# Assignment III: Graphing Calculator

## Objective

You will enhance your Calculator to create a graph of what the user has entered into their calculator which can be zoomed in on and panned around. Your app will now work not only on iPhones, but on iPads as well.

## Materials

- You will need to have successfully completed Assignment 2. This assignment builds on that. You can try to modify your existing program or create a new project (and reuse the classes you wrote by dragging them into the new project). In any case, be sure to save a copy of last week's work before you start.
- This AxesDrawer class will likely be very useful!

## Required Tasks

1. You must begin this assignment with your Assignment 2 code, not with any in-class demo code that has been posted. Learning to create new MVCs and segues requires experiencing it, not copy/pasting it or editing an existing storyboard that already has segues in it.
2. Rename the `ViewController` class you've been working on in Assignments 1 and 2 to be `CalculatorViewController`.
3. Create an entirely new MVC which graphs an arbitrary `x` vs. `y` function. It can have any public API you want (with the caveats below). Choose your Model carefully.
4. Your new graphing MVC must be a completely generic, reusable component. It cannot have any mention whatsoever of anything Calculator-related in any part of its implementation (not in its Model, not in its Controller and not in its View).
5. As part of your implementation of this new MVC, you are also required to write a *generic, reusable* `x` vs. `y` graphing subclass of `UIView`.
6. The graphing view must not own (i.e. store) the data it is graphing, even temporarily. It must ask for the data as it needs it, point by point. Your graphing view graphs an `x` vs. `y` function, it does not graph an array of points, so don't pass it an array of points.
7. Add a new button to your calculator's user-interface which segues to your new MVC and graphs what is in the `CalculatorBrain` at the time the button was touched using the memory location `M` as the independent variable. For example, if the `CalculatorBrain` contains `sin(M)`, you'd draw a sine wave. Subsequent input to the Calculator must have *no effect* on what the graphing MVC is showing (until the graphing button is touched again). Ignore user attempts to graph if the result in the `CalculatorBrain` is pending (e.g. there is a pending binary operation) at the time.
8. On iPad and in landscape on iPhone 6+ and 7+ devices, the graph must be (or be able to be) on screen at the same time as your existing Calculator's user-interface (i.e. in a split view). On other iPhones the graph should "push" onto the screen via a navigation controller.
9. Anytime a graph is on screen, a description of what it is being drawn should also be shown on screen somewhere sensible, e.g., if `sin(M)` is what is being graphed, then the string "`sin(M)`" should be on screen somewhere.
10. Your graphing view must be `@IBDesignable` and its scale must be `@IBInspectable`. The graphing view's axes should appear in the storyboard at the inspected scale.
11. Your graphing view must support the following three gestures:
    a. Pinching (zooms the entire graph, including the axes, in or out on the graph)
    b. Panning (moves the entire graph, including the axes, to follow the touch around)
    c. Double-tapping (moves the origin of the graph to the point of the double tap)

12. You cannot add any new public API to your `CalculatorBrain` as part of this assignment and you still cannot use the Swift types `Any` or `AnyObject` anywhere in your solution (except as part of a solution to Extra Credit 6).

