NATuG Technical Manual

Wolf S. Mermelstein and William B. Sherman

July 24, 2023

Contents

1 Navigating the Codebase

1.1 Comments

NATuG's codebase implements Python's two main types of comments: inline comments and docstrings.

Docstrings In Python docstrings are a type of specially formatted multi-line string that comes after class and function definitions to provide additional information about the schema of the structure. NATuG implements Google formatted docstrings for **ALL** functions and classes. All docstrings should be fairly straightforward, and don't need to go too deeply into depth about *how* things work; rather, they should focus more on how to use things and what things do.

Comments NATuG utilizes inline comments less so than it does docstrings, and wherever appropriate. Comments should be used to explain how things work or what things do. Comments should **not** be used for book-keeping, such as recording changes, insertions, or TO-DOs.

2 NATuG's Modules

To properly understand how NATuG functions, it is important to understand what parts comprise NATuG, and how they combine to form the overall program.

In the parent directory of NATuG there exist a few different folders, many of which are modules (a Python module is a folder with a `__init__.py` file in it), which each serve an important, distinct purpose. Within some of these parent modules live submodules, which contain files that relate to the folder that they are in, and so forth.

The following subsections provide descriptions of all the main folders of NATuG, including modules and non-module folders, which should help in subsequent explanation of how they all piece together.

2.1 Constants

The constants module is the most simple module to understand. In this module lies program-wide constant variables. For example, the number `0` represents `UP`-ness. In order to prevent code from becoming overly opaque, however, we `import DOWN from constants.directions` and then use the term `DOWN` in the code. It's just an alias.

2.2 Runner

This is heart of the program. `runner.py` within the runner module defines a class that manages the overall, entire program state. When NATuG is run a single `Runner` object is instantiated, and no others should ever be created. This instance has a reference in the `launcher.py` main file, but that reference shouldn't ever actually get used, except for the launcher to call its `setup()` method, which prepares it to then have its `boot()` function called by the launcher too. References to the runner are available elsewhere in the application, but they are distributed to components by the runner itself.

The runner class can be thought of as a container that holds eclectic instances of objects that are needed for the app to work. It contains things like the `QApplication`, which is QT's ultimate parent widget, and a flag that indicates whether the program has finished booting. Additionally, it contains a `Filehandler` object and possesses a `.snapshot()`, `.save()`, and similar methods, which lets you save the entire state of the program, which makes sense given that it has access to all parts of the current program state.

2.2.1 Managers

The most important part of the runner is its `.managers` attribute, which contain compartmentalized sub-containers for storing **instances** of the various data structures that represent nanotube (and configuration) data. The definitions of the managers can be found within the `runner/managers` folder, and each has a fairly similar schema. Most of the managers when initialized call an internal `setup()` function, which loads

the last program's saved state or some default from a file to the `.current` attribute of the manager, and then registers a `dump()` to be run at program exit using the `atexit` python library.

The relationship between runners and managers is a bit complex, so let's take a look at an example. The `ui` module, which will be discussed later, has a `config` submodule, which has various tabs, like the `domains` tab. The domains tab lets the user change properties of the domains of the nanotube, such as the internal angles of given domains in a given subunit. The table that lets you set the counts, angles, and other properties of domains. From the user's perspective, their changes persist so long as the program state persists. If they switch tabs and return to the domains tab in the config panel, they expect their settings to remain unchanged. When they update a given domain's interior angle, they expect the plots to update accordingly, and the new angle to take place of the old one in the current program state.

But of course, the table that *displays* the domain setting that they are planning on changing is just a customized widget that inherits PyQt `QWidget` and contains a `QTabWidget` that's filled with `QSpinBox` widgets. It's a UI Widget, and while the various subwidgets have states storing their own current values independently, the data is only meaningful when grouped together in a Python `object` designed to properly represent the information that is being displayed. To this end, the domain angle editing table, for the sake of this example, has a `fetch_domains()` and `dump_domains()` method to allow for dumping `Domains` instances into and out of the table. It automatically looks at its children widgets and extracts or sets the current value of each, so as to create the new `Domains` object representative of the UI elements' states, or dump an existing one. Those methods are abstractions designed to access states, but where does the program's current `Domains` object actually live?

