

What VoidLoop ViewPoint™ Is

VoidLoop ViewPoint™ is a desktop application for visualizing and interacting with high-speed serial data from embedded systems. The desktop app receives both data and configuration from a microcontroller such as Teensy or Arduino (using our provided, open-source libraries) or other serial source, then renders that stream as live plots with multiple traces and analysis views.

In practice, VoidLoop ViewPoint™ sits somewhere between a serial plotter, a live instrumentation tool, and a lightweight analysis environment.

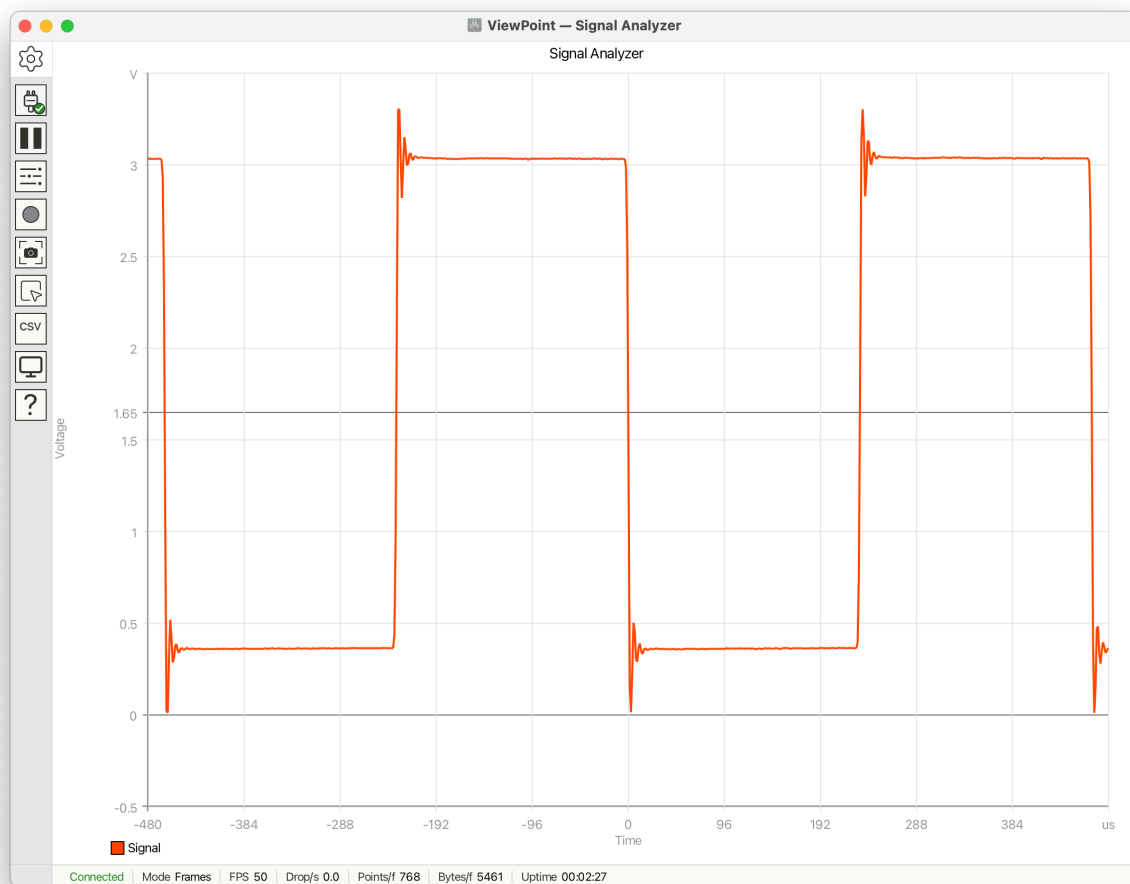


Table of Contents

What VoidLoop ViewPoint™ Is.....	1
What ViewPoint™ is and How to Use it.....	6
QuickStart	7
Upload a Sketch from the Arduino IDE	7
Open ViewPoint™ and Connect to the Teensy	8
Confirm the First Live Plot	9
Signal Input Requirements (Teensy 4.1)	10
Voltage Range Constraints	10
50 Ω Output Considerations.....	10
Desktop App and Arduino Library: Who Does What?	11
The Arduino Library Can Define	11
The Desktop App Handles	11
Common Terms	12
Plot Types	13
Update Modes.....	13
Display Modes.....	13
Using VoidLoop ViewPoint™	14
Main Window Layout.....	14
Sidebar Controls	16
Sidebar Icon Details.....	17
Main Desktop Features	18
Settings.....	18
Connection Workflow	18
Screen Capture Settings	19
CSV Capture Settings	20
Controls	21
Selecting a Plot.....	21
Axis Configuration	22
Scale	23
Auto-Scale Strategies	23
Ticks	24
Bounds and Range.....	24
Polar Properties.....	26
Frame Decay (Scatter and Polar plots)	27
Region of Interest	28
Plot Style.....	29
Display Modes.....	30
Trace Configuration	33

Changing the Trace Label	33
Changing the Trace Color	33
Derived Traces	34
Diagnostics Status Bar	35
Quick Access Settings	36
Reconnect	36
Retain App-Side Configuration	36
Analysis	37
Markers	37
Reference Lines and Bands	42
Trace Statistics	46
<i>The Arduino Library</i>	<i>47</i>
Install	47
Arduino Library Manager	47
Manual install	47
A Typical Sketch Outline	47
Concepts	48
view	48
Trace	48
Plot	48
Mode	48
PlotType	49
DisplayMode	49
Namespace Shortcut Macros	49
Legacy Mode	50
Configuration and Defines	50
Memory Profile Macros	50
Individual Override Macros	51
Common Data Shapes	52
Minimal Starting Examples	52
MCU Compatibility	53
Common Library Calls	54
Mapping Library Calls To What You See	54
Example Sketches by Use Case	55
ViewSauce	55
HelloPlotter	55
HelloScatter	55
HelloPolar	55
MultiSensorDashboard	55
SpectrumAnalyzer	56
SignalAnalyzer_A0	56
AliasingExplorer	56
WindowFunctionExplorer	56
VibrationAnalysisXY	56
WindRoseChart	56
Recipes	57

1. Labels	57
2. Axis ranges and grid spacing.....	58
3. Trace colors and the legend	59
4. Continuous vs Frames Modes.....	60
5. Custom grid colors and themes	63
6. Reference lines: trigger threshold and event marker	64
7. Oscilloscope-style centered view	66
8. Electrocardiogram-style strip chart	67
9. Large-dynamic-range sensor on a log axis	68
10. Multi-plot stacked monitor	69
11. Polar angular conventions.....	71
12. Scatter with breaks	73
13. Multiple display modes from the same frame stream	74
Quick Troubleshooting.....	76
Help and Feedback.....	78
Report an Issue from within ViewPoint.....	78
Appendix A: API Reference	79
Lifecycle	79
Plot Type, Mode, and Delay	80
Default Plot Configuration.....	80
Logarithmic Scale	81
Grid and Reference Lines	81
Display Modes.....	81
Multi-Plot	81
Trace Management	82
Messaging.....	82
State Queries	83
Advanced	83
Appendix B: Advanced Serial Command Reference.....	84
Command Format	84
Session Lifecycle	85
Common Values	85
Commands	86
[viewpoint]	86
[sketch].....	86
[cartesian].....	86
[scatter]	86
[polar]	87
[continuous].....	87
[frames]	87
[packet size].....	87
[trace labels]	88
[trace color]	88
[trace filter]	88
[add traces].....	89
[plot labels]	89
[units]	89

[axis]90

[log scale]90

[reference line]91

[grid colors]91

[display mode].....92

[time]92

[message] and [error].....92

Complete Example93

Appendix C: Library Color Reference.....94

Created by VoidLoop

Founded by Gregory Kovacs

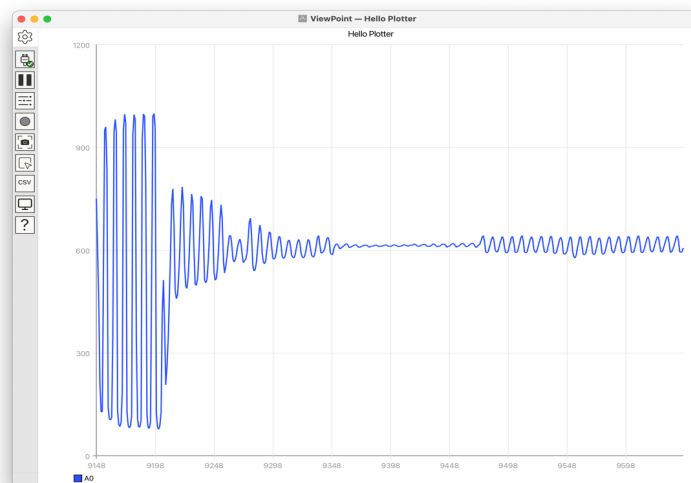
Written by Zachariah Magee

What ViewPoint™ is and How to Use it

The normal workflow looks like this:

1. A sketch on the device (microcontroller) sends commands and data over serial (USB emulated, RS-232).
2. ViewPoint™ receives those commands and uses them to configure plots, traces, labels, ranges, and modes.
3. ViewPoint™ renders the incoming data in real time.
4. The user can then further configure, inspect, tune, capture, or export what they see from the desktop UI.
5. Additional commands can be sent and interleaved with data to update configuration.

The key idea is that the sketch can define the starting structure, while the desktop app gives the user ways to analyze and interpret what's received.



Live HelloPlotter session. Working Cartesian plot with live data.

```
#include <ViewPoint.h>

void setup() {
  view.begin();
  view.setTitle("Hello Plotter");
}

void loop() {
  view.addData("A0", analogRead(A0));
  view.send();
}
```

Upload this (examples/Cartesian/HelloPlotter.ino), open the desktop app, and select the board's serial port from **Settings**. ViewPoint draws a single Cartesian plot titled "Hello Plotter" with one trace labelled **A0**, scrolling left to right as new samples arrive. With a floating input, the trace shows ambient ADC noise. See the detailed QuickStart guide in the following pages.

QuickStart

Upload a Sketch from the Arduino IDE

This walkthrough uses a Teensy 4.1 but will work similarly with other compatible boards.

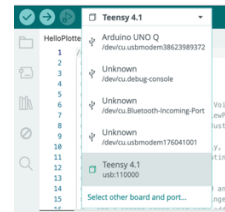
1. Connect your Teensy board to the PC with a USB cable.
2. In Arduino IDE, open:

File > Examples > ViewPoint > Cartesian > HelloPlotter
3. Read through the example comments if you want to understand the test behavior before uploading.

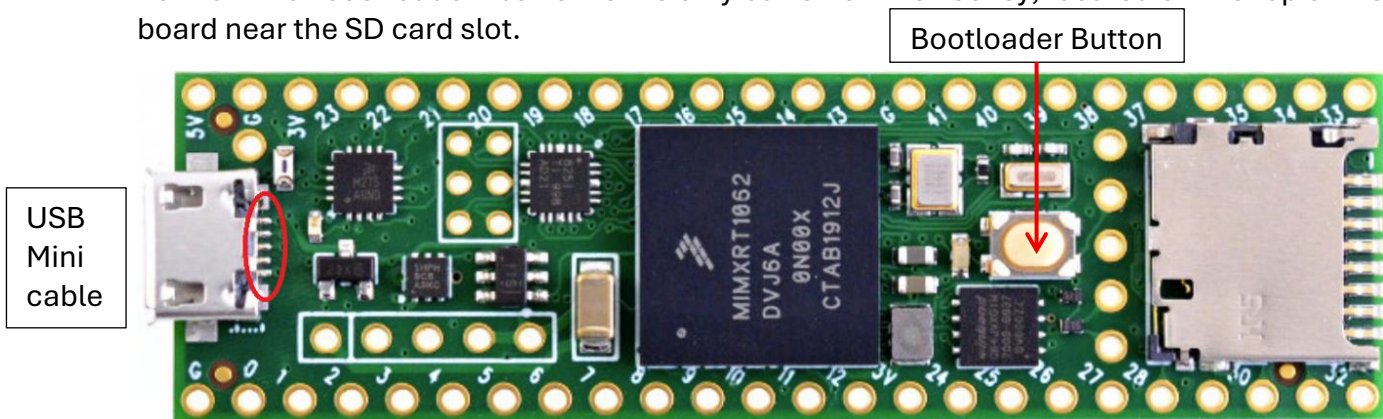
HelloPlotter is our recommended first test. It is the, “Hello, world!” of ViewPoint. The sketch uses the ViewPoint default configuration and streams one live trace labeled A0.



4. In Arduino IDE, choose your exact Teensy board model using the top dropdown.
6. Select the Teensy USB serial port from the board and port selector.
7. Click Upload: 
8. Wait for the compile and upload process to finish.



After programming, the Teensy may briefly disconnect and reconnect. This is normal. You may also see a dialog telling you to press the “Bootloader” button on the Teensy. This is also normal. The “bootloader” button is the only button on the Teensy, located on the top of the board near the SD card slot.

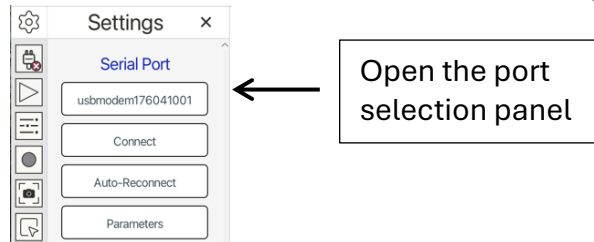


The sketch contains additional details on use with other MCUs (e.g., Arduinos and ESP32).

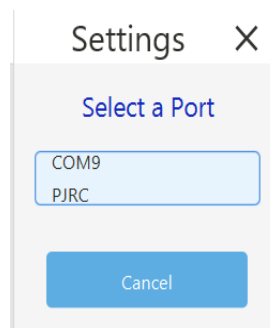
Open ViewPoint™ and Connect to the Teensy

1. Close Arduino Serial Monitor and Arduino Serial Plotter if either is open.
2. Open ViewPoint™.

3. In ViewPoint™, open the Settings panel by clicking the top left Gear icon 



4. In the Serial Port section, click the port button (the top button) and choose the Teensy device. The correct port will clearly say “PJRC: Teensy” or “Teensyduino: Teensy”



5. Click Connect:



Note: Because the Teensy 4.1 is running on emulated serial over USB, specific serial parameters are unused.



Disconnected

Connect/Disconnect icon looks like a plug. The red x signifies no active connection while the green checkmark signifies an active connection.

Meanwhile, the Play/Pause button stops and starts streaming without disconnecting from the serial port.

