="blue">Are you sure about the number of extrema?</font><br />'
```

This first check verifies that there is only one extremum point labeled in the input data. It also checks for a common error case where students label the end points of their function to provide more helpful feedback.

Checking the number of inflection points

```
if ip.get_number_of_points() != 0:
   msg += '<font color="blue">Are you sure about the number of extrema?</font><br />'
```

This particular function does not have any inflection points so this check verifies that the student did not put any unnecessary labels.

Checking the number of asymptotes

```
if va.get_number_of_asyms() != 2:
   msg += '<font color="blue"> Are you sure about the number of vertical asymptotes?</font
><br />'


if ha.get_number_of_asyms() != 1:
   msg += '<font color="blue"> Are you sure about the number of horizontal asymptotes?</fo
nt><br />'
```

This function should have 2 vertical asymptote labels and 1 horizontal asymptote label. If either of theses checks fail, appropriate feedback messages are provided.

Checking the position of the extremum point

```
if not cp.has_point_at(x=0):
   msg += '<font color="blue"> Check the x value of your critical point</font><br />'
```

The extremum point label for the function should be at position `x=0`.

Checking the positions of the asymptotes

```
if not va.has_asym_at_value(-1) or not va.has_asym_at_value(1):
   v1 = va.closest_asym_to_value(-1)
   v2 = va.closest_asym_to_value(1)
   msg += '<font color="blue"> Check the locations of your vertical asymptotes.  </font><b
r />'

if not ha.has_asym_at_value(2):
   ha1 = ha.closest_asym_to_value(2)
   msg += '<font color="blue"> Check the locations of your horizontal asymptotes. </font><
br />'
```

The vertical asymptotes should be at positions `x=-1` and `x=1`. The horizonal asymptote should be at the position `y=2`. As you can see, you can also get references to the actual asymptote values if you wanted to use them for further checks, or to customize your feedback messages.

Checking that the extremum point is on the freeform line

```
maxpt = cp.get_point_at(x=0)

if not f.has_value_y_at_x(maxpt.y, maxpt.x):
   msg += '<font color="blue"> Make sure your critical points lie on your function!</font>
<br />'
```

Here we get a reference to the extremum point and use that point's x and y coordinates to make sure that it is sitting on the freeform line of the function. The grader method does use a configurable threshold value to ensure pixel perfect placement is not necessary to pass this kind of check.

Checking the increasing and decreasing ranges of the freeform line

```
increasing_ok = f.is_increasing_between(-4, -1) and f.is_increasing_between(-1, 0)
decreasing_ok = f.is_decreasing_between(0, 1) and f.is_decreasing_between(1, 4)
if not (increasing_ok and decreasing_ok):
   msg += '<font color="blue"> Where should the graph be increasing and decreasing?</font>
<br />'
```

The freeform line drawn by the student should be increasing over the ranges `(-4,-1)` and `(-1,0)`. It should also be decreasing over the ranges `(0,1)` and `(1,4)`. Again these grading methods have configurable thresholds to ensure they don't unfairly fail student input.

Checking the value of the freeform line over specific ranges

```
if not f.is_greater_than_y_between(2,-4,-1):
   msg += '<font color="blue"> Your function seems to be in the wrong region on the interv
al (-4,-1)</font><br />'

if not f.is_greater_than_y_between(2,1,4):
   msg += '<font color="blue"> Your function seems to be in the wrong region on the interv
al (1,4)</font><br />'

if not f.is_less_than_y_between(0,-1,1):
   msg += '<font color="blue"> Your function seems to be in the wrong region on the interv
al (-1,1)</font><br />'
```

Here we are performing three sanity checks on the values of the freeform line over specific ranges. First, we check that `f(x) >= 2` over the range `(-4,-1)`. Second, we check that `f(x) >= 2` over the range `(1,4)`. Lastly, we check that `f(x) <= 0` over the range `(-1,1)`.

Checking the curvature of the freeform line

```
curvature_up_ok = f.has_positive_curvature_between(-4, -1) and f.has_positive_curvature_be
tween(1, 4)
curvature_down_ok= f.has_negative_curvature_between(-1,1)

if not (curvature_up_ok and curvature_down_ok):
   msg += '<font color="blue"> Where is the function convave up and concave down?</font><b
r />'
```

The final checks we will perform make sure that the freeform line has the expected curvature over specific ranges. The curvature should be positive over the ranges `(-4,-1)` and `(1,4)`. The curvature should be negative over the range `(-1,1)`.

Putting it all together

Combining all the code above into a single function gives us the following. You will notice that the error message variable `msg` is tested at multiple points during the evaluation and used as an early failure condition. If the numbers of expected labels are not correct, then future checks are likely to not be able to run on the data so returning early ensures the student gets good feedback.