That's exactly where the idea of runner and managers comes in. They are designed to help organize and manage the program's state. The **overall config panel** that contains all the various configuration tabs, like the nucleic acid profile tab and the domains tab, has a reference to the runner and a reference to its child tables. Its child tables fire events that tell the panel when an update has been made, where then the panel can call it's child's `fetch_domains()` method, and store the instance returned in `self.runner.managers.domains.current` (and the same process is done by other managers too, simply by swapping out the "domains" for, say, "nucleic_acid_profile"). More discussion of how events work with NATuG will be provided later, but the general idea is that a widget can have fire-able events, and other widgets can subscribe to those events to do things when they are fired, such as is the case in this example.

Another interesting implication of the manager system that NATuG utilizes is that it makes it possible in the future to allow for instances of the overall program to run simultaneously. This ability may be useful if editing multiple nanotubes side-by-side is desirable, or perhaps for superposition of structures. Modification would need to be made to externalize application-wide attributes of the runner, such as the `QApplication` instance that it holds, but attributes that hold the program state such as the managers could easily be ported to a reusable class, and UI components could be built out to provide various views of various states by allowing for the input of lists of runners, or perhaps tabs that contain copies of the existing widgets. None of this is implemented yet, but it should hopefully provide some insight into why the system currently is the way it is.

Words of Caution Giving UI elements access to the program's runner instance by passing it in upon initialization should be avoided whenever possible. Children should "speak" to their parent widgets to tell them to update the program's state, but access to the runners should be limited to only outermost UI elements like the config panel widget itself, for a few reasons. Firstly, the runner is powerful. Mutating attributes of the runner, such as its managers, can have extremely unexpected and program wide implications. By restricting access to who can *access* it, it's much easier to debug the program. Secondly, requiring a runner instance to initialize a child component is also problematic because it makes it harder to allow for reuse of that component. For instance, the `TripleSpinbox` widget is used to allow for the storing and dumping of three integer inputs, for determining how many nucleosides to generate data for. If that element required a runner instance so that it could fetch the current count configuration from the `DomainsManager`, its parent would then also require a runner instance for the sole purpose of *drilling* the reference down to that `TripleSpinbox`. Instead, whenever possible, we provide the outermost parent the runner instance, which utilizes the state to assign data to its inner children.

2.3 Saves

The "saves" folder is **not** a Python module, but it **does** still take on a specific shape that is important to understand. The various "managers" (discussed in 2.2.1) rely on files in this folder for loading up and shutting down properly, since NATuG is designed to provide a degree of cross-execution state persistence.

`launcher.py`, the script to boot NATuG, automatically will create missing folders to build up a properly-shaped saves folder if necessary. Additionally, the **DEBUG** flag in the launcher script will automatically delete all the restored-state files if it is set to `True`.

Within the saves folder, there must be the following subfolders:

- **Domains:** Within this folder is a *presets* subfolder, which contains all the preset domain configurations that NATuG possesses, along with a *restored.csv* file, which is loaded in as the current program state when the program next boots. If *restored.csv* cannot be found, *presets/tengon.csv* is used instead.
- **nucleic_acid:** This folder is a container for all the nucleic acid profiles that the user can choose from in the nucleic acid profile manager in the config tab of NATuG. NATuG will default to *restored.json* upon booting, but if it cannot be found *MFD_B-DNA.json* is loaded instead.
- **Snapshots:** This folder is used to store complete program save files. Each file holds an entire program state, and they are numbered where higher numbers represent more recent program states. The snapshots in this folder are available for loading by the user in the snapshots tab of the config panel, or via simple control+z and control+shift+z shortcuts.
- **strands:** This folder currently doesn't store much. Right now, it has a subfolder called *presets* that has preset viral sequences that can be loaded into the sequence property of strands.