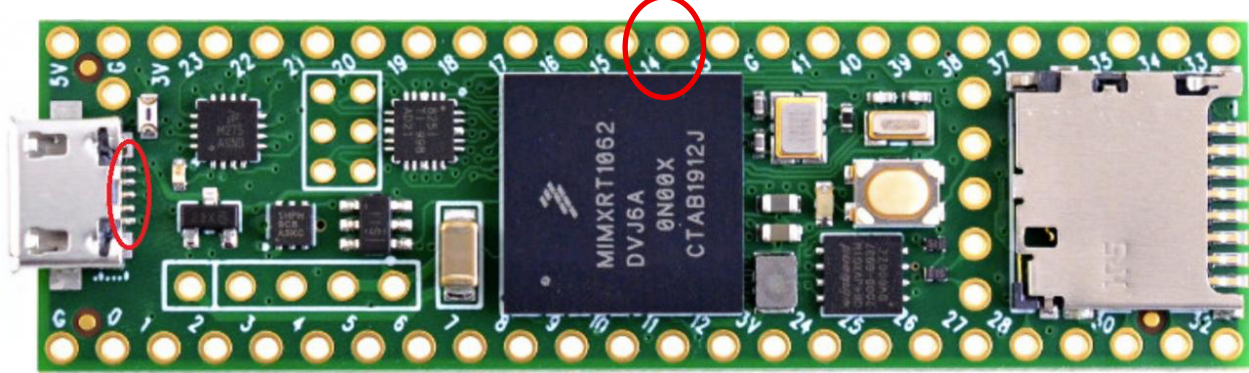


Connected

Confirm the First Live Plot

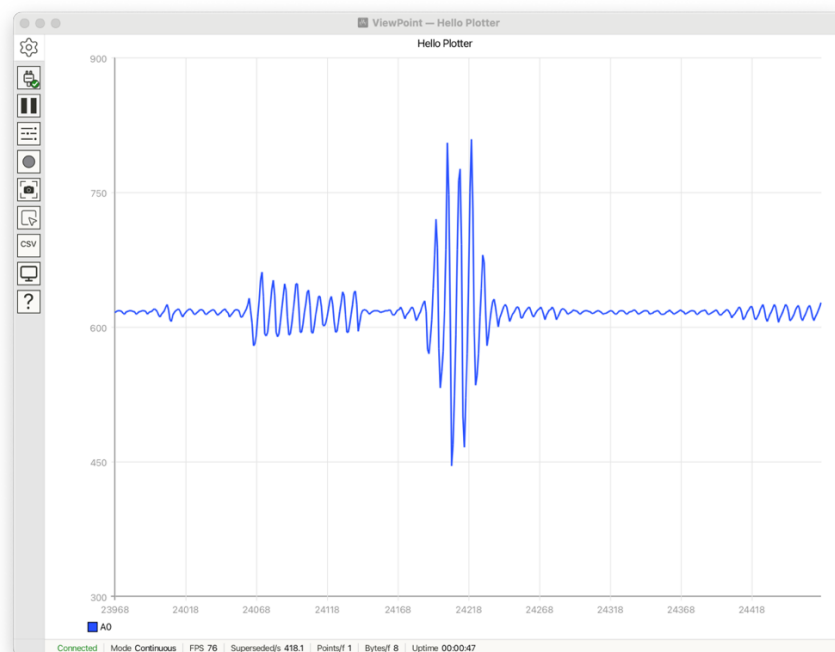
After the connection is established, ViewPoint should begin showing a live trace named A0. A0 is sampling at ~500Hz on pin 14 (Analog 0) using the Teeny's onboard ADC with 10-bit resolution.

Pin 14 (A0)



If nothing is connected to A0, the signal will usually show floating analog noise. That is still a valid first-pass test because it confirms that data is being streamed from the Teensy into ViewPoint.

For a more obvious changing signal, see [Signal Input Requirements](#) first, and then connect a signal generator, potentiometer, or sensor to A0 and watch the plot update in real time. Once you've added the "ADC" library to your Arduino libraries folder `SignalAnalyzer\_A0` is a more robust live signal example that offers 800kSPS and 400kHz bandwidth in both the time and frequency domain. Be careful, depending upon your MCU, it's recommended to use 3.0VPP with 1.65 offset and 50 Ohm termination.



Signal Input Requirements (Teensy 4.1)

Before diving into VoidLoop ViewPoint™, it's important to note the input signal requirements for sketches like *SignalAnalyzer_A0* and *LiveLissajous*.

When interfacing an external signal or function generator with the Teensy 4.1 analog input, it is critical to ensure that the input signal remains within the safe operating range of the ADC.

Voltage Range Constraints

The Teensy 4.1 ADC operates over a **0 V to 3.3 V input range**. Applying voltages outside this range can result in:

- ADC clipping (invalid measurement)
- Nonlinear distortion
- Potential damage to the microcontroller

To safely interface a bipolar signal (e.g., sine wave centered around 0 V), the signal must be **level-shifted and amplitude-limited**:

- **Maximum amplitude:** < 3.3 V peak-to-peak
- **Recommended offset:** 1.65 V (midpoint of ADC range)

50 Ω Output Considerations

Many laboratory signal generators assume a **50 Ω load impedance**. This affects the actual output voltage:

If the generator is set to output (e.g.) 3.0 Vpp **into 50 Ω** , but the Teensy input is **high impedance**, the actual voltage seen can be **~2× higher** than expected.

This can easily exceed the 3.3 V limit and damage the input.

Recommended Mitigation

Use one of the following:

- Enable **“Hi-Z” output mode** on the signal generator (if available), or
- Use a **50 Ω feedthrough termination** (BNC terminator) at the input

This ensures the generator behaves as intended and prevents overvoltage conditions.

Desktop App and Arduino Library: Who Does What?

The Arduino Library Can Define

- Plot type, such as Cartesian, Scatter, or Polar
- Update mode, such as Continuous or Frames
- Packet Size
- Trace labels and colors
- Axis ranges and units
- Plot titles and labels
- Display modes, such as Spectrogram or Persistence
- And the actual streamed data

The Desktop App Handles

- Port discovery and connection
 - Live rendering and layout
 - Viewing and editing plot properties
 - Constraining axes, enabling features, hiding traces
 - Analysis, markers and inspection tools
 - Serial monitoring
 - Screenshot and screen recording
 - CSV capture
 - Diagnostics and runtime feedback
-

Common Terms

Term	Meaning
Plot	A chart area or viewport. A session may have one plot or several.
Trace	A stream of values drawn inside a plot. A single sketch can send multiple traces.
Packet Size	How many points are grouped together before a frame or block is considered complete. For Cartesian plots, this is the number of points drawn across the horizontal axis.
Continuous Mode	Data updates as a running stream. Good for live telemetry or time-series monitoring.
Frames Mode	Data is sent as bounded frames. Good for FFTs, windows, or repeated fixed-size datasets.
Cartesian	Standard X-Y plotting, often used like a time or sample-index plot. User defines the Y value, X is auto-incremented.
Scatter	Paired-value plotting. Useful for trajectories, phase plots, and constellations. User defines X and Y.
Polar	Radial plotting using angle and magnitude. Useful for directional or rotational data. User defines rho and optionally theta
Display Mode	A rendering style layered onto a plot, such as Persistence, Spectrogram, or Gradient. The default is None.
Marker	An inspection tool for reading or tracking a point of interest on a trace. Several marker modes exist including Standard, Delta, Harmonics, and Spanning.
Derived Trace	A trace computed from another trace, such as a moving average or max hold.
Serial Monitor	A text-oriented tool for viewing serial traffic and optionally sending commands.

Plot Types

Plot Type	Best For	Typical Data Shape
Cartesian	time-series, sampled values, general telemetry	one value per point, or standard X-style progression
Scatter	phase plots, paired channels, constellations, trajectories	paired values (x, y)
Polar	directional, rotational, angular data	magnitude plus angle

See [Recipe 12](#) to see Scatter in action and [Recipe 11](#) for additional information on Polar.

Update Modes

Update Mode	Best For	Notes
Continuous	live streaming telemetry	updates continuously as new values arrive, with a configurable delay after each point
Frames	FFTs, windows, fixed-size blocks	sends bounded packets and can mark frame completion, with an optional delay after each packet

See [Recipe 4](#) for more information on update modes.

The update mode is generally set from the ViewPoint library and controlled via serial commands, however, it can be manually cycled in the Diagnostics Status Bar at the bottom edge of the screen, by clicking on the **Mode** section. None → Continuous → Frames → None.

Display Modes

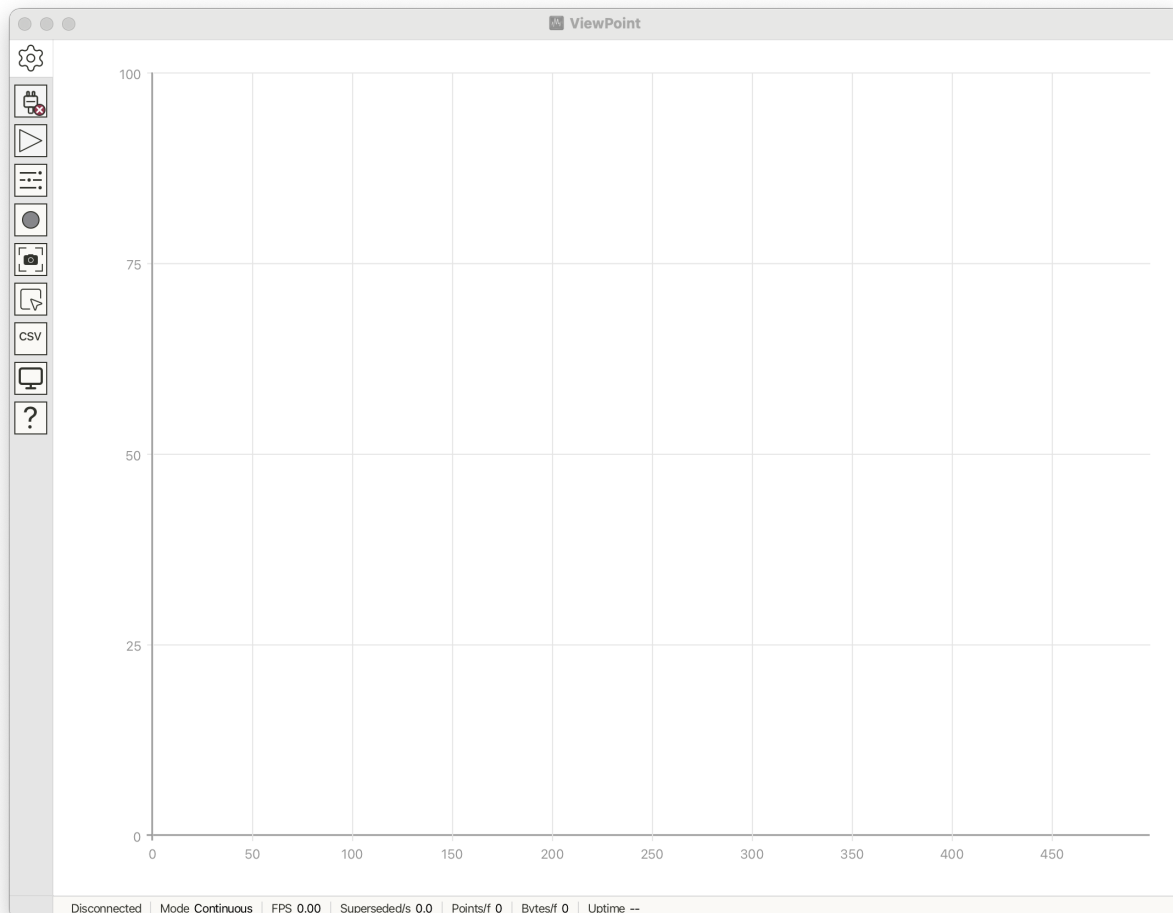
Display modes are **only available for Cartesian plots in Frames mode**. If you are using Continuous mode or a non-Cartesian plot type, these modes will not be available.

Display Mode	What It Does
None	normal plotting without an added persistence effect
Persistence	keeps older frames of samples visible with alpha-blending
Spectrogram	waterfall or time-frequency style rendering
Gradient	accumulates intensity (color-spectrum) for each pixel based on hit count

See [Recipe 13](#) for Display Modes in action.

Using VoidLoop ViewPoint™

Main Window Layout



Main app window. Full window showing the sidebar, plot canvas, and side panel.

The ViewPoint window is organized into three main areas:

- **Sidebar** — a vertical icon strip on the left edge. Each icon opens a feature or toggles a control.
- **Canvas** — the central area where plots are rendered, AKA the plotting area.
- **Side Panel** — opens to the right of the canvas when you click certain sidebar icons (Settings, Controls, Analysis).

- **Diagnostics** — status bar located at the bottom of the canvas and includes current information regarding the connection, update rate, and errors (if any occur).
- **Additional Windows** — the Serial Monitor, Report an Issue..., and About dialogs open in a separate window

Sidebar Controls

The sidebar runs top to bottom in this order:



Sidebar controls. Close-up of the sidebar icons and their order.

1. **Settings** — opens the connection and preferences panel.
2. **Connect / Disconnect** — toggles the serial connection.
3. **Play / Pause** — starts or pauses live data streaming (does not disconnect).
4. **Controls** — opens the plot and trace editing panel. Diagnostics are at the bottom of this panel.
5. **Screen Recording** — starts or stops video capture of the session.
6. **Screenshot** — captures a still image of the current view (copied to clipboard by default).
7. **Analysis** — opens signal inspection tools including markers and trace statistics.
8. **CSV Capture** — starts or stops raw data logging to a CSV file.
9. **Serial Monitor** — opens the text-oriented serial view.
10. **Help / About** — version, product information, and in-app documentation.

Sidebar Icon Details

Connect/Disconnect and Play/Pause

Connection vs disconnection is indicated within the icon.



Disconnected



Connected

Streaming vs Pause is indicated within the icon as well



Paused



Streaming

Capture: Screen Recording

Recording is indicated by the icon pulsing red



Not Recording



Recording

Capture: CSV

CSV capture is similarly indicated by the icon pulsing red.



Not Capturing



Capturing CSV

voidloop

The stream can be paused while remaining connected. Below are the various states the two icons can have:

Disconnected



Paused



Connected



Streaming



Connected



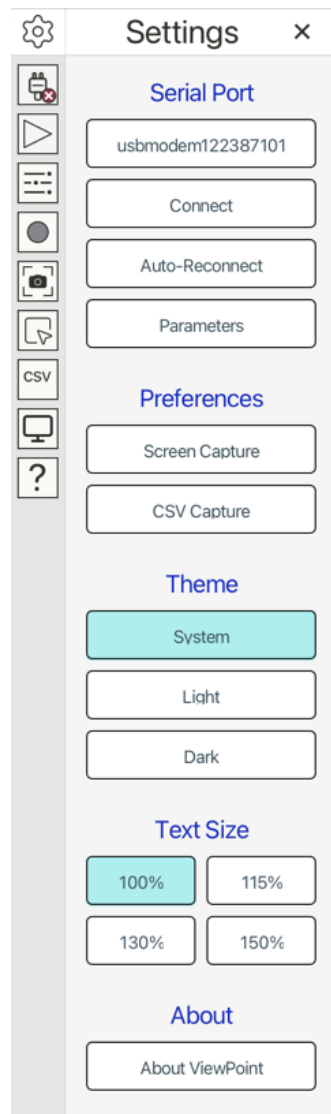
Paused



Main Desktop Features

Settings

The Settings panel is available from the gear icon at the top of the sidebar. From there, users can select a serial port, adjust port parameters, and enable Auto-Reconnect to automatically restore the last connected port after firmware uploads. Settings also include preferences for screen and CSV capture, theme selection, and application-wide font size presets. On macOS, system theme changes may not always be applied automatically.

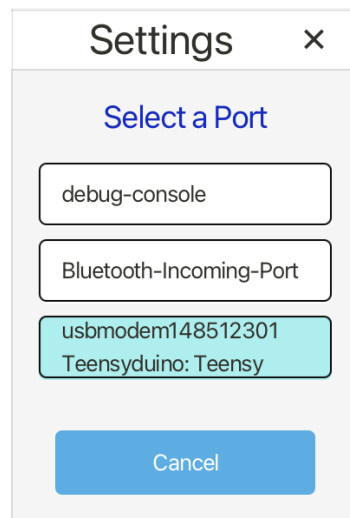


Connection Workflow

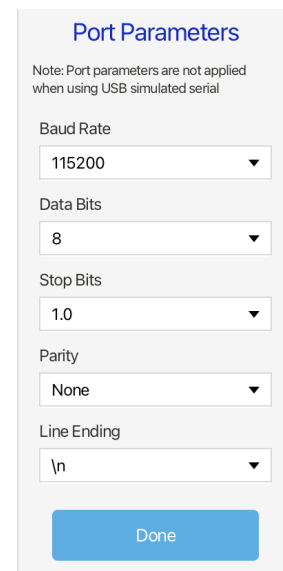


Open Settings from the sidebar to access port selection and serial parameters. Select your device from the port list, then click Connect (the plug icon or the “Connect” button on the Settings panel).