```
import sketchresponse
from grader_lib import GradeableFunction, Asymptote
```

```
problemconfig = sketchresponse.config({
    'width': 750,
    'height': 420,
    'xrange': [-3.5, 3.5],
    'yrange': [-4.5, 4.5],
    'xscale': 'linear',
    'yscale': 'linear',
    'plugins': [
        {'name': 'axes'},
        {'name': 'freeform', 'id': 'f', 'label': 'Function f(x)', 'color':'blue'},
        {'name': 'vertical-line', 'id': 'va', 'label': 'Vertical asymptote', 'color': 'gra
y', 'dashStyle': 'dashdotted'},
        {'name': 'horizontal-line', 'id': 'ha', 'label': 'Horizontal asymptote', 'color':
'gray', 'dashStyle': 'dashdotted'},
        {'name': 'point', 'id': 'cp', 'label': 'Extremum', 'color': 'black', 'size': 15},
        {'name': 'point', 'id': 'ip', 'label': 'Inflection point', 'color':'orange','size'
: 15}
    ]
})

@sketchresponse.grader
def grader(f,cp,ip,va,ha):

    f = GradeableFunction.GradeableFunction(f)
    cp = GradeableFunction.GradeableFunction(cp)
    va = Asymptote.VerticalAsymptotes(va)
    ha = Asymptote.HorizontalAsymptotes(ha)
    ip = GradeableFunction.GradeableFunction(ip)

    msg=''

    if cp.get_number_of_points() != 1:
        if cp.get_number_of_points() == 3:
            msg += '<font color="blue">Are you sure about the number of extrema? (note tha
t you should not label the endpoints of your function)</font><br />'
        else:
            msg += '<font color="blue">Are you sure about the number of extrema?</font><br
 />'


    if ip.get_number_of_points() != 0:
        msg += '<font color="blue">Are you sure about the number of extrema?</font><br />'

    if va.get_number_of_asyms() != 2:
        msg += '<font color="blue"> Are you sure about the number of vertical asymptotes?<
/font><br />'


    if ha.get_number_of_asyms() != 1:
        msg += '<font color="blue"> Are you sure about the number of horizontal asymptotes
?</font><br />'
```

```
    if msg != '':
        return False, msg
    else:
        if not cp.has_point_at(x=0):
            msg += '<font color="blue"> Check the x value of your critical point</font><br
 />'
        if not va.has_asym_at_value(-1) or not va.has_asym_at_value(1):
            v1 = va.closest_asym_to_value(-1)
            v2 = va.closest_asym_to_value(1)
            msg += '<font color="blue"> Check the locations of your vertical asymptotes.
</font><br />'

        if not ha.has_asym_at_value(2):
            ha1 = ha.closest_asym_to_value(2)
            msg += '<font color="blue"> Check the locations of your horizontal asymptotes.
 </font><br />'

        maxpt = cp.get_point_at(x=0)

        if not f.has_value_y_at_x(maxpt.y, maxpt.x):
            msg += '<font color="blue"> Make sure your critical points lie on your functio
n!</font><br />'

        increasing_ok = f.is_increasing_between(-4, -1) and f.is_increasing_between(-1, 0)
        decreasing_ok = f.is_decreasing_between(0, 1) and f.is_decreasing_between(1, 4)
        curvature_up_ok = f.has_positive_curvature_between(-4, -1) and f.has_positive_curv
ature_between(1, 4)
        curvature_down_ok= f.has_negative_curvature_between(-1,1)


        if not (increasing_ok and decreasing_ok):
            msg += '<font color="blue"> Where should the graph be increasing and decreasin
g?</font><br />'

        if not f.is_greater_than_y_between(2,-4,-1):
            msg += '<font color="blue"> Your function seems to be in the wrong region on t
he interval (-4,-1)</font><br />'

        if not f.is_greater_than_y_between(2,1,4):
            msg += '<font color="blue"> Your function seems to be in the wrong region on t
he interval (1,4)</font><br />'

        if not f.is_less_than_y_between(0,-1,1):
            msg += '<font color="blue"> Your function seems to be in the wrong region on t
he interval (-1,1)</font><br />'


        if not (curvature_up_ok and curvature_down_ok):
            msg += '<font color="blue"> Where is the function convave up and concave down?
</font><br />'

    if msg == '':
```

```
            return True,'Good Job'
        else:
            return False, msg
```

Testing the script

Once the script is written, you can run the script in the local testing server. See the Test a Grading Script on a Local Server tutorial for details on installing and running the testing server.

There is already a copy of this grader script in the `grader_scripts` directory so all you need to do is start the server and point your browser of choice to the url:

```
http://localhost:5000/complex_grader
```

You should see the configured Sketch Tool. If you sketch the function shown at the beginning of this tutorial, you should see an accept message. Any other sketches should return a reject message.

How To Test Your Grading Scripts Locally

The SketchResponse tool is designed to be used as a web application so when it is properly installed the Sketch Tool and Grader Lib will be hosted on a server somewhere. However, while you are designing and implementing a grading script for a particular problem, it is very convenient to be able to test it on a locally running version of the SketchResponse tool. This allows you to get immediate feedback on how well your grading script works without having to upload anything to your server.

Flask

The local testing server is implemented using Flask. If you do not have flask you will need to install it, which is easily done by running the command:

```
$ pip install -r requirements.txt
```

Running the server

Building the sketch tool distribution

Before you can run the local server, you need to build a local copy of the sketch tool. If you have already done this and you have not modified the sketch tool in the interim you can skip this step. Otherwise, follow the instructions here for details on building the sketch tool distribution.

Start the grader local server

To run the local server, make sure the grading script you are testing is in the `grading_scripts` directory. Then run the following command from the repository root directory:

```
$ python server.py
```

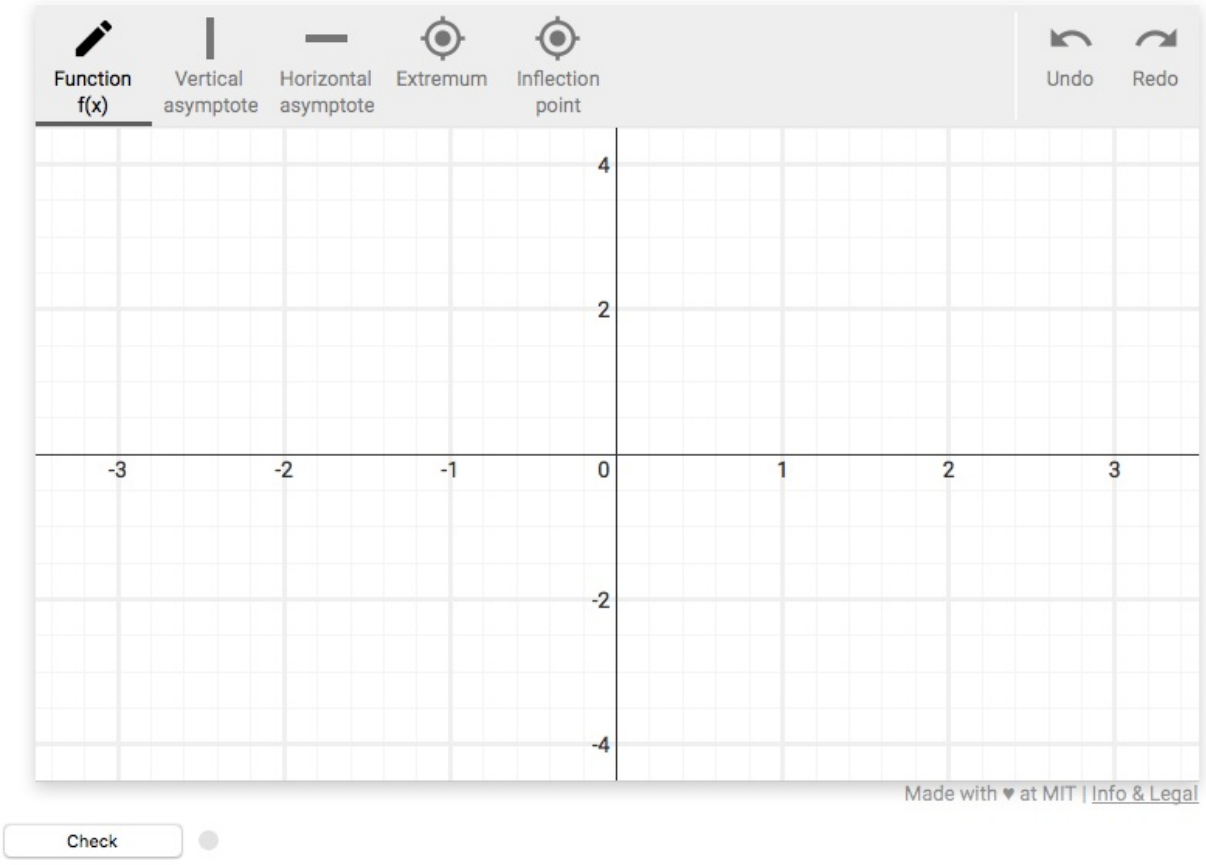
You should see a message telling you that the server is running on `http://localhost:5000`.

Testing your script

Open your web browser of choice and put the following in the url bar, where is the filename of your grader script (leaving the .py extension off):