2.4 Structures

The structures module is perhaps the most critical part of NATuG. This module contains the actual definitions of all the various different pieces of information needed to represent nanotubes, such as that of the Domains or the Nucleic Acid Profile. Additionally, logic pertinent to specific structures, such as Domains, lives within the definitions of the classes it is pertinent to as class methods. This means that program-critical functions like the **conjoin** function to create junctions, or the **link** function to create linkages, live in files in this module.

Before exploring the specific parts of this module, one must understand how the module is structured. An understanding of the deliberate structure of the module is important not only so that specific classes can be easily located, but also to assure consistency when further structures are inevitably implemented into NATuG.

The structures module is broken down into categories of structures, including one for domains, helices, strands, points, profiles, and more. Each category contains various class definitions for structures pertinent to the category, but broken down into subfiles of the module with the name of the class that they contain. Higher level structures may contain other lower level structures from the same module—for instance, the `Domains` class contains `Subunit` objects, which contain `Domain` objects. Likewise, some structures inherit from other structures, such as is the case for `Nucleoside` and `NEMid` objects, which inherit from `Point`.

To import a given structure, you could use an absolute import statement along the lines of `import structures.domains.domain.Domain`, or `from structures.domains.domain import Domain`, but that's annoying to type out and somewhat redundant. Though Python, unlike javascript, doesn't have default-imports for modules, a similar idea is used by placing imports into the `__init__` folders of the modules. For example, in `structures/domains/__init__.py`, there exist relative imports like `from .domain import Domain`. That statement adds `Domain` to the namespace of the domains module itself so that it can be accessed via `structures.domains.Domain`, even though the definition lives in `structures.domains.domain.Domain`—in other words, it's an alias, and you should always use them and add them for structures that are intended to be accessed outside of the given structure submodule itself. `Domains` require access to the `Strand` object for instance, since they are able to generate strands, and so `structures/strands/__init__.py` provides an alias that domains uses for the import.

2.4.1 Profiles

The profiles structure category includes the most important type of structures, profiles of constants. They can be thought of as simple key value tables with specific slots of input, and also various helper methods to compute additional constants from their data. It is the case that, perhaps unintuitively, beyond that description the interfaces of the profiles (of which there currently are two) are not much alike.

Note: Profiles many include various properties defined using the `@property` decorator. This decorator defines a function that is computationally inexpensive, and allows you to access the given property through simple dot-syntax, as if the function underneath the `@property` tag were a property assigned directly to *self*. The reason that the values that the properties return aren't assigned to the instance automatically at initialization is because profiles are mutable, and the constants that they store can change. One easy optimization here could be to properly cache the values of the properties and recompute them whenever any of the values of the profile changes, but this has not yet been implemented.

Nucleic Acid Profile The nucleic acid profile contains the constants of the specific nucleic acid that the user is working with, and instances of `NucleicAcidProfile` are passed down to just about every other structure of NATuG. To change a property of a nucleic acid profile instance, you can simply overwrite it with a new value. Some values are properties that are automatically computed based on other properties, and those properties cannot be overwritten.

The Nucleic Acid Profile provides a `to_file()` and `from_file()` method, which are simply wrappers on `dataclasses.asdict()` and `io` to dump the user set key values of the nucleic acid profiles to a `.json` file. For this application, `.json` makes sense since it is a very simple filetype for storing key-value pairs.

Additionally, and importantly to understand, the Nucleic Acid Profile also has a `.update()` property, which lets you update the **entire** Nucleic Acid Profile in-place. Recall from earlier how the current instance of the various structures lives within `runner.managers.<manager-name>`. The Nucleic Acid Profile is no different. When you use `from_file()`, or other methods elsewhere in NATuG that return `NucleicAcidProfile` objects, these objects are brand new. The `.update()` method lets you **copy** over the properties of a different `NucleicAcidProfile` over to a specific instance, so that wherever the profile is referenced the reference is up-to-date.