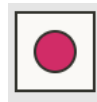
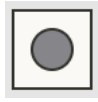
If the serial port you are looking for is not detected, enVsure Arduino Serial Monitor and Arduino Serial Plotter are closed.



The port selection panel (accessed by clicking the first button on the settings panel) shows what ports are currently available. The button identifies both the manufacturer and product when available.



Serial port parameters can be accessed and edited by selecting the **Parameters** button on the main settings panel. These parameters are only relevant for standard non-USB emulated serial.



Screen Capture Settings

Settings ✕

Screen Capture

Save to

☒ Capture plot only

Screenshot

File type

☒ Copy to Clipboard

☐ Save to File

Recording

Backend

Audio

System audio requires BlackHole on mac...

FPS

Screen Capture controls how ViewPoint™ saves screenshots and recordings. The **default save location** is within the desktop directory, under individual **screenshots** and **recordings** folders. This location can be changed with the Save to **Choose...** button.

Capture plot only exports just the plot area; turn it off to capture the full application window.

Screenshots can be copied to the clipboard, saved to file, or both, with support for PNG and JPG.

For recording, **JCodec** works out of the box and requires no external install, but it is best for lower frame rates (10 FPS) and does not support audio.

FFmpeg requires a [separate installation](#), but delivers much higher frame rates, better quality, and audio support.

PNG Sequence saves individual PNG frames at the selected frame rate instead of a single video file.

On macOS, system-audio recording requires BlackHole or a similar virtual audio device.

FFmpeg is external software with no affiliation to VoidLoop or ViewPoint™. Using it requires a separate installation – available with package managers per operating system or by following the link available above and in VoidLoop ViewPoint™.

Once FFmpeg is installed, VoidLoop ViewPoint™ should automatically detect it, however, if it does not, ensure FFmpeg is added to your system path or enter the custom path into the text box in the settings panel. The textbox is only visible if FFmpeg is not automatically found.

When FFmpeg is installed and the backend selection is set to **Auto**, FFmpeg will be used by default.

Screen capture is limited by default to either 1024 MB or 60 minutes, these settings can be adjusted or turned off from within the settings menu and persist across application runs.

PNG sequence captures are limited to 10,000 frames.

Recording

Backend

FFmpeg not found on PATH. Install FFmpeg to enable the FFmpeg backend and audio recording.

Audio

System audio requires BlackHole on mac...

FPS

Limits

☒ File size

MB

☒ Duration

min

☒ PNG frame count

frames



Settings

CSV Capture

Save to

Choose...

/Users/zachariah/Desktop/CSV

Format

Timestamp Format

ISO_Nanos

Limits

File size

250

MB

Rows

1000000

rows

Inputs

Keyboard

Ready

Start Capture

Previous

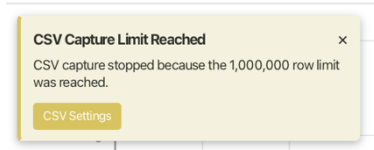
CSV Capture saves trace data to a .csv file for analysis and recordkeeping. Each capture includes timestamps and the active trace values, with selectable timestamp formats to match different workflows.

Keyboard capture can also be included, adding typed notes or event markers as a separate CSV column during recording. Use CSV capture when you need raw numeric data rather than screenshots or video.

Timestamps can be sent from the ViewPoint™ library **send** function.

CSV capture is primarily supported for **Continuous** mode although no restrictions are in place for **Frames** mode.

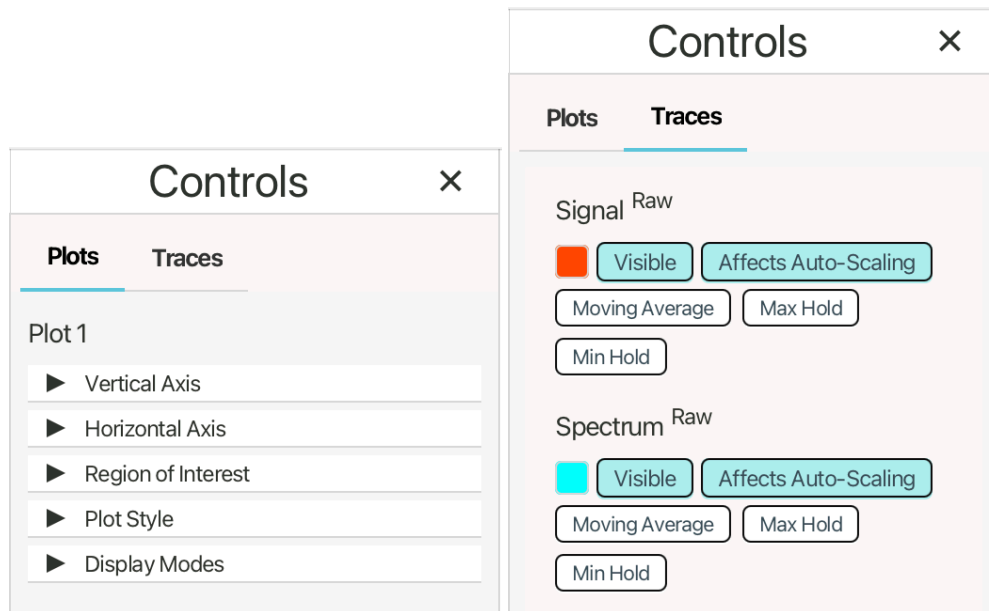
Restrictions are in place for the maximum allowed file size or number of rows. These limits are adjustable from within the CSV Capture settings panel and persist across application runs. When limits are reached, a notification alerts the user that the capture was halted.



Example Output

Temp	Vib	Press	Status
35.23	6.35	974.15	1.0
35.24	3.81	973.89	1.0
35.31	1.35	973.75	1.0
35.39	-0.71	973.31	1.0
28.91	-3.02	1045.46	1.0
28.98	-2.97	1045.04	1.0
29.07	-2.05	1045.53	1.0
29.16	-2.62	1045.11	1.0
29.12	0.91	1045.2	1.0
29.04	3.6	1045.59	1.0
29.1	6.57	1041.56	1.0
29.16	3.84	1037.93	1.0
29.17	3.14	1034.19	1.0

Controls



Controls panel. Plot and trace settings.

The Controls panel lets users view and edit plot and trace settings: axis ranges, titles, labels, trace colors, and more. Users can enable display modes, add derived traces (moving average, max/min holds), hide traces and edit trace colors. Everything that can be done with serial commands and more can be done directly within the Controls panel.

Selecting a Plot

The Controls panel edits the currently selected plot. To select a plot, click inside the plot area. When more than one plot is visible, the selected plot is indicated by a subtle blue triangular marker along the left edge of the plot – like the illustration on the left side of this paragraph.

You can also select plots with the keyboard using Ctrl+1 through Ctrl+8. The shortcut number is 1-based: Ctrl+1 selects the first (top) plot, Ctrl+2 selects the second plot down, and so on. If the requested plot number is higher than the number of visible plots, VoidLoop ViewPoint™ selects the last available plot.

Selected Plot Identifier



Axis Configuration

The Axis sections control the scale, range, labels, units, and tick layout for the selected plot. Cartesian and scatter plots show Vertical Axis and Horizontal Axis sections. Polar plots show Radial Axis and Angular Axis sections.

Labels and Units

Cartesian and scatter axes can display a Label and Units. The label is the axis title shown beside or below the axis for the vertical and horizontal axis respectively. Units are short suffixes used in tick labels and readouts, such as V, Hz, or s. Each unit takes the place of the maximum tick value on the axis.

Labels

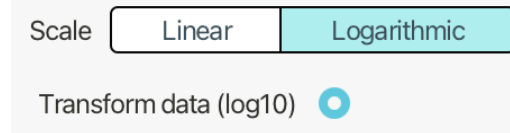
Label

Units



Scale

Axes that support logarithmic scaling can be set to Linear or Logarithmic. Logarithmic scale is available on Cartesian vertical axes, scatter horizontal and vertical axes, and polar radial axes. It is not available on Cartesian horizontal sample-index axes or polar angular axes.



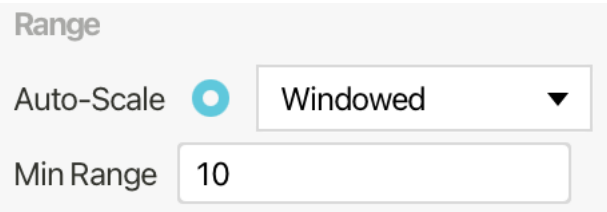
Scale

Transform data (log10) ☒

When Logarithmic is selected, *Transform data* ($\log_{10}$) controls how values are handled. When enabled, VoidLoop ViewPoint™ applies a $\log_{10}$ transform to incoming values. When disabled, the data is treated as already logarithmic, and the axis bounds must remain positive.

Auto-Scale Strategies

When Auto-Scale is enabled, the Scaling Strategy controls how VoidLoop ViewPoint™ chooses visible bounds over time. These strategies are available for auto-scaled linear data axes, except on Cartesian horizontal axes, which auto-increment by sample index.



Range

Auto-Scale ☒

Min Range

Min Range sets the smallest span the axis is allowed to show while auto-scaling, defaulting to 10.



Min Range

Windowed is the default method, scaling to the current data window with additional padding and clean tick intervals.

Expand Only grows the axis when new minimum or maximum values appear but does not shrink back automatically.

Peak Decay holds recent peaks and gradually relaxes back toward the current data range. The *Decay Rate* parameter controls how quickly the held bounds relax.



Decay Rate

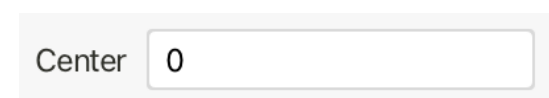
Debounced reduces visual jitter by limiting how often bounds update. *Interval* sets the minimum time between updates. *Threshold* sets the relative change required to force an immediate update.



Interval (ms)

Threshold

Symmetric Zero keeps the axis symmetric around a center value. *Center* (default: 0) sets the midpoint of the visible range.



Center

Baseline Zero locks one side of the axis to a baseline value and scales the other side. *Baseline* (default: 0) sets the anchor value. *Lock Min* keeps the minimum at the baseline; when Lock Min is off, the maximum is locked instead.

Baseline Lock Min ☒

Robust Percentile ignores outliers by scaling to percentile bounds rather than absolute minimum and maximum. *Low Percentile* and *High Percentile* set the percentile range used for scaling.

Low Percentile

High Percentile

Ticks

Ticks are available for manual, non-logarithmic axes.

Auto enables derived tick spacing.

Divisions divides the visible range into a specified number of major intervals.

Step Size provides a way to specify major and minor tick spacing directly. Major ticks are labeled. minor ticks are drawn as smaller grid intervals and are not labeled.

Ticks

☒ Auto ☐ Divisions ☐ Step Size

Ticks

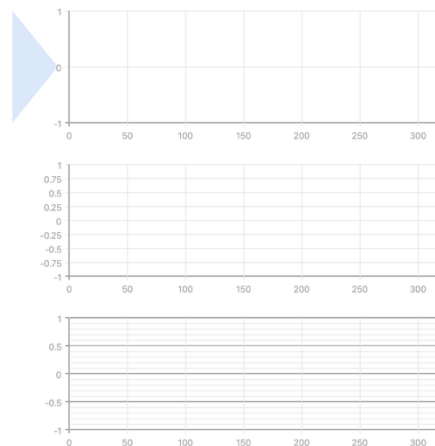
☐ Auto ☒ Divisions ☐ Step Size

Divisions

Ticks

☐ Auto ☐ Divisions ☒ Step Size

Major Minor



Bounds and Range

By default, an axis auto-scales to fit the incoming data. The Cartesian horizontal axis auto-increments with the sample index. When Auto-Scale or Auto-Increment is disabled, **Bounds: Min to Max** sets the fixed visible range for the axis. The minimum must be less than the maximum.

Range

Auto-Scale ☐

Bounds: Min to Max

to

Vertical Axis – Auto-Scale

Range

Auto-Increment ☐

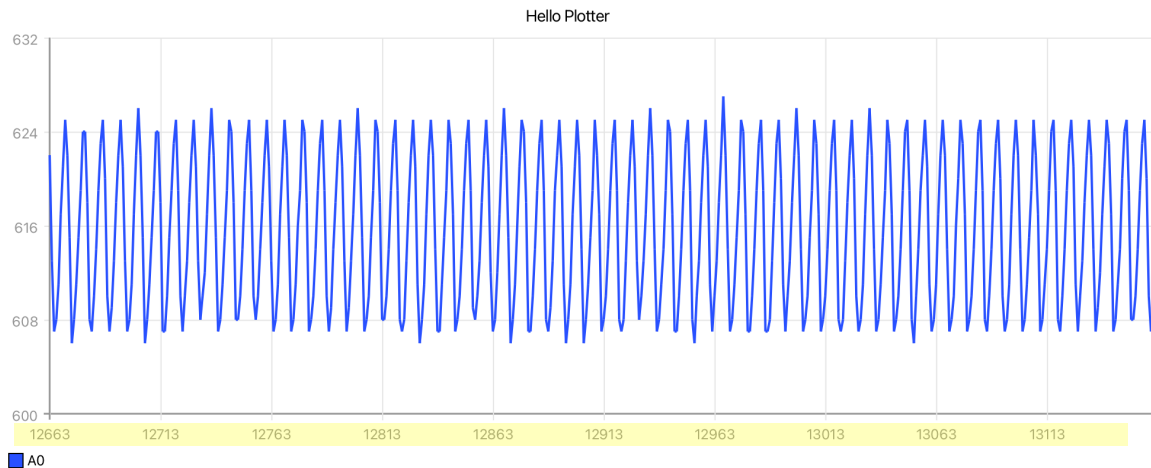
Bounds: Min to Max

to

Horizontal Axis – Auto-Increment

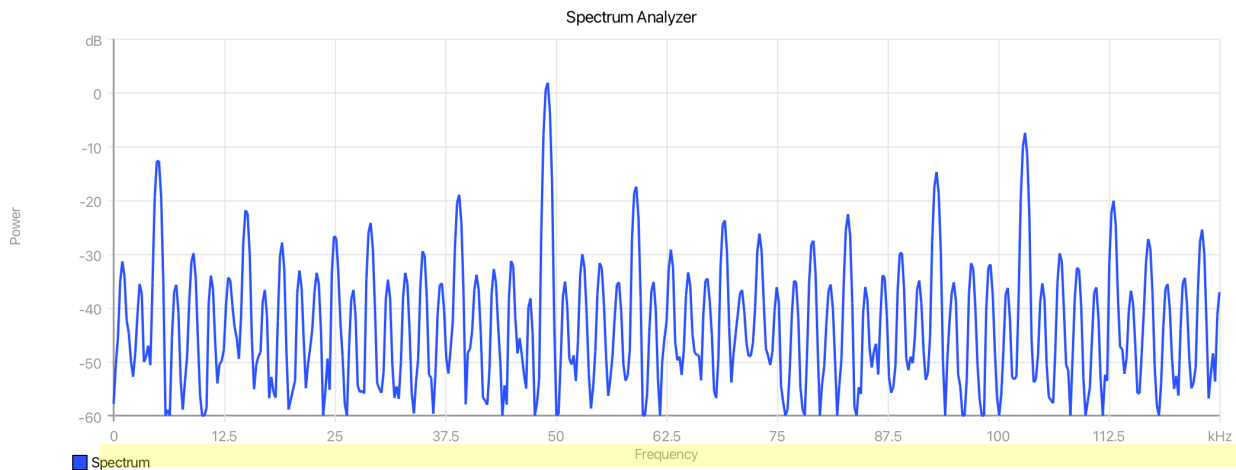
Horizontal Axis: Auto-Increment

For Cartesian plots, the horizontal axis represents the sample index. When Auto-Increment is enabled, the horizontal axis advances automatically from the first sample in the current data window to the last sample.