```
$ http://localhost:5000/<grader script name>
```

You will now see a locally running copy of the SketchResponse Sketch Tool using the configuration options you defined in your grading script. You can use it to draw test input for your grader script and press the 'Check' button in the bottom left corner to run your grader on the functions you have drawn.



Deploying SketchResponse

SketchResponse was designed to be a stand-alone application that should be relatively easy to integrate into whatever application you have. The details of deploying to custom applications are unfortunately beyond the scope of this documentation. Needless to say, you will need to provide hosting for the Sketch Tool front-end distribution, the python Grader Library back-end, and a small amount of static html to tie it all together (see the */static/* directory and */templates/index_sketch_tool.html* in the codebase for a simple example of this).

However, as SketchResponse was designed with the intention of using it to develop online courses for mitX and edX, there is a quickstart guide to deploying your SketchResponse problems to edX below.

edX Quick Start Guide to Deploying SketchResponse Problems

This document is a brief guide on how to use SketchResponse with edX. There are two sets of information that you need to deploy:

1. the python grader library that support your grading scripts
2. the problem XML (which will include your grading scripts).

Uploading the Grader Library

1. Run the following command from the root of the SketchResponse repository:

   ```
   $ python edxzip.py
   ```

   This will create a zip archive called `python_lib.zip` in the root directory of the repository.

2. In edX Studio, from the "Content" menu at the top of the page, select "Files & Uploads".

3. Click the green "+ Upload New File" button and upload the `python_lib.zip` file.

Notes

If your workflow involves importing an XML course, you may still need to follow these steps in Studio; `python_lib.zip` may not be detected automatically in the static/ folder of your course. The steps are not required every time you upload your course, though; `python_lib.zip` won't be deleted by course uploads.

Creating SketchResponse problems

1. Create a new Advanced Problem in Studio (or a if authoring in XML).
2. Use the following problem template, replacing the appropriate sections with your own content:

```
    <problem>
    <p>Replace this text with your own problem description.</p>

    <script type="loncapa/python"><![CDATA[

    # Include your grading script here (beginning with import sketchresponse)

    ]]></script>

    <customresponse cfn="grader" expect="See solution.">
        <jsinput width="800" height="550" gradefn="getGrade" get_statefn="getState" set_st
atefn="setState" html_file="TODO.html#$problemconfig" sop="false"/>
    </customresponse>


    <solution>

    <p>Provide the solution to your problem here. You may want to include an image of a co
rrect solution as well as text.</p>

    </solution>


    </problem>
```

Notes

Unlike most other input types, you cannot combine SketchResponse problems with other input types (or other SketchResponse inputs) in a single tag; doing so will lead to students losing attempts and other unexpected behavior.

Problem Configuration Plugins

This document provides a description of all the SketchResponse plugins and how to declare them. Declaring any plugin in a grader's problem configuration enables the selection of that plugin in the javascript front-end tool.

Any of these plugins can be declared multiple times causing multiple instances of the plugin to be enabled in the javascript tool. See the Complex Grader Example for an example where this is used.

Table of Contents

Axes

Adds horizontal and vertical axes with major and minor ticks and their associated gridlines.

- `'name': 'axes'` - the name key *must* have the value 'axes'
- `'xmajor': <number>(default: 1)` - the major tick spacing for the x axis
- `'ymajor': <number>(default: 1)` - the major tick spacing for the y axis
- `'xminor': <number>(default: 0.25)` - the minor tick spacing for the x axis
- `'yminor': <number>(default: 0.25)` - the minor tick spacing for the y axis

E.g.

```
{'name': 'axes'}
```

Background

Sets the background image for the drawing canvas. The default is an grid.

E.g.

```
{'background': '/static/app/axes.png'}
```

Freeform

The Freeform plugin adds a button to the tool to draw freeform lines on the axes. It has four parameters that must be defined:

- `'name': 'freeform'` - the name key *must* have the value 'freeform'
- `'id': <unique identifier string>` - the id key *must* have a *unique* value. This value is used as the key for the data created by this plugin in the JSON string returned to the grader function.
- `'label': <descriptive string>` - the label key should be given a descriptive string. This string will be used to label the selection button in the javascript front-end tool.
- `'color': <a color string>` - the color key should be give a color string that javascript recognizes. A listing of color names can be found here.

E.g.