Perhaps an example would be of help to better understand the application of this class method. If somewhere in the user interface for Nucleic Acid Profile configuration a property of the Nucleic Acid Profile is changed, and the `fetch_nucleic_acid_profile()` method of the nucleic acid profile configuration tab is called to fetch the new `NucleicAcidProfile`, we end up with a brand new object with the current profile data. The parent widget that fetched the new profile has access to the `runner`, so it may be tempting to try to mutate the current `NucleicAcidProfile` instance by writing `self.runner.managers.nucleic_acid_profile.current = new_nucleic_acid_profile`. However, this is problematic since it will actually overwrite `nucleic_acid_profile.current` with the new profile, while all references all throughout NATuG will continue to point to the old profile. The old profile may be out of sync, and it won't get garbage collected, which is extremely problematic.

2.4.2 Domains

The domains category of structure is one of the more complex structure types. The `Domains` object is the ultimate parent object, which contains information about all the domains in the nanostructure, along with some metadata, but in order to do so it has multiple layers of children. Below lies a breakdown of the various components of `Domains`, from the bottom up.

Domain A `Domain` object is the child of a specific `Subunit` of domains. It stores data about a specific domain, such as its interior angle, helical switches, and generation counts (which is stored in a `GenerationCount` object, defined in the same file as `Domain`). It also has some aliases to properties of its parents, which currently are implemented but not used.

Subunit A `Subunit` object is a container for a group of `Domains` that is repeated a certain number of times for a rotationally symmetric structure, or not at all for a structure with a symmetry factor of 1. It

has methods similar to that of a list, such as `.remove()` and `.append()`, since it is for most purposes a list of `Domains`.

A **template** subunit is a subunit that has its `template` attribute set to `True`, as to indicate that it is the template for all copies that are not templates. The `.domains()` method of the overall parent `Domains` class calls the `.subunits()` method, which generates the subunits, for which only one is a template, and that is the only real case where the template flag comes into play.

Domains A `Domains` object is the main container for all things domain related. It contains a template subunit and symmetry factor, and provides helper functions such as `.count` to get the total number of domains with symmetry taken into account, or `.domains()` to get a list of all domains across all subunits.

It is important to have a good understanding of how references in Python work before working with `Domains` objects, since it is very easy to produce unexpected results when generating and mutating domains. When you call `.domains()`, for instance, all the `Domain` objects that are returned are brand new objects, and are not attached to anything. Changing a property of one of the returned domains will not change the overall `Domains` object. To mutate a specific domain in the `Domains` container, you'll want to mutate the domain in the template subunit directly, by writing something along the lines of `main_domains_object.subunit[1].theta_m = 8`, or swapping that `Domain` reference out for a different one.

2.4.3 Points

The points category of structure is probably the most intuitive type of structure. There's three main types of points, which are all descendants of the `Point` object: `NEMid`, `Nucleoside`, and `Nick`. There's also a `Pseudo` point, which is meant to be a placeholder point, but it is currently not used.

Points are datastructures, and, though they provide some useful methods like getting its helical_index,

Point The parent class of the specific types of points is the `Point` class. The point stores various pieces of data, such as its coordinates, angle, helical index, and more. These data are set automatically by the `.compute()` method of `DoubleHelices`, but can be overwritten in case the user wants to modify them.

Points also provide some helper methods, like `.overlaps()` to see if the point overlaps another point, or `.is_endpoint()` to see if the point is an endpoint of its strand. If a specific condition is checked often of a point, making it a method of point may not be a bad idea.

Styling `Point` objects also contain styles, which dictate the specific styles of the point. The styles of a specific point are ephemeral, however, since the `.style()` method of `Strands` will automatically overwrite the styles of all the points in all the strands. Once the points have had their styles computed, before everything gets restyled (as would happen when junctions were created, the user clicked the restyle button, or the user changes the color of a specific strand, for example), the state of the point's style can be changed to alter its appearance in a consistent way. To change the state of a point, call `Point.change_state()` with a new state. Currently "default," "highlighted," and "selected," are supported.