Horizontal Axis: Domain Labels

There are use-cases where the horizontal axis represents a domain other than the sample index, like the frequency domain, this can be achieved by setting the axis range for the horizontal axis. This action disables Auto-Increment and adjust the horizontal axis labeling, not the actual bounds – by default the same set of data is in view, only the tick labels are adjusted.



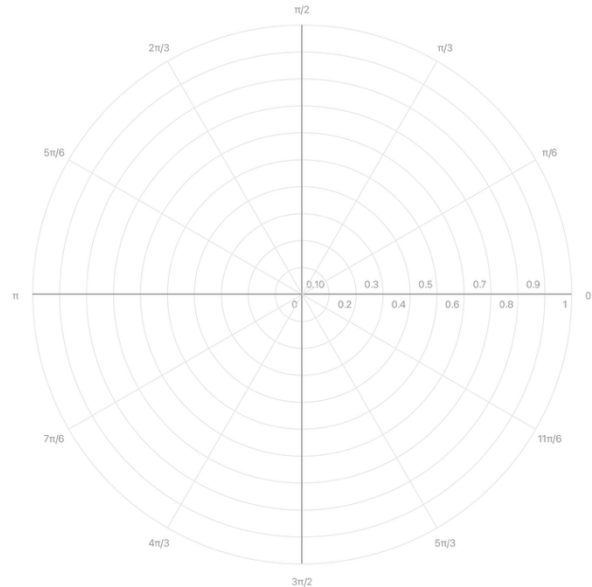
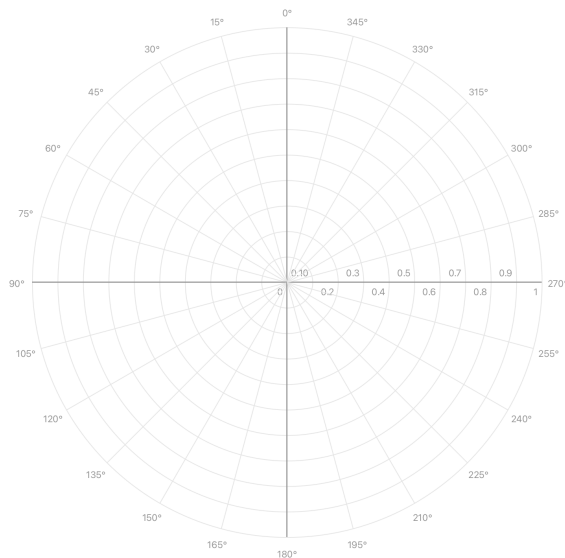
The vertical axis uses Auto-Scale to fit the incoming trace data. Additional information is available on the various auto-scaling techniques available under [Auto-Scaling Strategies](#).

For scatter plots, both horizontal and vertical axes represent data values. When Auto-Scale is enabled, each axis can scale to the data assigned to that axis.

Polar Properties

Polar plots use a Radial Axis and an Angular Axis. The Radial Axis behaves like a value axis and supports auto-scale, [scaling strategies](#), manual [bounds](#), [ticks](#), and [logarithmic scale](#).

The Angular Axis controls theta. **Theta Units** switches the angular display between *Degrees* and *Radians*. **Offset** rotates the angular axis by the specified amount. **Step** sets the angular spacing between major divisions.



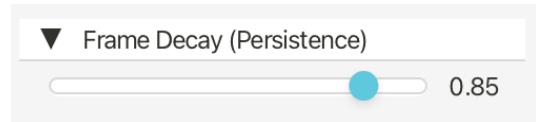
Theta Units	Degrees
Offset	90
Step	15

Theta Units	Radians
Offset	0
Step	30

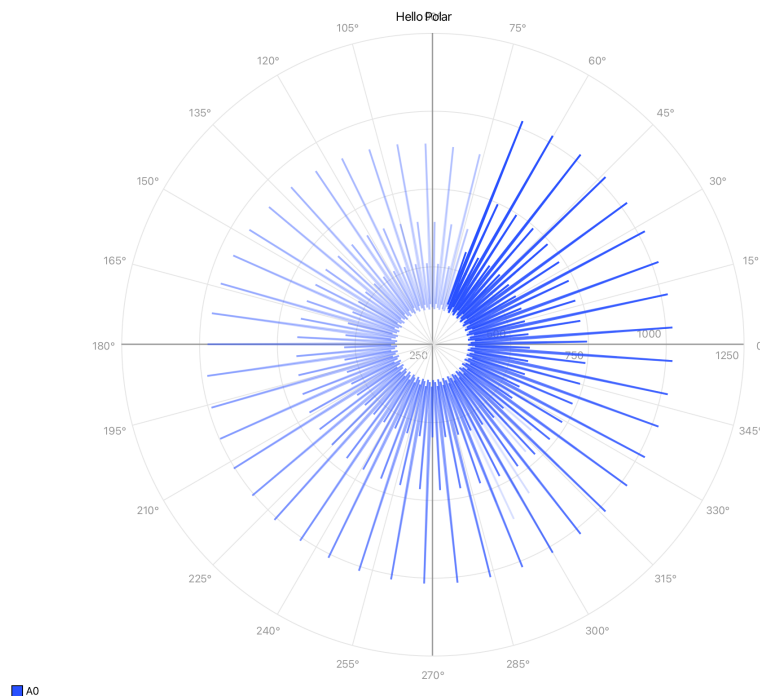
Polar plots do not expose Cartesian-style axis labels or units in the Controls panel.

Frame Decay (Scatter and Polar plots)

Frame Decay applies a per-sample alpha gradient across the points within a single frame, mapping sample age to opacity: the most recently received sample is drawn fully opaque, and each preceding sample is attenuated by a decay factor. The decay value sets the length of this gradient, higher values extend the visible trail by fading older samples more slowly, while lower values collapse the frame down to only the newest points.



This matters because scatter and polar traces are not single-valued along a monotonic axis. Unlike a time-domain line plot, where each x maps to one y and ordering is implicit in position, a scatter or polar trace can revisit the same region, wind through multiple revolutions, or cross over itself within one frame. Spatial position alone therefore carries no information about sample order. Frame Decay restores that ordering by encoding it as an opacity ramp, so the eye can recover direction of travel and distinguish leading edge from trailing history even where the geometry overlaps.

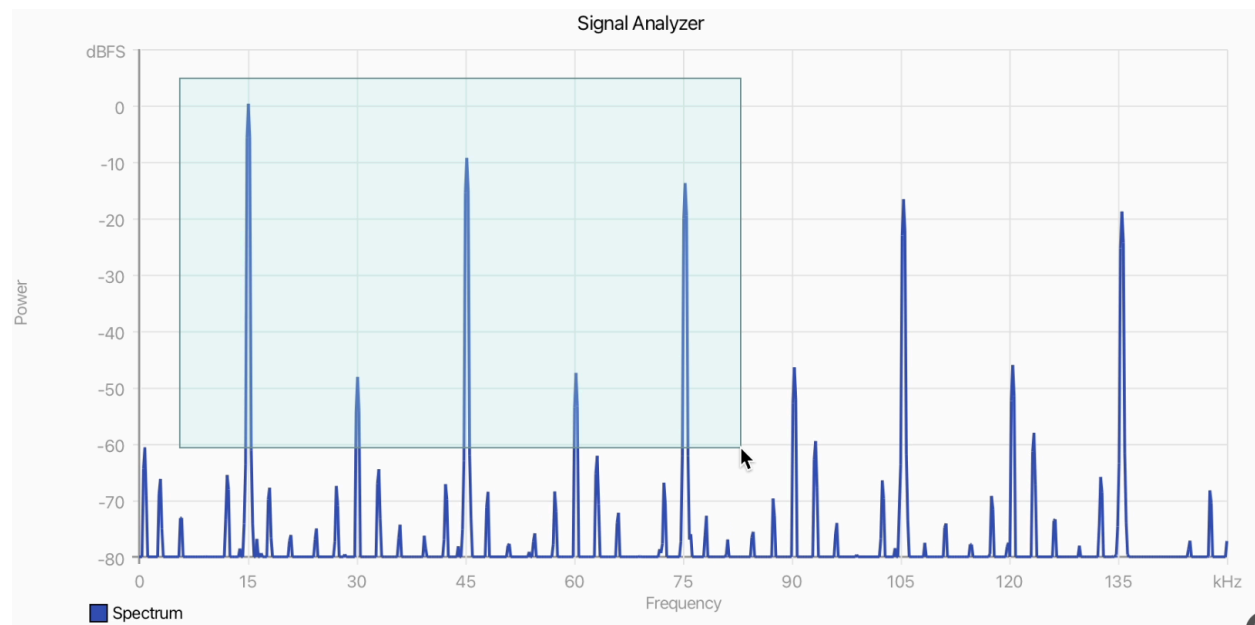


Frame Decay is distinct from the Persistence display mode, and the two operate at different scopes:

- Frame Decay acts within a single frame, in both Continuous and Frames update modes, fading samples relative to one another based on their order of arrival. It governs intra-frame point opacity only.
- Persistence acts across frames, in Frames mode only, compositing previously rendered frames beneath the current one to show evolution over time.

Region of Interest

The Region of Interest section lets you zoom into part of a Cartesian or scatter plot.



Select enables ROI selection mode. With selection mode active, **click and drag** inside the plot to choose a region. Releasing the mouse zooms the plot to that region. Press **Z** to toggle selection mode from the keyboard.

Pan is available after the plot is zoomed. When Pan is enabled, drag inside the plot to move the zoomed region. Press **P** to toggle Pan, or **right-click and drag** to pan temporarily.

Fit to Window restores the full plot bounds and keeps ROI selection active. **Double-clicking** inside the plot performs the same fit-to-window action. Press **F** while ROI mode is active to fit to window. Press **Esc** to exit ROI mode.

Note: These keyboard shortcuts are case-insensitive – Shift-Z and `z` both activate zoom, Shift-P and `p` both activate panning, etc.

Region of Interest

Region of Interest

Region of Interest

Plot Style

Plot Style controls visual styling for the selected plot.

Plot Colors sets the plot background, axis and tick text color, major grid line color, and minor grid line color.

Stroke and Size set the major grid line thickness, minor grid line thickness, and axis/tick label font size.

Labels controls whether horizontal and vertical axis labels are shown.

Reset All clears plot-specific style overrides and returns the plot to the default style.

▼ Plot Style

Plot Colors

Background Text

Major Minor

Stroke and Size

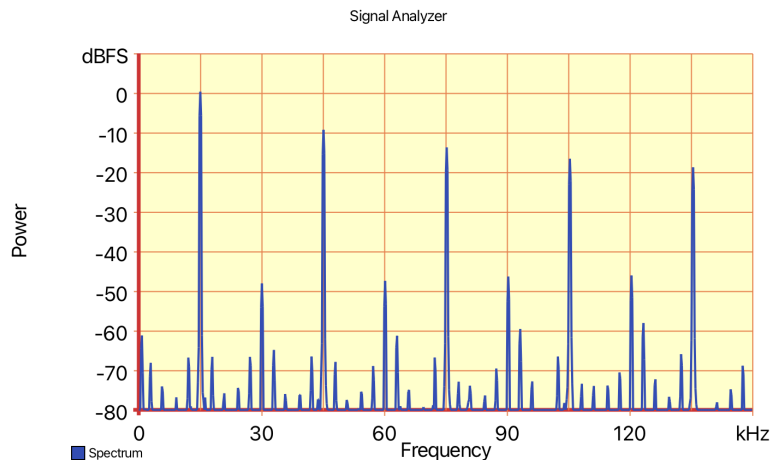
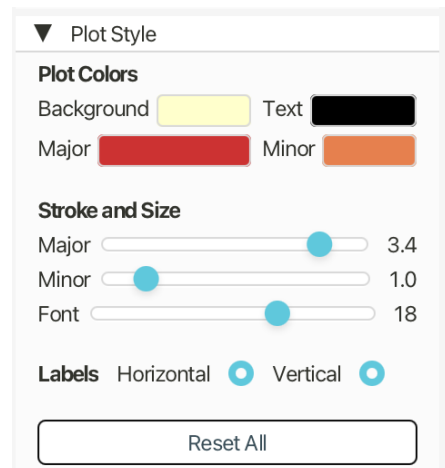
Major 1.5

Minor 1.0

Font 12

Labels Horizontal ☒ Vertical ☒

Reset All



Display Modes

Display Modes are available for non-polar plots when the plotter in the *frames* update mode. Only one display mode per plot can be active at a time.

Grid Opacity controls how visible the grid lines are while a display mode is active. Enabling a display mode usually lowers grid opacity so the rendered history is easier to see.

Persistence

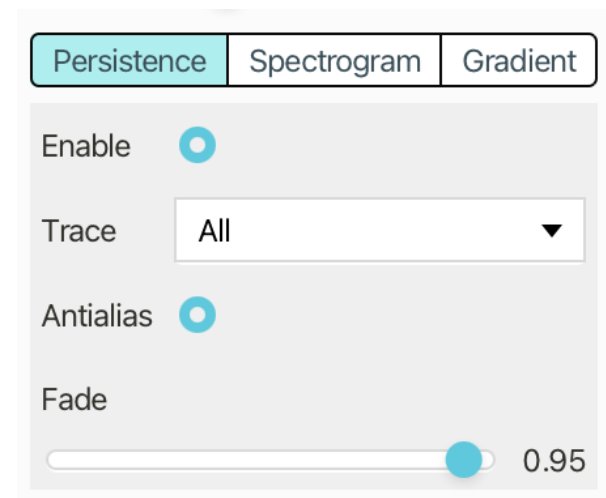
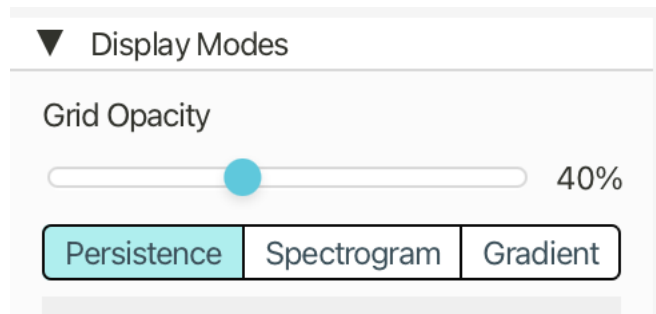
Persistence overlays recent frames to show signal persistence over time.

Enable turns the mode on or off.

Trace selects which trace contributes to the persistence image, or *All* for all traces.

Antialias smooths persistence line edges.

Fade controls how quickly older persistence frames disappear; lower values fade faster, while higher values retain history longer.



Spectrogram

Spectrogram renders incoming data as a time-history image where amplitude is mapped to color.

Enable turns the mode on or off.