## Hints

1. Forgetting to set the class of a `UIViewController` or a custom `UIView` in the Identity Inspector in Xcode is a common error. You'll need to do this when you rename `ViewController` to `CalculatorViewController` and for both the new `UIViewController` and the new `UIView` that you are creating in this assignment. Also, if you rename a `class` or `struct`, don't forget to rename the file that contains it (if appropriate).
2. To make the drawing of the graph much easier, a class which can draw a graph's axes in the current drawing context is provided (`AxesDrawer`). Notice that this class's drawing method (`drawAxes`) takes the `bounds` to draw in and two other arguments: `origin` and `pointsPerUnit` (this last is essentially the "scale" of the graph). You will very likely want to mimic this (i.e. having `var`s for origin and scale) in your generic graphing view.
3. Your `CalculatorBrain` code should not need to be touched for this assignment.
4. By now you should solidly understand MVC. Be clear in your mind what your Model is for each of your two MVCs in this assignment. Make sure you do not violate the rules of MVC (by, for example, having a your View talk to your Model directly).
5. Remember that the Model should be the UI-independent heart of what it is your MVC does or shows. A great example is the `CalculatorBrain`, which is obviously the heart of an MVC which is a calculator. So when you are thinking about what the Model for your new graphing view controller is, ask yourself, "what (fundamentally) does this MVC show?" Then just pick something which simply represents that.
6. A Model does not have to be a `class` or `struct` that you author (like `CalculatorBrain`). It can be **any type whatsoever**.
7. Think carefully about what code goes in your new graphing `UIView` versus what goes in your new graphing MVC's Controller. In general, you probably want the `UIView` to be as powerful as possible on its own.
8. This assignment is being made available before the lecture that demonstrates how to have multiple MVCs in your application is given. If you want to get started on this assignment before that lecture (which is recommended!) think about creating an entirely new application that has only one MVC: your new graphing MVC. Just pick some convenient function for it to draw for development purposes (`cos(x)` for example). Then, after the next lecture, when you've learned how to make multiple MVCs, you can just drop your reusable graphing MVC (including it's reusable graphing `UIView`) into your Assignment 3 Calculator application.
9. When you drag and drop your graphing view into the scene of your new MVC, you can use Reset to Suggested Constraints (in the menu brought up by the button in the lower right corner of InterfaceBuilder in Xcode) to set its "pin to the edges" constraints as long as you have used the dashed blue lines to put your graphing view in

the right spot in your new MVC's scene. Then you can double check what that did in the Size Inspector in the Utilities pane. If you mess up doing any of these constraints operations, remember that Xcode has undo.

10. It is quite possible to do this assignment with no more autolayout than is described above.
11. The `UIViewController` subclass for your new graphing MVC and the generic graphing `UIView` subclass are the only new classes you should have to write from scratch for this assignment. If you think you need to be writing other classes, you might be overdoing it.
12. Don't freak out when you drag out a `UISplitViewController` and it brings along all kinds of other view controllers along with it. It's just Xcode trying to be helpful. You can safely delete those in your storyboard and use ctrl-drag to wire up **your** MVC scenes (inside navigation controllers) in their places.
13. It'd be nice for the origin of your graph to default to the center of the `UIView`. But be careful where/when you calculate this because your `UIView`'s `bounds` are not set until it is laid out for the device it is on. You can be certain your `bounds` are set in your `draw(CGRect)` of course, but be careful not to **re**-set the origin if it's already been set by the user.
13. Your graphing MVC probably wants do something sensible when graphing discontinuous functions (for example. it should only try to draw lines to or from points whose `y` value `.isNormal` or `.isZero`). To make things simpler on this front, it's okay if your graphing view improperly graphs a function that rapidly goes through a huge swing in value across a single pixel by drawing an almost vertical line between those two points even if the function is actually probably discontinuous there (e.g. `tan(x)`). It would be cool to try to detect this case within some tolerance, though, and not draw that vertical line (up to you).
14. Don't overcomplicate your `draw(CGRect)`. Simply iterate over every pixel (not point) across the width of your view (or, better, across the area you are being asked to draw in per the argument to `draw(CGRect)`) and draw a line to (or just "move to" if the last datapoint was not valid) the next datapoint you get (if it is valid).
15. The coordinate system you are drawing in inside your `draw(CGRect)` is not the same as the coordinates your data is in (because, for example, your drawing coordinates have the origin in the upper left, but the data's origin is probably somewhere else in the view; not to mention your graph might be scaled). Be clear in your mind as you write your code **which of these two coordinate systems** a given `var` or argument to a function is (and should be) in.
16. The `AxesDrawer` knows how to draw on pixel (not point) boundaries (like your `draw(CGRect)` should), but only if you tell it the `contentScaleFactor` of the drawing context you are drawing into.