2.4.4 Helices

The `helices` category of structure represents all the helices of the nanostructure. It is important to grasp how helices are different from strands both fundamentally and in implementation. When domain settings are modified and the strands and helices are generated, the data that the strands and helices store is roughly the same. However, helices represent each individual double helix within its own domain, not taking into account strand exchanges.

Strands are generated based off of `Helix`es, and can be further mutated by the user. Technically `Helix`es are also sometimes mutated by the user, but this really only happens when nicks are created. `Helix` objects, however, are designed to be easy and computationally cheap to generate, but, as a tradeoff, are harder to mutate and cannot be reshaped (i.e. you can't change the length of a helix after it has been generated, since they implement fixed-sized arrays). Strands, on the other hand, are designed to store highly mutable data. In the early days of NATuG `Strand`s took on a linked list data structure, but now they implement a dynamically resized array, since they allow for faster lookups, which turns out to be more important than

quick additions. Strands store chains of point and container objects, such as `Nucleoside`s, `NEMid`s, and `Linkage`s (which are containers of `Nucleosides`). These two datatypes are completely different, so it is important not to mix them up, even though they contain references to the same `Point` objects.

Helix The `Helix` class is a child of a `DoubleHelix` that on its own is merely a highly customized container, designed to be loaded with data by its grandparent `DoubleHelices` object. Once loaded with data, the Helix provides some extremely useful methods that are used (also by its grantparent `DoubleHelices` object) to generate a `Strands` container object.

The most important attribute of the `Helix` is the `.data` property, which holds a few fixed sized arrays of data, such as `x_coords` and `angles`. It also has a `Point`s array, which, when the `Helix`'s `.points()` method is called fills up with the `NEMid` and `Nucleoside` objects that are computed based on its numerical data.

Double Helix The `DoubleHelix` class is, at its essence, a container for two `Helix` objects. Internally it stores its two helices by direction—`UP` and `DOWN`—in a two-element array of `Helix`es, but its interface provides many different ways to access either, since each helix of a given `DoubleHelix` has many different aliases. For instance, you can access the "right" helix with `DoubleHelices.right_helix`, or the zeroed helix with `DoubleHelices.zeroed_helix`. Extended explanations of what the aliases represent can be found in the `DoubleHelices`'s docstring.

DoubleHelices The `DoubleHelices` object is an extremely important higher-level element that contains **all** the `Helix` objects, neatly packaged into `DoubleHelix` bundles. Though it requires a list of `DoubleHelix` instances to initialize, it offers a useful `from_domains()` classmethod to create an instance from a `Domains` instance instead, which automatically takes care of creating the `DoubleHelix`es.

Perhaps the most important functions of all of NATuG live inside the definition of a `DoubleHelices`. Its `.compute()` method computes the coordinate and angle data for all the child `Helix`es, and its `.strands()` method implements the `Helix`es' `.strand()` methods to create an overall `Strands` object.

2.4.5 Strands

The strands category of structure is a much more flexible datastructure that stores arrays of points in a form that allows for easy merges, splits, removals, appendages, and the like. Strands also control the styling of points and outlines in the side view plot, and are what provide the ability to create junctions, linkages, and other actions that alter the current strands.

Strands The `Strands` object is an unordered container for many `Strand` objects. The container provides various methods to mutate strands, including the `.conjoin()` function to create junctions and the `.nick()` function to create nicks. Generally there is only ever one instance of `Strands` alive at a given time, and all `Strand` objects belong to it.

`Strands` are also responsible for storing a list of all of the current `Nick`s in play, which is useful to know, since to otherwise obtain the nicks you'd need to traverse every single strand one at a time.

Strand A `Strand` object contains an array under its `.items` attribute of ordered points. These points are stored in a container called a `StrandItems`, which provides some useful methods for unpacking the points like `by_type()` or `unpacked()` (useful for unpacking items in subcontainers, like linkages).

Styling Strands also have **styles** stored under their `.styles` attribute, which store various graphical style information like the thickness of the strand, the color of the strand, and whether or not the strand is highlighted. The properties within the styles of a strand are read by the `Plotter` classes later on to visually plot the `Strands`.