Draw Traces overlays normal trace lines on top of the spectrogram.

Trace selects the source trace.

Down-Sample chooses how each time bucket is reduced: **Max** preserves peaks, while **Average** smooths values.

Update Rate sets the time bucket size.

Threshold clamps values below the amplitude floor to black.

Color Map selects the color gradient.

Persistence

Spectrogram

Gradient

Enable ☒

Draw Traces ☐

Trace

Spectrum ▼

Down-Sample

Max ▼

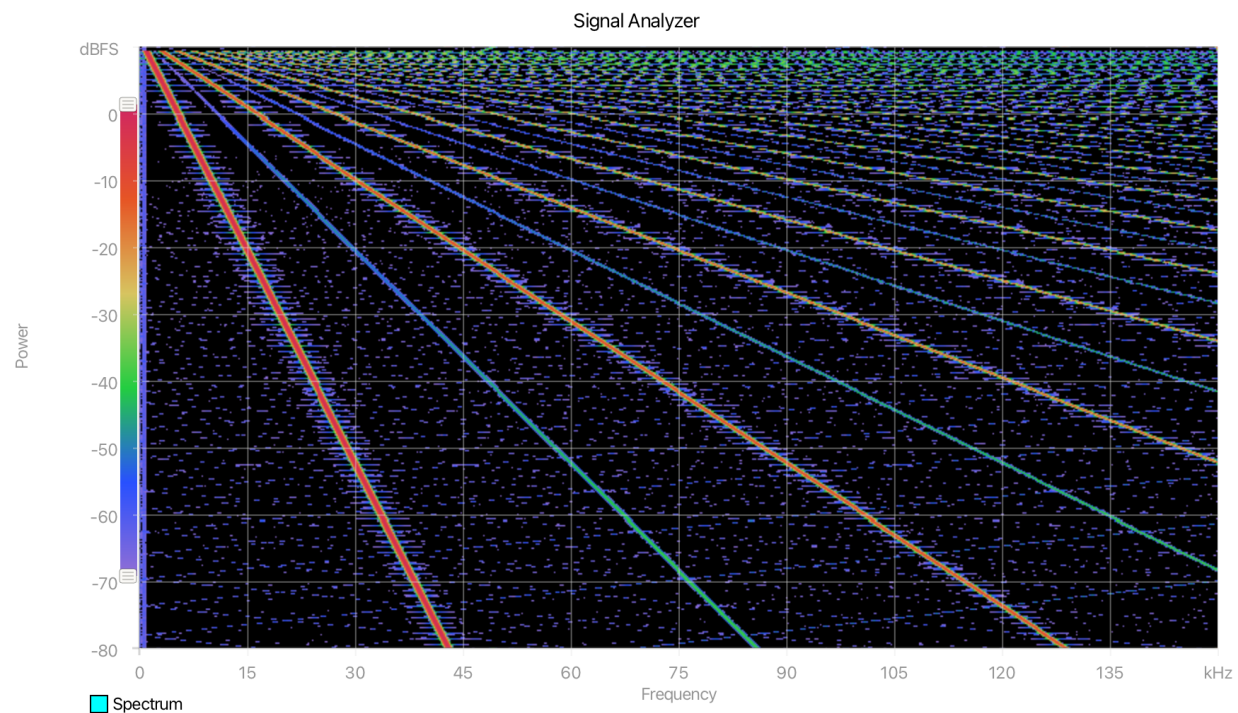
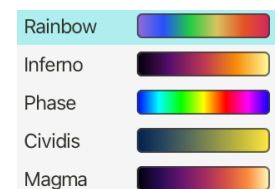
Update Rate

1ms

Threshold ☒

Color Map

Rainbow ▼



Gradient

Gradient accumulates hit counts per pixel and maps frequency to color.

Enable turns the mode on or off.

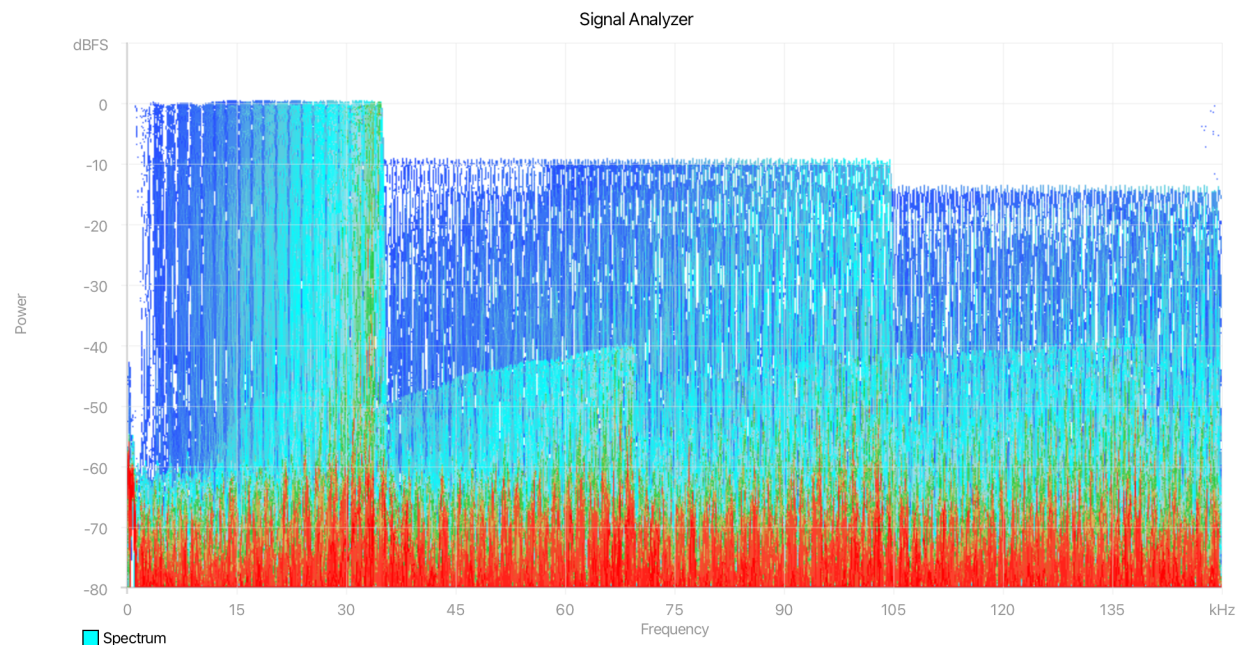
Draw Traces overlays normal trace lines on top of the gradient image.

Trace selects the source trace, or All.

Decay Frames sets how often accumulated hit counts decay.

Decay sets the per-frame fade factor; lower values fade older data faster.

Persistence	Spectrogram	Gradient	
Enable	☒	Draw Traces	☐
Trace	All ▼		
Decay Frames	7		
Decay	<input type="range"/> 0.95		



Trace Configuration

The **Traces** tab controls how each trace is displayed and how it participates in plot scaling and derived views.

Each trace card shows the trace label, its raw or processed state, color, and available display options.

Visible shows or hides the trace on the plot. Hiding a trace does not delete it, stop the data from being received, or delete it from being saved to CSV.

Affects Auto-Scaling controls whether the trace is included when the plot automatically chooses its axis range. Turn this off when you want to keep a trace visible without letting the trace affect the scale of the plot.

Changing the Trace Label

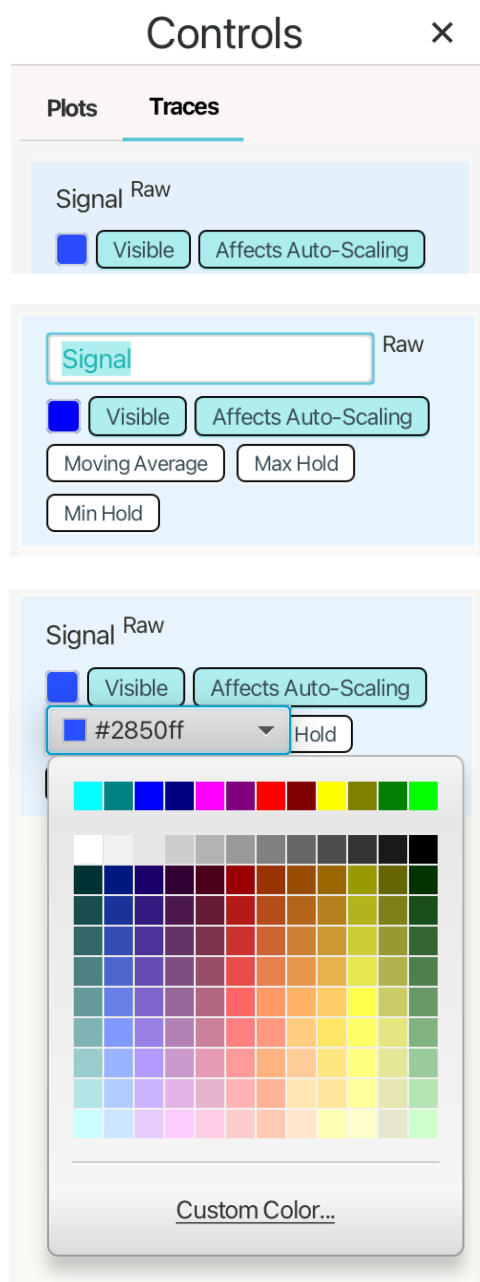
To rename a trace, click the trace label. The label becomes editable.

Type the new name, then press **Enter** or click away to apply the change. The updated label is used in the plot legend, markers, statistics, and other trace selection controls.

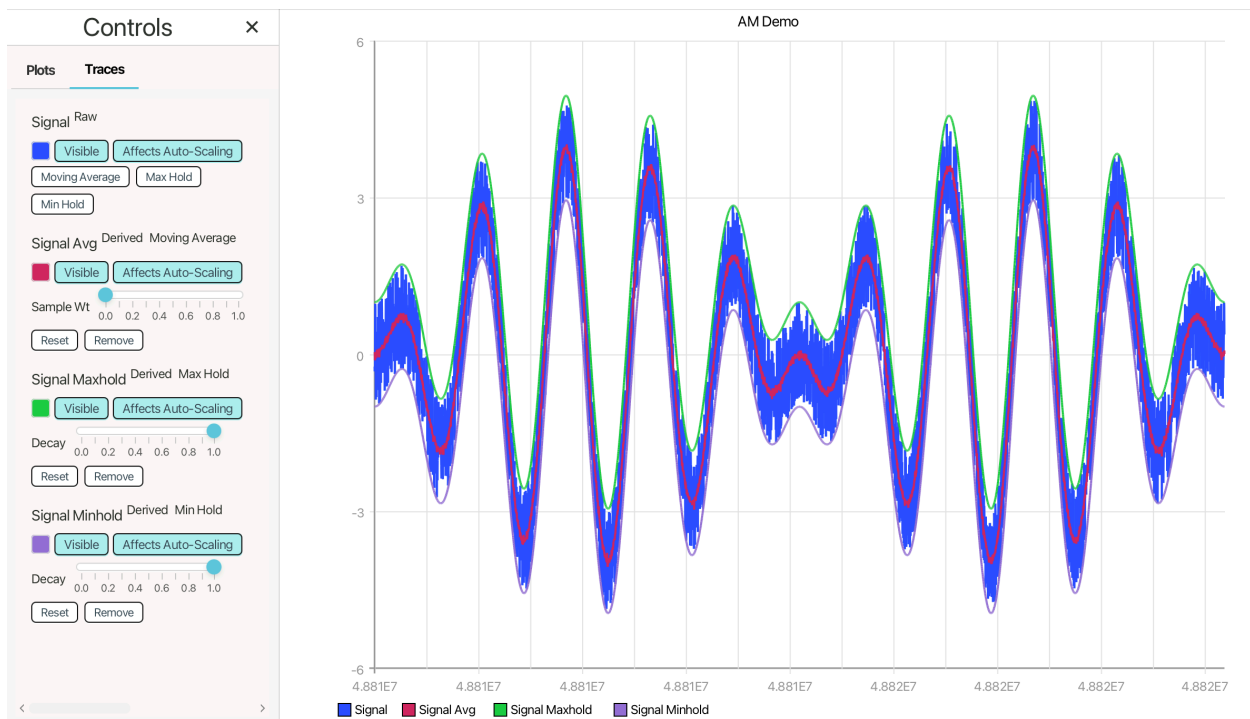
Changing the Trace Color

To change a trace color, click the color swatch next to the trace controls.

Choose a color from the palette, or select **Custom Color** to define a specific color. The new color is applied to the trace line and to matching UI indicators such as legends, markers, and statistics entries.



Derived Traces



Derived traces create a new trace from an existing trace. The original trace remains unchanged, and the derived trace appears as its own trace with its own label, color, visibility, and auto-scaling controls.

Use the derived trace buttons on a raw trace to create a processed version:

Moving Average creates a smoothed trace from the original signal. Use **Sample Wt** to control how strongly new samples affect the average. Lower values respond more slowly and appear smoother; higher values respond more quickly and follow the original trace more closely.

Max Hold creates a trace that holds the maximum observed value over time. Use **Decay** to control how quickly the held maximum falls back toward the live signal.

Min Hold creates a trace that holds the minimum observed value over time. Use **Decay** to control how quickly the held minimum rises back toward the live signal.

Each derived trace can be shown or hidden with **Visible** and included or excluded from auto-scaling with **Affects Auto-Scaling**.

Use **Reset** to clear the derived trace history and start its calculation again from the current signal. Use **Remove** to delete the derived trace.

Diagnostics Status Bar

Connected	Mode Continuous	FPS 0.00	Superseded/s 0.0	Points/frame 4	Bytes/frame 24	Uptime 00:00:50	Link OK
-----------	-----------------	----------	------------------	----------------	----------------	-----------------	---------

The diagnostics status bar is always visible at the bottom of the VoidLoop ViewPoint™ application window and provides insights into the current data stream including a compact summary of current performance

Connected / Disconnected shows whether the primary serial connection is active.

Mode shows the current update mode. **Continuous** commits data as it arrives while **Frames** commits data at frame boundaries. **None** appears before a mode has been selected or when no active mode is available. **Note:** Click to cycle between modes, None → Continuous → Frames → None. This provides a way to force the update mode when data is received from a legacy (non-ViewPoint library) sender.

FPS shows how many new-data frames are being drawn per second. This measures rendered data updates, not idle screen redraws.

Drop/s shows completed frames per second that were not drawn before newer data arrived. In **Continuous** mode this label changes to **Superseded/s**, because older data may be replaced by newer data as part of normal continuous streaming. Superseded is not considered an error.

Points/frame shows the total number of data points in the most recently completed frame across all traces.

Bytes/frame shows the number of raw serial bytes received for the most recently completed frame before parsing.

Uptime shows the elapsed time since the current connection started.

Link monitors the health of the data line. When data is received regularly it reads **OK** whereas if the line is quiet for more than 10 seconds, it will read **Quiet**.

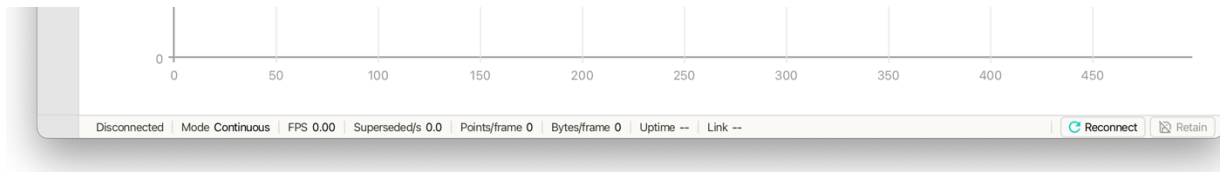
When a receive or parsing issue occurs, the status bar replaces the normal per-frame fields with issue-specific indicators:

RxDrop/s replaces **Points/frame** when incoming serial data blocks are being dropped because the receive queue is full.