```
{'name': 'freeform', 'id': 'f', 'label': 'Function f(x)', 'color':'blue'}
```

Point

The Point plugin adds a button to the tool to draw points on the axes. It has five parameters that must be defined:

- `'name': 'point'` - the name key *must* have the string value 'point'
- `'id': <unique identifier string>` - the id key *must* have a *unique* value. This value is used as the key for the data created by this plugin in the JSON string returned to the grader function.
- `'label': <descriptive string>` - the label key should be given a descriptive string. This string will be used to label the selection button in the javascript front-end tool.
- `'color': <a color string>` - the color key should be give a color string that javascript

recognizes. A listing of color names can be found here.

- `'size': <int>` - the size key must be given an integer value. It sets the pixel diameter of the point drawn by the plugin.

E.g.

```
{'name': 'point', 'id': 'cp', 'label': 'Extremum', 'color': 'black', 'size': 15}
```

Vertical Asymptote

The Vertical Asymptote plugin adds a button to the tool to draw vertical lines on the axes. It has five parameters that must be defined:

- `'name': 'vertical-line'` - the name key *must* have the value 'vertical-line'
- `'id': <unique identifier string>` - the id key *must* have a *unique* value. This value is used as the key for the data created by this plugin in the JSON string returned to the grader function.
- `'label': <descriptive string>` - the label key should be given a descriptive string. This string will be used to label the selection button in the javascript front-end tool.
- `'color': <a color string>` - the color key should be give a color string that javascript recognizes. A listing of color names can be found here.
- `'dashStyle': <line dash string>(default: 'solid')` - the dashStyle key should have a string description of the dash style to used for drawing the line. Possible values: 'dashed', 'longdashed', 'dotted', 'dashdotted', 'solid'.

```
{'name': 'vertical-line', 'id': 'va', 'label': 'Vertical asymptote', 'color': 'gray', 'das
hStyle': 'dashdotted'}
```

Horizontal Asymptote

The Horizontal Asymptote plugin adds a button to the tool to draw horizontal lines on the axes. It has five parameters that must be defined:

- `'name': 'horizontal-line'` - the name key *must* have the value 'horizontal-line'
- `'id': <unique identifier string>` - the id key *must* have a *unique* value. This value is used as the key for the data created by this plugin in the JSON string returned to the grader function.
- `'label': <descriptive string>` - the label key should be given a descriptive string.

This string will be used to label the selection button in the javascript front-end tool.

- `'color': <a color string>` - the color key should be give a color string that javascript recognizes. A listing of color names can be found here.
- `'dashStyle': <line dash string>(default: 'solid')` - the dashStyle key should have a string description of the dash style to used for drawing the line. Possible values: 'dashed', 'longdashed', 'dotted', 'dashdotted', 'solid'.

```
{'name': 'horizontal-line', 'id': 'ha', 'label': 'Horizontal asymptote', 'color': 'gray',
'dashStyle': 'dashdotted'}
```

Image

Adds an image to the drawing canvas.

- `'name': 'image'` - the name key *must* have the value 'image'
- `'scale': <number>(default: 1)` - multiplier to scale the size of the image.
- `'align': <alignment string>(default: '')` - possible values, 'top', 'left', 'bottom', 'right', ''.
- `'offset': [<number>,<number>]` - array of x, y offsets (default value[0, 0]).
- `'src': <path to image file>` - the path to the image file to insert.

```
{'name': 'image', 'align': 'bottom', 'src': '/static/image.png'}
```

Grading Library Application Programming Interface (API)

The SketchResponse API was designed to name grading functions as clearly as possible. We recognize that our primary users are not necessarily going to be professional programmers and so detailed, readable, explicit library function names are prioritized.

As you explore the API, you will find that almost all of the functionality provided allows for the user to customize thresholds to determine correctness of given input. There are default thresholds provided for all of the functionality of this api, which has been determined through experimentation and actual usage data from students in MIT and edX courses.

There are three modules that you may interact with in the Grader Library API. The majority of the grading functions are accessed through the GradeableFunction module. Functionality to do with evaluating asymptotes is found in the Asymptote module. A handful of grading functions either take Point objects as arguments, or return them as results so a brief description of it is included here as well.

The Grader Library API is also hosted separately with a searchable index here. Direct links to the individual modules are below.

- Asymptote
- GradeableFunction
- Point

Offline SketchResponse Documents

For convenience, PDF copies of both this documentation and the API documentation can be found at the links below.

- SketchResponse_v1.0.pdf
- SketchResponseAPI_v1.0.pdf