What makes the styling of strands slightly confusing is that these specific style properties take the form of `StrandStyle` objects, which hold the value of the style and a "automatic" flag. This is because by default strands are styled in a specific way, and the values are set for the strands by the parent `Strands` container, but if automatic is set to false the parent `Strands` container does not overwrite the styles.

2.5 Tools

The tools folder is simply a place to put all miscellaneous Python scripts that are not directly hooked into NATuG, but have logic that may be of use down the road.

2.6 UI

The UI module contains all the widgets and window definitions needed to provide a graphical user interface for interaction and preview of the current instances (which live in the runner's managers) of the various structures of NATuG. The UI module is broken into various different parts, and it is easiest to understand how they piece together by taking a top-down approach.

Most of the UI submodules that take the form of panels in the main window have a `Panel` class that groups together their various counterparts. For example, the config panel has many tabs, and tabs have different parts like tables and input boxes. Each tab has a `Panel`, and then there's another `Panel` that lives in the `config` submodule itself that imports the panels from all the subsubmodules and displays them as a list of tabs. It may be confusing that all the panels are named `Panel`, but they are located within sensical subfolders, and can be imported with Python's *as* syntax if needed for further clarity (i.e. `from ui.config import Panel as ConfigPanel`).

2.6.1 Window

The `Window` submodule is the class with the definition for the main window of NATuG. It has a `setup()` method which calls various private methods that configure various parts of the main window. The runner automatically handles this setup phase before displaying the window.

In PyQt the main window of the application is generally a `QMainWindow` descendant, and this is the case for NATuG too. PyQt has various useful slots for UI components in this class, such as an area for a toolbar, a status bar, dock widgets, and the like. The most important widget in the main window is the "central widget," which in the case of NATuG is a `QSlider` widget that contains both the current top and side view plots.

The window imports from various UI submodules to construct one neatly integrated main window, but most of the actual logic for the submodules lives in their own respective folders. Since the main window is a property of the runner, anything with access to the runner has access to any property of the main window, and can do things like refresh the plots or alter the current config tab, for instance.

2.6.2 Config

The config submodule contains all the relevant widgets for the config panel, which is the tabbed widget on the right side of NATuG. The config panel has a few different tabs, which have their own panel definitions in `ui/config/tabs/[tabName]`. Config imports the various tabs and loads them into the a `QTabWidget`.

Updating plots Each of the submodules of config have access to an instance of `runner`, and are able to mutate the current program state. For all the submodules, when mutating the program state they use a conditional `if RefreshConfirmer.run():`, which will automatically prompt the user to save their changes or cancel if changes are to be lost if things refresh, or returns `True` if there is no risk. Once they have updated the program state, they send an `updated` signal which the config panel is subscribed to, and that triggers the config panel's `_on_tab_updated()` method, which checks if auto-updating is enabled and then conditionally updates the plots.

The file `dockable.py` within the config module is simply a wrapper for the config panel that allows it to be docked to the main window. No important logic is contained there, and it is simply a means to allow the config panel to detach from the main window.

2.6.3 Dialogs

The dialogs submodule contains definitions for various complex dialogs that fetch information from users. For simple warnings, `utils.warning()` should be used, and for simple confirmations, `utils.confirm()` should be used.

Generally dialogs are implemented by passing in an object of some sort during initialization and then subscribing to updates or the close event. Take, for example, the `StrandConfig` dialog. The side view plot automatically hooks a function onto the strand-click event such that a `StrandConfig` is displayed whenever a strand is clicked in the side view plot. The `Strand` object that is clicked is passed in on initialization, and the user can use the dialog to change various settings of the strand like its color and thickness, along with viewing various properties. The plot knows to refresh after updates have been made/the panel is closed because before even displaying the `StrandConfig` dialog the side view plot calls `dialog.updated.connect(self.refresh)` to automatically trigger a refresh whenever the `update` event of the dialog is `.emit()`ted.

This is the case for most dialogs, even for ones that seem like they have no effect on program state. For example, the informers do not appear to change anything, and their main purpose is to display information. However, they change the current style state of a given point object to `highlighted`, which requires a side view plot refresh.