ParseErr/s replaces **Bytes/frame** when incoming data blocks are being discarded because parsing or processing failed.

Values that indicate a problem are highlighted in the status bar. If the issue stops, the bar returns to the normal **Points/frame** and **Bytes/frame** display.

Quick Access Settings



Enabled:   **Disabled:**  

At the right end of the [Diagnostics Status Bar](#), two toggle buttons give one-click access to connection preferences you would otherwise change from the Settings panel. Because they live in the always-visible status bar, you can see and change them without opening any panel.

Each button shows its state in place: a **teal icon** when the setting is on, and a **dimmed icon and label** when it is off. Hover over a button for a tooltip describing its current state and **left-click** to toggle it. When the window is narrow the text labels collapse to icons to save space, but the buttons themselves always remain visible.

[Reconnect](#)  

Reconnect toggles **Auto-Reconnect**. When it is on, VoidLoop ViewPoint™ automatically restores the last connected port when that device reappears. For example, after a firmware upload causes the board to re-enumerate. When it is off, you reconnect manually using the plug icon or the **Connect** button on the Settings panel.

This is the same preference as **Auto-Reconnect** on the [Settings](#) panel, changing it in either place updates the other.

[Retain App-Side Configuration](#)  

Retain toggles whether your manual configuration and overrides are kept across connections.

When it is on, the adjustments you have made in the app — including derived traces, trace statistics settings, labels, reference lines, and plot and trace overrides — are preserved when a new connection starts instead of resetting. When it is off, each new connection starts from a clean configuration.

Turn **Retain** on when you have tuned a view you want to keep between sessions or across firmware uploads and turn it off when you would rather every connection start fresh.

Note: **Retain** applies to manual, app-side configuration. Settings that the firmware sends over the [viewpoint] protocol when a connection is established are still applied as usual, however, persisted app-side configuration may override the serial settings.

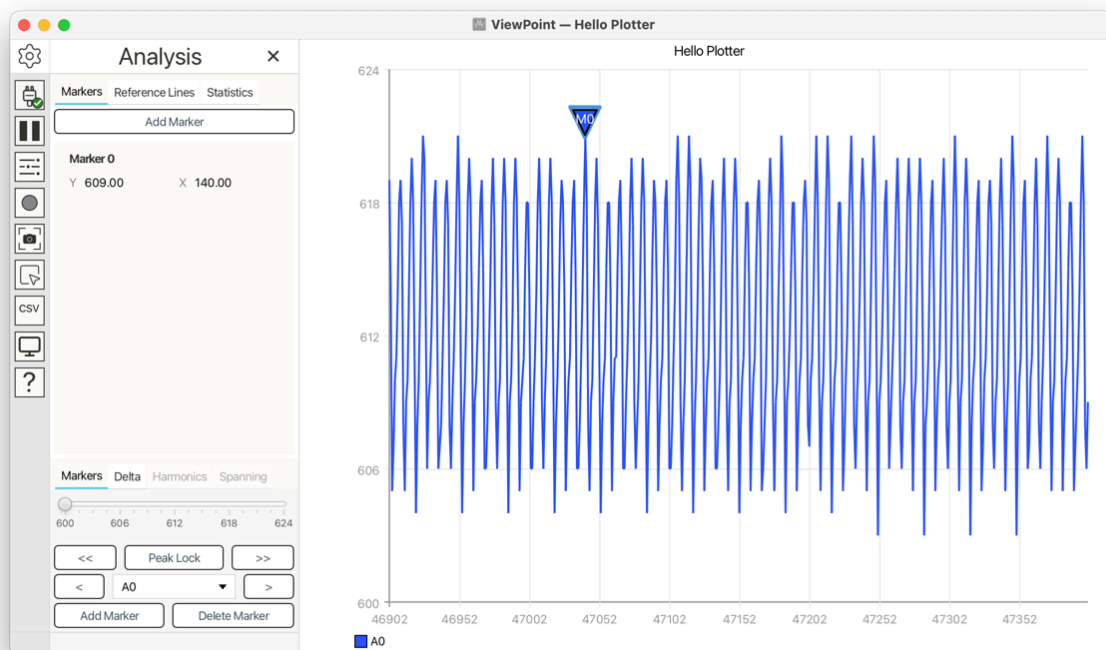
Analysis

The Analysis panel provides signal inspection tools including live markers, reference lines and bands, and trace statistics.

These tools work directly on the traces shown in the plots, making them useful for identifying peaks, measuring spacing or amplitude differences, tracking thresholds, and summarizing signal behavior while data is streaming or while paused.

Markers

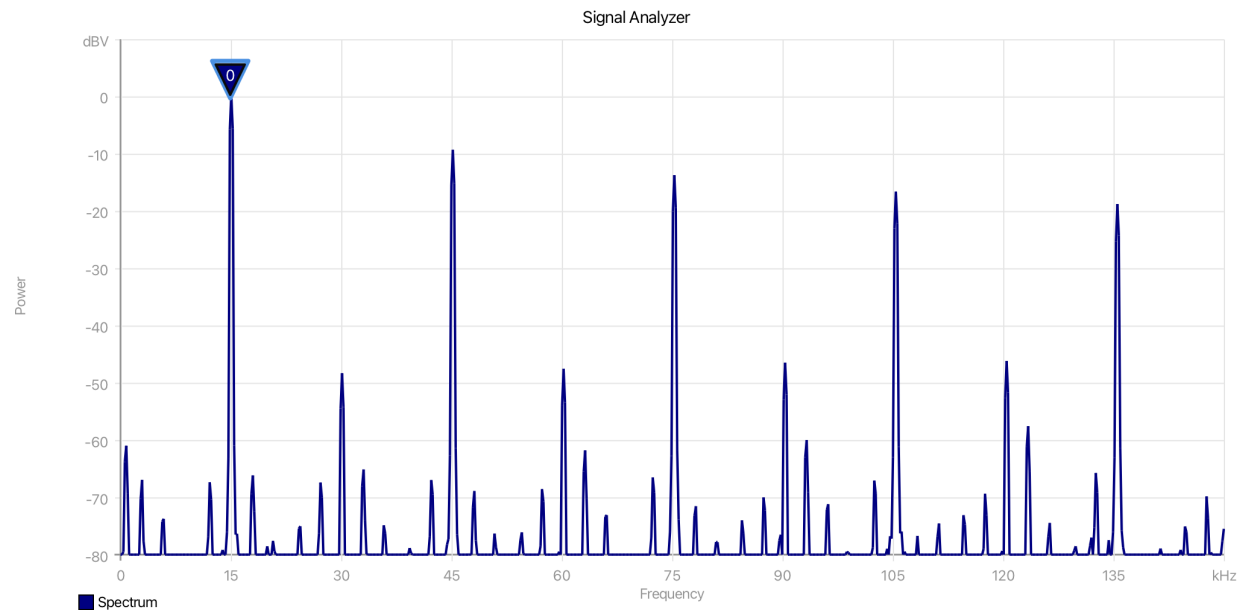
- **Standard** — place a marker at a specific point to read its value.
- **Delta** — measure the difference between two points.
- **Harmonic** — display harmonic series markers.
- **Spanning** — mark a range on the trace.
- **Peak Lock** — automatically track the peak value.



Analysis panel with markers. Marker readout and live trace inspection tools.

All marker sub-panels include similar UI for traversing peaks and traversing traces manually. This UI will be explicitly explained in the Standard Markers section below.

Standard Markers



Marker 0 Plot 1 Spectrum

Y 0.29 dBV

X 15.06 kHz

Marker 0 Plot 1 Spectrum Peak

Y 0.29 dBV

X 15.06 kHz

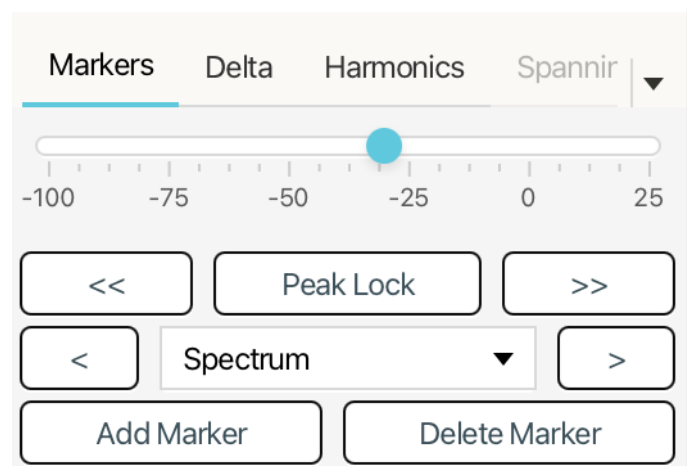
Each marker in the collection has an ID, the plot number, and trace name. For standard markers the Y value is on the left and the X value is on the right. Peak lock enabled is shown on the right.

The **Threshold** slider sets the minimum value used during peak searches. Values below the threshold are ignored.

Peak Lock locks the marker to the current peak and attempts to track that peak as it moves across the plot.

Use << and >> to search for the **previous** or **next** peak respectively.

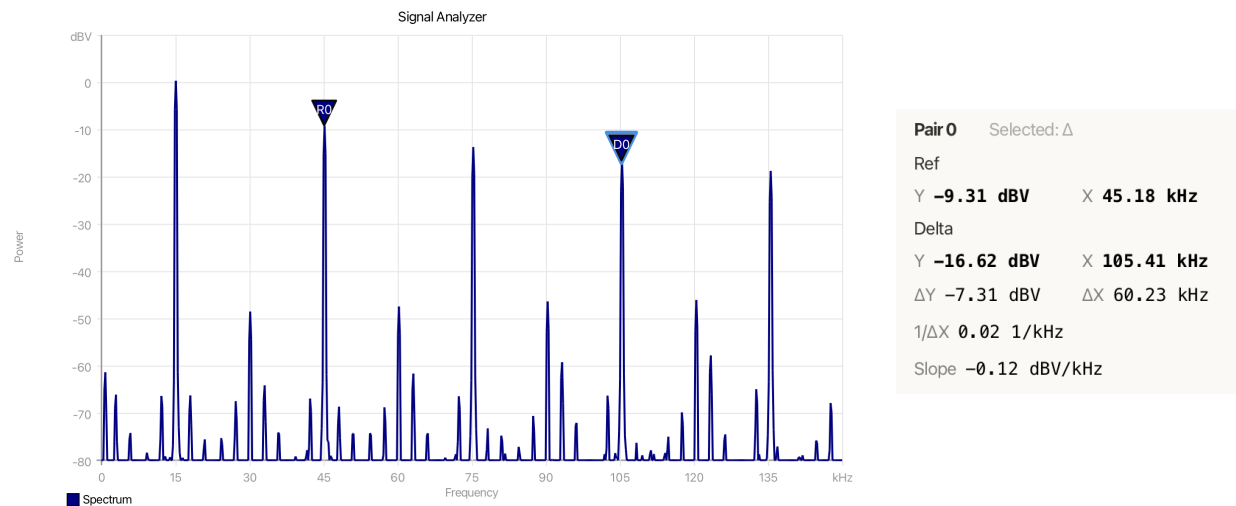
Use < and > to move the marker manually, one point at a time. Hold either button to move between points more quickly.



Use the trace dropdown to choose which trace the marker analyzes.

Add Marker creates a new marker while **Delete Marker** removes the currently selected marker.

Delta Markers



The **Delta** panel measures the difference between two linked markers: a reference marker and a delta marker.

Ref shows the position and value of the reference marker.

Delta shows the position and value of the second marker in the pair.

ΔY displays the vertical difference between the two markers, while **ΔX** displays the horizontal difference. These values are useful for comparing amplitude and frequency spacing between two points on the plot.

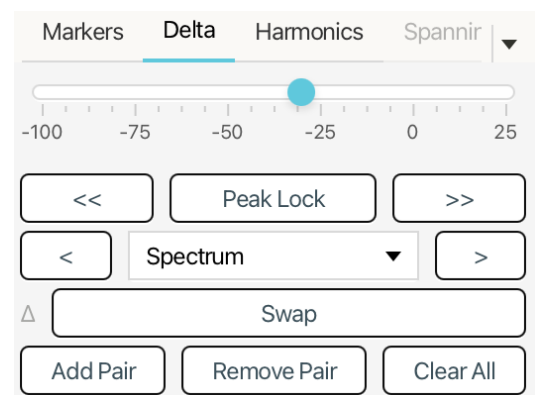
$1/\Delta X$ displays the inverse of the horizontal spacing. For frequency plots, this can be useful when relating frequency spacing to time or period.

Slope shows the rate of change between the two markers, calculated from the vertical difference over the horizontal difference.

The **Threshold** slider, **Peak Lock**, peak search buttons, manual movement buttons, and trace dropdown work the same way as in the Markers panel, but apply to the currently selected marker in the delta pair.

Use **Swap** to exchange the reference and delta markers in the selected pair.

Add Pair creates a new delta marker pair. **Remove Pair** deletes the currently selected pair. **Clear All** removes all delta marker pairs.



Harmonic Markers

Fundamental Plot 1 Spectrum

Y **0.29 dBV** X **15.06 kHz**

Harmonic 3 Plot 1 Spectrum

Y **-9.31 dBV** X **45.18 kHz**

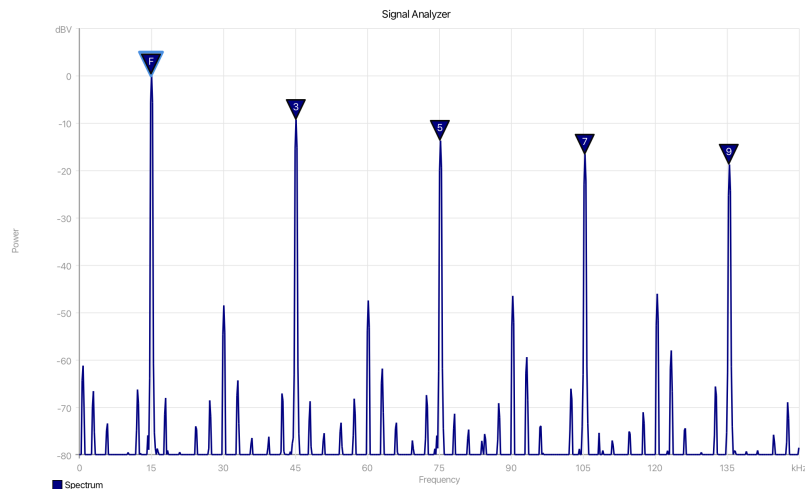
Harmonic 5 Plot 1 Spectrum

Y **-13.75 dBV** X **75.29 kHz**

Harmonic 7 Plot 1 Spectrum

Y **-16.62 dBV** X **105.41 kHz**

Harmonic 9 Plot 1 Spectrum

Y **-18.79 dBV** X **135.53 kHz**

The **Harmonics** markers finds and labels harmonics related to a selected fundamental frequency.

The **F** marker will automatically attempt to find the fundamental frequency, however, you can use the marker controls to position the fundamental marker on the frequency you want to analyze. **Peak Lock**, peak search, manual movement, and the trace dropdown work the same way as in the Markers panel.

Count sets the highest harmonic number to search for.

Filter controls which harmonics are shown in the results.

Use **Harmonics Markers** to search for harmonics based on the selected fundamental. Detected harmonics are marked on the plot and listed with their harmonic number, trace name, amplitude, and frequency.