17. Don't forget to use property observing (`didSet`) to cause your view to note that it needs to redisplay itself when a property that affects how it looks gets changed.
18. Make sure you set your `UIViewContentMode` properly (this can be done in the storyboard).
19. Your gestures will probably be *handled* by the graphing view, but will probably want to be "turned on" by your Controller. For this reason, the methods that handle the gestures shouldn't be `private` in your graphing view.
20. This assignment will probably require a bit more code than your first two assignments did, but it still can be done in well under 100 lines of code.

## Things to Learn

Here is a partial list of concepts this assignment is intended to let you gain practice with or otherwise demonstrate your knowledge of.

1. Understanding MVC
2. Value Semantics
3. Creating a new subclass of `UIViewController`
4. Universal Application (i.e. different UIs on iPad and iPhone in the same application)
5. Split View Controller
6. Navigation Controller
7. Segues
8. Subclassing `UIView`
9. `UIViewContentMode.redraw`
10. Drawing with `UIBezierPath` and/or Core Graphics
11. `CGFloat/CGPoint/CGSize/CGRect`
12. Gestures
13. `contentScaleFactor` (pixels vs. points)

## Evaluation

In all of the assignments this quarter, writing quality code that builds without warnings or errors, and then testing the resulting application and iterating until it functions properly is the goal.

Here are the most common reasons assignments are marked down:

- Project does not build.
- Project does not build without warnings.
- One or more items in the Required Tasks section was not satisfied.
- A fundamental concept was not understood.
- Code is visually sloppy and hard to read (e.g. indentation is not consistent, etc.).
- Your solution is difficult (or impossible) for someone reading the code to understand due to lack of comments, poor variable/method names, poor solution structure, long methods, etc.
- UI is a mess. Things should be lined up and appropriately spaced to "look nice."
- Public and private API is not properly delineated.
- The boundaries of MVC are violated.
- The Swift API Design Guidelines were not properly adhered to.
- Your application contains no memory cycles (i.e. an instance of a reference type points, directly or indirectly, to another instance of a reference type that points back to the first instance).

Often students ask "how much commenting of my code do I need to do?" The answer is that your code must be easily and completely understandable by anyone reading it. You can assume that the reader knows the SDK, but should not assume that they already know the (or a) solution to the problem.

## Extra Credit

We try to make Extra Credit be opportunities to expand on what you've learned this week. Attempting at least some of these each week is highly recommended to get the most out of this course.

1. Have your graphing button in your main Calculator scene reflect whether or not it is currently possible to graph what has been entered so far (i.e. whether there is a pending result or not). You could just disable it, but maybe a different graphic or something? This is a very easy task, so don't expect a lot of extra credit for it!
2. Preserve origin and scale between launchings of the application. Where should this be done to best respect MVC, do you think? There's no "right answer" to this one. It's subtle.
3. Upon rotation (or any `bounds` change), maintain the origin of your graph with respect to the center of your graphing view rather than with respect to the upper left corner.
4. Figure out how to use Instruments to analyze the performance of panning and pinching in your graphing view. What makes dragging the graph around so sluggish? Explain in comments in your code what you found and what you might do about it.
5. Use the information you found above to improve panning performance. Do NOT turn your code into a mess to do this. Your solution should be simple and elegant. There is a strong temptation when optimizing to sacrifice readability or to violate MVC boundaries, but you are NOT allowed to do that for this Extra Credit!
6. When your application first launches, have it show the last graph it was showing (rather than coming up blank). You could reset the Calculator MVC upon relaunch to the last state it was in as well (which may or may not be the same thing as what the graph was showing). Be careful not to violate MVC in your solution, though (each MVC its its own independent world). The simplest persistence mechanism in iOS is `UserDefaults`, but only Property Lists can be put there, so you will have to figure out how to get the function you are graphing into a Property List-compatible form. Because of that, you *are* allowed use the Swift types `Any` or `AnyObject` to implement this Extra Credit item.