2.6.4 Menubar

The menu bar is the top area of the screen with submenus like "File," "View," and "Edit." At the moment the logic for the options in these menus is very simple. Each menu in the menu bar has its own definition, and they are linked together in `menubar.py`.

2.6.5 Plotters

The plotters are the widgets responsible for plotting strand data. They inherit from `Plotter`, which provides an `export()` method, which inherits from `pyqtgraph.PlotWidget`.

There are two plotters, a `TopViewPlotter` and `SideViewPlotter`. Both plotters have a `.plot_data` attribute that stores a `PlotData` item, defined separately in the respective plotter's file. The idea behind the plot data object is to store information about the currently plotted data, in case the program state changes and the data that is currently plotted falls out of sync. This is not really taken advantage of as-is, but the plot data does stay up to date with what is actually plotted, which may be useful down the line.

Both plotters have plot methods, but these methods should not be called directly. The data passed in (e.g. Strands, Domains) will automatically be plotted at initialization, and to update the plot to match data that has changed the respective plotter's `.refresh()` method should be called, which will automatically clean up the plot and plot new data. Likewise, both plotters have `._prettify()` methods to add axis labels, gridlines, and the like, but that will also automatically be called during refreshing and initialization. The plotters also both provide various useful events, defined at the top of their classes, which are documented in the docstring of the respective plotter. These events are subscribed to in the Panels submodule of UI, which will be discussed subsequently.

The plotters can be customized through parameters set at initialization (that can also be changed later), like the `TopViewPlotter`'s `.rotation` attribute/argument, which controls how rotated the plot is, or, for the side view plotter specifically, through alteration of its `.modifiers`; the side view plot's modifiers change the sizing of various plot features by multipliers stored in a `PlotModifiers` instance.

2.6.6 Panels

The panels submodule contains wrappers for the side and top view plotters. They store under a `.plot()` attribute their `TopViewPlotter` or `SideViewPlotter`, and they act as display frames and event managers. They are both `QGroupBox` descendants, which is a type of PyQt class that allows for adding a nice looking label above items. They also both possess methods that hook into the various events of their respective plotter. For example, the side view panel has a `_on_linkage_clicked()` method that is hooked to the `SideViewPlotter.linkage_clicked` event, and defines how to handle the event (in this example, it makes a `LinkageConfig` dialog).

Though these widgets are frames that store a plot internally, they do do additional logic upon refreshes, like reassigning references, so while you could refresh a plot through `runner.window` instance by doing something like `runner.window.side_view_plot.plot.refresh()`, you should use `runner.window.side_view_plot.refresh()` instead. In both cases the Plotter's `refresh()` method will eventually get called.

2.6.7 Resources

The resources submodule is pretty simple in purpose: it stores various graphical resources that NATuG relies on, such as icons. `workers.py` provides helper functions, like `fetch_icon()`, but that's about it.

2.6.8 Toolbar

The toolbar is currently a list of buttons of side view plot modes. PyQt supports multiple toolbars, so down the line another toolbar may be added for top view plot modes. `actions.py` has a `Actions` class, which is a `QButtonGroup` of actions that are mutually exclusive (only one action can be chosen at a time). The various actions are defined within the `Actions` class, and inherit from `Action` some basic styles. Each action has an ID that is a constant that can be found in the constants module. `toolbar.py` assembles the actions and other basic toolbar widgets like the action repeater button into one continous toolbar to be placed at the top of the main window in `Window`.

2.6.9 Widgets

This module is meant for miscellaneous widgets that can be reused throughout the app and are not tied to specific structures or features. They should be designed with re-usability in mind. Modules here are imported throughout the application, and these widgets are fairly generic.

For example, the `TripleSpinbox` lets a user choose three integer values, and provides an API to set/get the values. It's very simple, and even though it only currently has one use, it could be implemented elsewhere, or in an entirely different project all together. If a module relies on any NATuG-specific data structure it probably shouldn't go into the `widgets` module.