Markers
 Delta
 Harmonics
 Spannr

-100 -75 -50 -25 0 25

<< Peak Lock >>

< Spectrum >

Count 12 Filter All

Find Harmonics Clear

The results list shows the **Fundamental** first, followed by detected harmonics such as **Harmonic 3**, **Harmonic 5**, and so on. Each entry reports the marker's **Y** value and **X** position.

Use **Clear** to remove the harmonic markers and results from the plot.

Spanning Markers



The **Spanning** panel places linked markers at the same sample index across multiple plots. This makes it easier to compare what is happening at the same point in the data across different views, such as a signal plot and its spectrum.

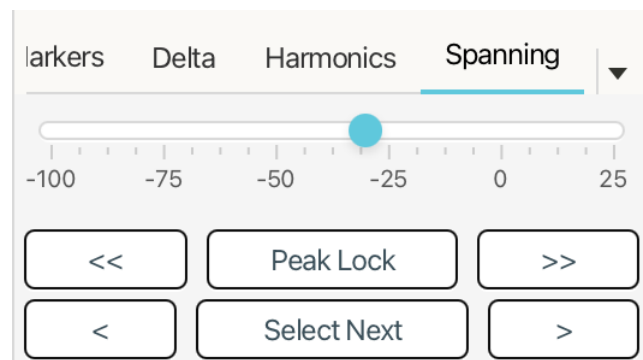
Use the slider to move the spanning marker through the shared data index.

Peak Lock locks the selected spanning marker to the current peak and attempts to follow that peak as it moves.

Use << and >> to search for the previous or next peak.

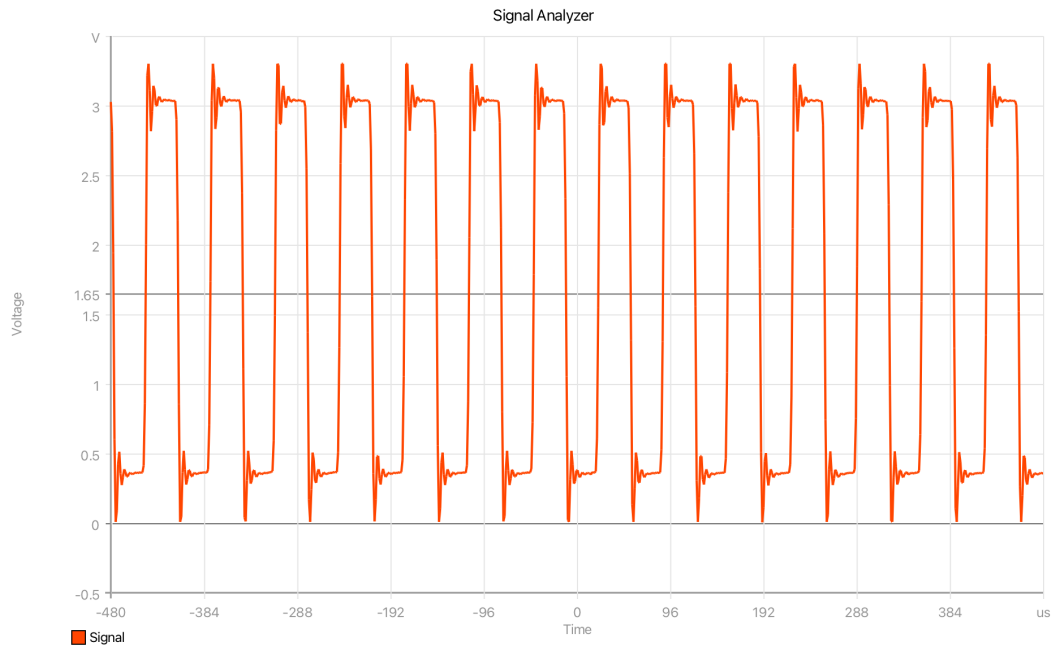
Use < and > to move the spanning marker manually, one point at a time. Hold either button to move through points more quickly.

Use **Select Next** to switch which spanning marker is currently selected.



The results list shows one entry for each plot where the spanning marker is visible. Each entry reports the marker name, plot name, trace name, **Y** value, and **X** position for that plot.

Reference Lines and Bands



The **Reference Lines** panel adds horizontal or vertical guide lines to the selected plot. Reference lines can be fixed at a value, track a live trace statistic, or change visibility based on how often a trace crosses the line.

Use **Add Line** to create a new reference line on the current plot. Use **Remove** to delete the selected line. If multiple plots are available, **Clone to All** copies the selected line to the other plots.

Axis chooses whether the line belongs to the X or Y axis. **Value** sets the line position for modes that use a fixed value. **Label** adds optional text next to the line.

Add Line

Remove

Axis

Y

Value

1.65

Label

Color

Stroke

1

Style

Solid

Enabled

☒

Static

Tracking

Crossing Feedback

Color, **Stroke**, and **Style** control the line's appearance. **Enabled** shows or hides the line without deleting it.

The list shows the reference lines for each axis. Each entry includes the line color, label, value, stroke weight, style, mode, and whether the line is enabled.

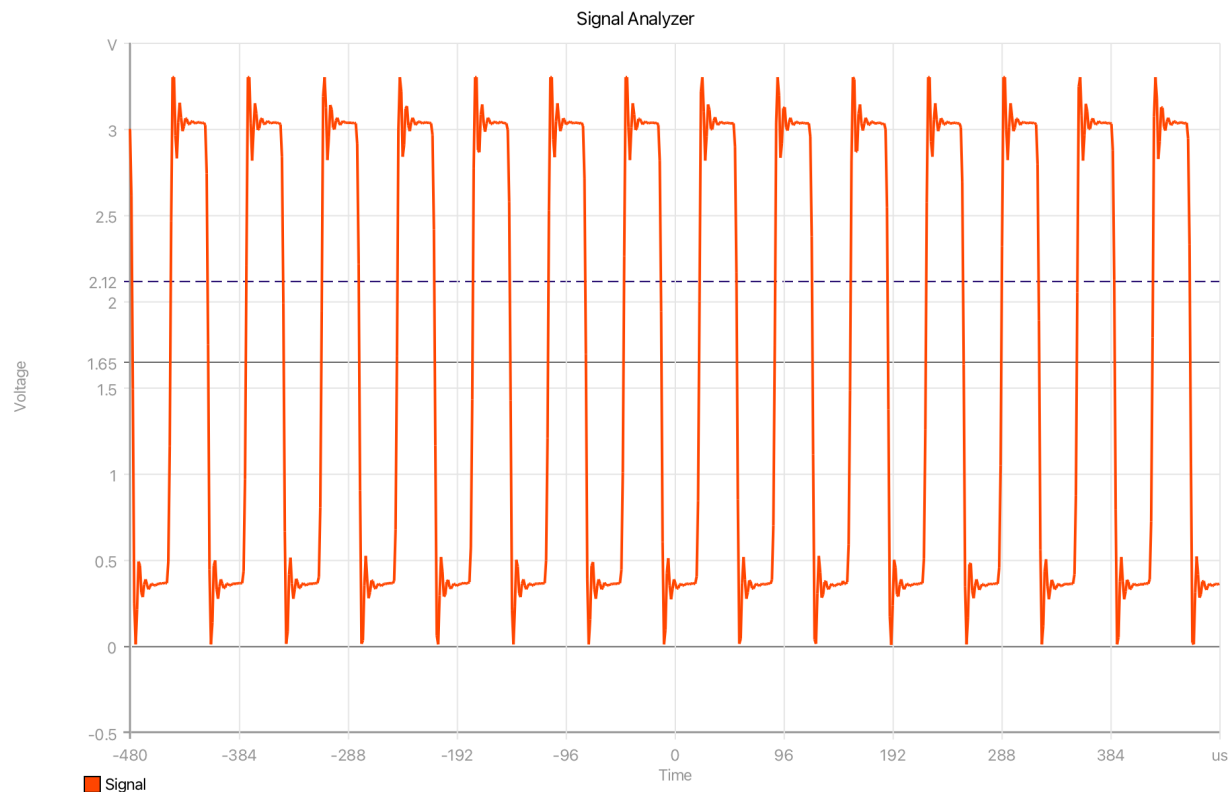
Static

Static mode places the reference line at a fixed axis value.

Use **Value** to set the exact position of the line. Static lines are useful for marking thresholds, limits, expected levels, or other constant reference points.

Y Axis	
0.00 wt: 1.0 Solid	Serial Static
1.65 wt: 1.0 Solid	Serial Static

Tracking



Tracking moves the reference line automatically based on a statistic from a selected trace.

Use **Source Trace** to choose the trace the line follows. Use **Statistic** to choose the value to track, such as minimum, maximum, mean, RMS, standard deviation, or latest value.

Tracking lines update as the plotted data changes. This is useful for showing a live RMS level, average value, peak level, or other changing measurement directly on the plot.

Zero Mean Min/Max RMS

Use the quick add menu at the top of the collection to add frequently used values.

The VoidLoop ViewPoint™ library can add and update reference lines based upon data and conditions using `updateVerticalReferenceLine` and `updateHorizontalReferenceLine` for more custom tracking.

0.00 wt:1.0 Dashed Tracking
UI Override Tracking RMS

Add Line Remove

Axis Y

Label

Color

Stroke 1

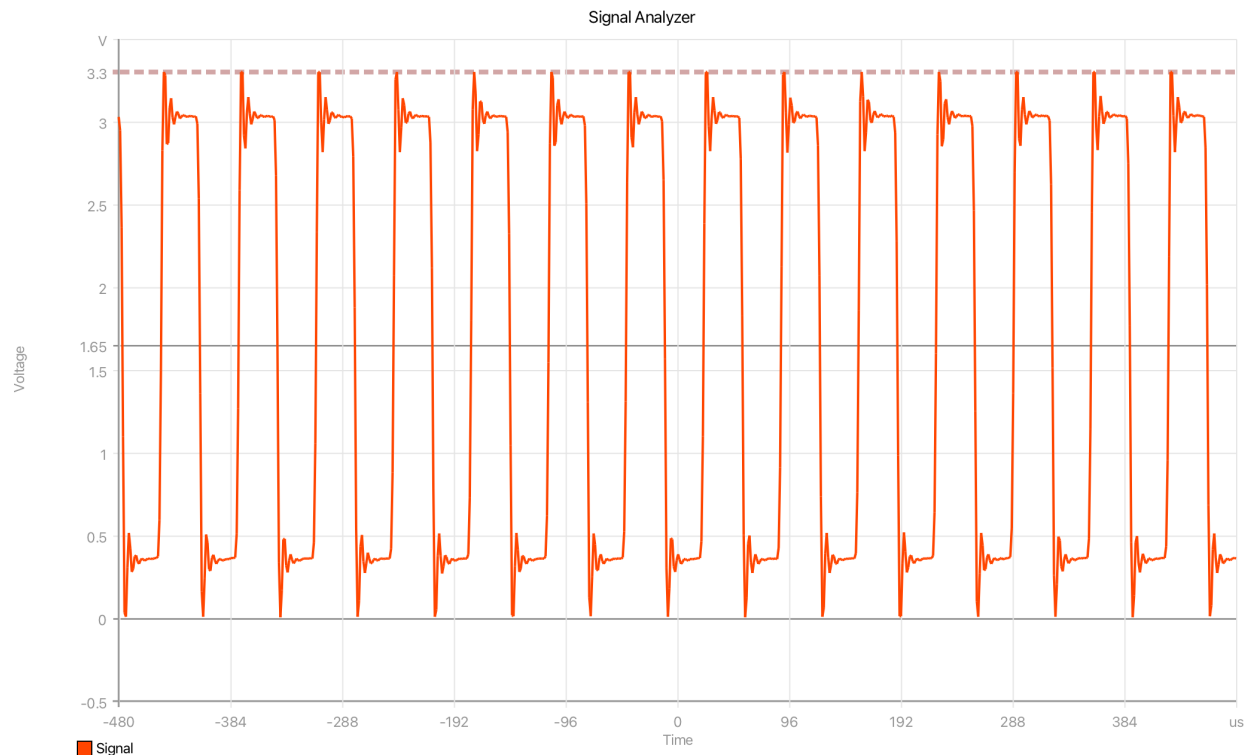
Style Dashed

Enabled

Static Tracking Crossing Feedback

Source Trace Signal

Statistic RMS

Crossing Feedback

Crossing Feedback places the line at a fixed value and changes the line opacity based on how often the selected trace crosses it.

Use **Value** to set the crossing level. Use **Source Trace** to choose which trace is analyzed.

Max Crossings sets how many crossings are needed for the line to reach maximum opacity. Lower values make the feedback become stronger with fewer crossings.

Alpha Range sets the minimum and maximum opacity used by the feedback. The line starts near the minimum opacity when there are few or no crossings, then becomes more visible as the crossing count increases.

The readout shows the current number of crossings and the computed alpha value for the selected line.

3.30 wt: 1.0 Dashed Crossing

UI Override Crossing

Axis

Y

Value

3.3

Label

Color

Stroke

4

Style

Dashed

Enabled

Static

Tracking

Crossing Feedback

Source Trace

Signal

Max Crossings

100

Alpha Range

0.1

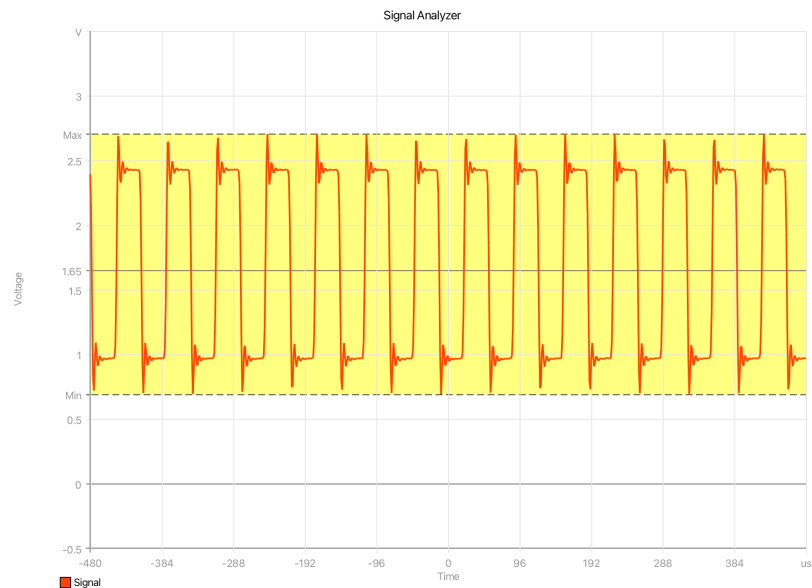
to

1

Apply

Crossings: 28, Alpha: 0.35

Reference Bands



Reference bands fill the area between two reference lines. Before creating a band, add at least two reference lines on the same axis.

In this example, two tracking reference lines were created first: one tracking the signal minimum and one tracking the signal maximum. The reference band was then created between those two lines, filling the live range between **Min** and **Max** as the tracked values update.

Use **Add Band** to create a new band from existing reference lines. Use **Remove** to delete the selected band.

Upper Line selects the reference line used as the upper edge of the band. **Lower Line** selects the reference line used as the lower edge. The selected lines define the band's boundaries.

Fill Color controls the color and transparency of the shaded region.

Enabled shows or hides the band without deleting it.

The bands list shows each band by its two boundary lines, such as **Max ↔ Min**, along with the axis it belongs to and whether it is enabled. Because bands are linked to reference lines, changing or tracking the underlying lines also changes the filled band.

Min ☒
 0.00 wt: 1.0 Dashed Tracking
 UI Override Tracking Min

Max ☒
 0.00 wt: 1.0 Dashed Tracking
 UI Override Tracking Max

Lines Bands

Add Band Remove

Upper Line Max ▼

Lower Line Min ▼

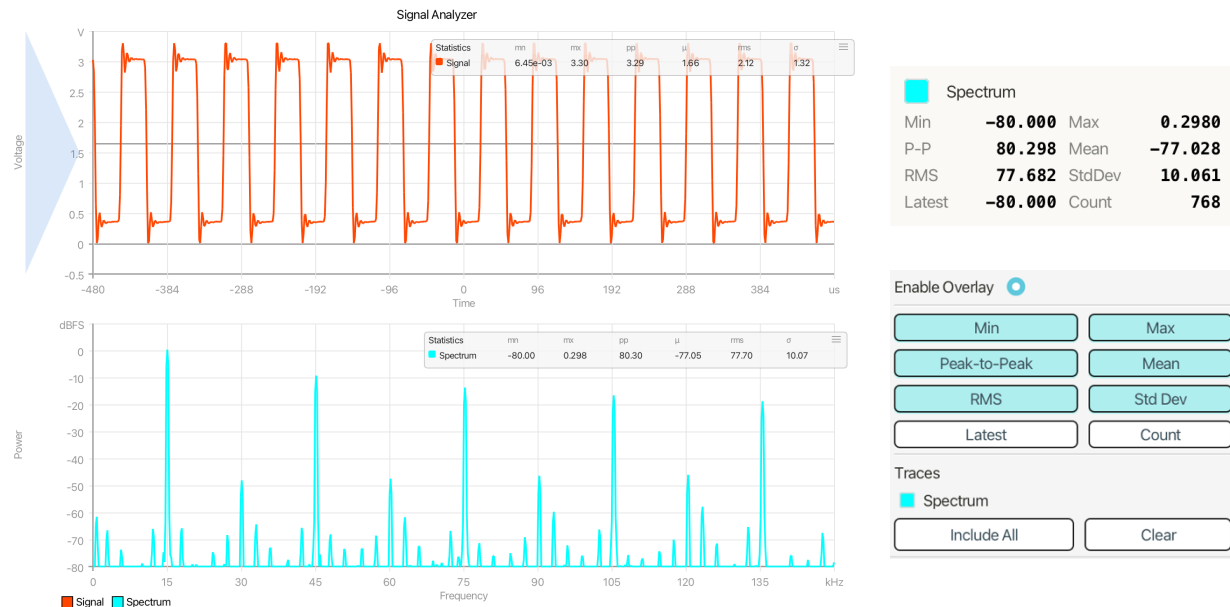
Fill Color

Enabled ☒

Y Axis

Max ↔ Min ☒
 Y Axis Enabled

Trace Statistics



The **Statistics** panel calculates live statistics for selected traces. Each selected trace is shown with its trace color, name, and current values.

The available statistics are **Min**, **Max**, **Peak-to-Peak**, **Mean**, **RMS**, **Std Dev**, **Latest**, and **Count**.

Min and **Max** show the lowest and highest values in the current data window. **Peak-to-Peak** shows the difference between the maximum and minimum values. **Mean** shows the average value, **RMS** shows the root mean square value, and **Std Dev** shows the standard deviation. **Latest** shows the most recent value, and **Count** shows the number of samples used for the calculation.

Use **Enable Overlay** to show or hide the statistics overlay on the selected plot.

The statistic buttons control which values are displayed. These selections apply across all statistics overlays, so enabling or disabling a value such as **RMS** affects every plot overlay.

The **Traces** section selects which traces are included for the current plot. If a plot contains multiple traces, you can include more than one trace in the overlay. Each included trace appears as its own row, using the trace color and name.

Use **Include All** to add every trace assigned to the current plot. Use **Clear** to remove all traces from the statistics overlay for that plot.

The overlay appears directly on the plot and shows the selected statistics for the included traces. Each plot can have its own overlay and its own selected traces, while the set of visible statistic columns is shared across all overlays.

The Arduino Library

Install

Arduino Library Manager

In the Arduino IDE, open **Tools → Manage Libraries**, search for **ViewPoint**, and click **Install**.

Manual install

Clone or download the repository into the Arduino libraries directory:

```
cd ~/Documents/Arduino/libraries
git clone https://github.com/voidloopers/ViewPoint.git
```

The libraries folder is found within the Arduino sketchbook folder, which the location of can be found within the Settings dialog. Click Arduino IDE -> Settings and look for the Sketchbook location text box. The libraries folder is not necessarily auto created - if it does not yet exist, simply create the folder and copy ViewPoint into it.

Restart the Arduino IDE so it picks up the new library.

Confirm the install by opening the **HelloPlotter** example (**File → Examples → ViewPoint → Cartesian → HelloPlotter**) and uploading it to a board.

A Typical Sketch Outline

Most sketches follow the same shape:

```
#include <ViewPoint.h>
```

```
void setup() {
  view.begin();
  // Optional configuration here
}

void loop() {
  view.addData("Signal", analogRead(A0));
  view.send();
}
```

The usual pattern is:

1. include the library,
2. call view.begin(...),
3. optionally configure the plot,
4. add data in loop(),
5. call view.send().

Concepts

A handful of terms recur through the rest of this document and through every example sketch. Each one maps to a single object or enum in the library. This section is meant to be a brief overview - see the recipes section for in depth examination with screenshots.

view

view is the global Plotter instance — the single-entry point for every sketch. Including `<ViewPoint.h>` brings it into scope. Sketches call `view.begin()`, `view.addData(...)`, `view.send()`, and the rest of the configuration API on this one object.

```
#include <ViewPoint.h>
```

```
void setup() { view.begin(); }
void loop() { view.addData("A0", analogRead(A0)); view.send(); }
```

Trace

A **trace** is a named series of float samples drawn inside a plot. A trace can carry single values (`addData("A0", value)`) for Cartesian plotting, or paired values (`addData("Orbit", x, y)`) for Scatter and Polar plotting. Traces are created on first reference by label; numeric ids can also be used. A sketch can stream multiple traces in the same `send()` cycle.

Plot

A **plot** is one panel in the plot area (canvas): a pair of axes, the traces drawn inside, an optional title, and optional grid styling. A session starts with one plot. Calling `view.setNumberOfPlots(n)` adds more. Per-plot configuration is reached through `view.plot(i)`, which returns a `Plot` exposing the same axis, title, units, display-mode, and reference-line calls available on view itself. When a sketch defines multiple traces and multiple plots without explicit mapping, the desktop app auto-maps traces to plots in index order — first trace to the top plot, last trace to the bottom plot.

Mode

A sketch streams data in one of two **modes**.

Mode	Behavior	Good for
<code>Mode::Continuous</code>	each <code>send()</code> appends one point per trace	live telemetry, scrolling sensor streams
<code>Mode::Frames</code>	each <code>send()</code> replaces the whole frame	FFTs, windowed analysis, fixed-size blocks

Continuous mode is the default. Frames mode is selected with `view.begin(frames, packetSize)` or `view.setMode(frames)`. Frames mode sends a `[frames]complete=true`

marker at the end of each packet, which the desktop app uses to align display modes like Persistence and Spectrogram.

PlotType

A **plot type** selects how the desktop app renders the data.

Plot type	Best for	Typical data shape
<code>PlotType::Cartesian</code>	time series, sampled values, general telemetry	one value per point
<code>PlotType::Scatter</code>	phase plots, paired channels, trajectories	paired values (x, y)
<code>PlotType::Polar</code>	directional, rotational, angular data	magnitude plus angle

Cartesian is the default. Scatter and Polar take paired data via `addData(label, x, y)`.

DisplayMode

A **display mode** is a rendering style layered onto a Cartesian plot in Frames mode. On any other plot type or in Continuous mode, the display mode is ignored.

Display mode	What it does
<code>DisplayMode::None</code>	normal rendering, no overlay
<code>DisplayMode::Persistence</code>	older frames remain visible with alpha blending
<code>DisplayMode::Spectrogram</code>	waterfall, time on one axis and frame index on the other
<code>DisplayMode::Gradient</code>	per-pixel hit count rendered as a color spectrum

Namespace Shortcut Macros

`viewpoint` is the library's namespace. Every enum value has a fully qualified form (`viewpoint::PlotType::Scatter`, `viewpoint::Mode::Frames`) and a short macro form for the most common values (this is provided for added convenience and approachability):

Macro	Expands to
<code>cartesian</code>	<code>viewpoint::PlotType::Cartesian</code>
<code>scatter, xy</code>	<code>viewpoint::PlotType::Scatter</code>
<code>polar</code>	<code>viewpoint::PlotType::Polar</code>
<code>continuous</code>	<code>viewpoint::Mode::Continuous</code>
<code>frames</code>	<code>viewpoint::Mode::Frames</code>
<code>persistence</code>	<code>viewpoint::DisplayMode::Persistence</code>
<code>spectrogram</code>	<code>viewpoint::DisplayMode::Spectrogram</code>
<code>gradient</code>	<code>viewpoint::DisplayMode::Gradient</code>

The Hello sketches use the macro form. Showcase sketches that pull in several enums commonly write using `namespace viewpoint;` and use the fully qualified forms instead.

If a macro name collides with another library in the sketch, opt out before the include:

```
#define USE_VIEWPOINT_WITH_NAMESPACE
#include <ViewPoint.h>
```

With the opt-out in place, only the fully qualified forms are available.

Legacy Mode

`view.setLegacyMode(true)` switches the library into a data-only stream suitable for non-ViewPoint receivers — the Arduino IDE **Serial Plotter**, third-party serial plotters, or headless CSV logging. In legacy mode the library skips the [viewpoint] handshake, omits frame-complete markers, and clamps each `send()` to 500 points so the Arduino Serial Plotter parser stays in sync.

```
void setup() {
    view.begin();
    view.setLegacyMode(true);
}
```

Without legacy mode, the library waits for the desktop app to announce its protocol version before emitting any configuration commands. Until that handshake completes, only data lines are sent — so the Arduino Serial Plotter still sees a clean CSV stream by default. Legacy mode is the right switch when the sketch is targeting Serial Plotter deliberately and wants to opt out of waiting on the handshake entirely.

Configuration and Defines

Two layers of build-time configuration tune the library's footprint and behavior: a memory profile that sets a coherent group of defaults, and individual override macros that fine-tune any single value.

Memory Profile Macros

A memory profile is one of three preset bundles. Define it before `#include <ViewPoint.h>`:

```
#define VIEWPOINT_FULL
#include <ViewPoint.h>
```

If no profile is set, the library auto-selects one from the target board. Small AVR boards (UNO R3, Nano, Leonardo, Micro) get `VIEWPOINT_MINIMAL`. Mega 2560 and UNO R4 (Renesas) get `VIEWPOINT_LITE`. Every other board falls through to `VIEWPOINT_FULL`. The library emits a `#warning` line at compile time when it auto-selects `MINIMAL` or `LITE`, so a build that picked a tight profile by accident is visible in the compiler output.

A FULL build on a Teensy 4.1 costs roughly 4–6 KB of static RAM. A MINIMAL build on an UNO R3 fits in roughly 1.5 KB of SRAM with room for typical sketch state.

Individual Override Macros

Every value the memory profile sets is also accessible as its own define. Set any of these before `#include <ViewPoint.h>` to override a single field without changing the rest of the profile.

Macro	Default (MINIMAL / LITE / FULL)	What it controls
VIEWPOINT_MAX_TRACES	2 / 4 / 16	Maximum simultaneous traces
VIEWPOINT_MAX_PLOTS	1 / 2 / 8	Maximum plot panels
VIEWPOINT_MAX_TRACES_PER_PLOT	2 / 4 / 16	Trace ids per plot's association list
VIEWPOINT_MAX_REFERENCE_LINES	2 / 4 / 8	Reference lines per axis
VIEWPOINT_MAX_DERIVED_TRACES	1 / 4 / 32	Derived-trace slots — may be removed in a future release; the desktop app's UI is the recommended path
VIEWPOINT_LABEL_SIZE	16 / 24 / 32	Bytes per trace and plot label, including null terminator
VIEWPOINT_UNITS_SIZE	8 / 12 / 16	Bytes per axis-units string
VIEWPOINT_TITLE_SIZE	16 / 32 / 64	Bytes per sketch and plot title
VIEWPOINT_DEFAULT_PACKET_SIZE	50 / 100 / 500	Default points per frame in Frames mode
VIEWPOINT_TX_BUF_SIZE	32 / 128 / 512	CommandEncoder transmit-batching buffer, necessary for certain MCU's in <i>frames</i> mode (ESP32, Arduino Giga)
USE_VIEWPOINT_WITH_NAMESPACE	(off)	Disables the namespace shortcut macros so identifiers like <code>scatter</code> and <code>frames</code> stay in the user's namespace

Override one value like this:

```
#define VIEWPOINT_MAX_TRACES 8
#include <ViewPoint.h>

void setup() { view.begin(); }
void loop() {}
```

Common Data Shapes

Pattern	Typical Use
<code>view.addData("Signal", value)</code>	single-value Cartesian trace
<code>view.addData("XY", x, y)</code>	paired-value Scatter trace
<code>view.addData("Polar", radius, angle)</code>	Polar trace using magnitude and angle
repeated <code>view.addData(...)</code> calls before <code>view.send()</code>	multi-trace update in one cycle

Minimal Starting Examples

- **HelloPlotter** — simplest Cartesian example (start here)
- **HelloScatter** — simplest Scatter example
- **HelloPolar** — simplest Polar example

MCU Compatibility

The VoidLoop ViewPoint™ library supports three memory profiles: Minimal, Lite, and Full. On current builds, small AVR boards auto-select VIEWPOINT_MINIMAL, Mega 2560 and UNO R4-class Renesas boards auto-select VIEWPOINT_LITE, and most other 32-bit boards default to VIEWPOINT_FULL unless the sketch overrides the profile before `#include <ViewPoint.h>`. Use this table as the current support snapshot for the Arduino library:

Board / Family	Suggested Profile	Core Library	VoidLoop SignalCore	Verification Status	Notes
Teensy 4.1	Full	Verified	Verified	Hardware verified	Primary reference platform and main development target
Arduino Giga R1	Full	Verified	Verified	Hardware verified	DSP path tested on hardware
ESP32 Dev Module	Full	Verified	Build-verified	Partial hardware verification	Many core features tested on hardware; DSP currently compile-verified only
Arduino UNO Q	Full	Verified	Build-verified	Partial hardware verification	Uses the Zephyr core
Raspberry Pi Pico	Full	Verified	Verified	Hardware verified	Both Arduino board profiles tested
STM32 Nucleo-F446RE	Full	Build-verified	Build-verified	Hardware unverified	Current STM32 status is based on STM32duino compile checks
AVR Uno, Nano, Leonardo, Micro	Minimal	Build-verified	Not recommended	Limited / unverified	Basic plotting compiles, but SRAM is too tight for the DSP examples

Hardware verified means the sketch was flashed and observed working in ViewPoint. Build-verified means `arduino-cli` compile succeeded, but runtime testing on the device is still pending. Community contributions are welcomed on the [forum threads](#) by board type!

Common Library Calls

These are the most important sketch-side calls. A full API reference is in Appendix A.

Call	Purpose
<code>view.begin()</code>	start with default settings
<code>view.begin(plotType, mode, packetSize)</code>	start with explicit plot type, update mode, and packet size
<code>view.setPlotType(...)</code>	choose Cartesian, Scatter, or Polar
<code>view.setMode(...)</code>	choose Continuous or Frames
<code>view.setDelay(...)</code>	set the delay for the current mode
<code>view.setPacketSize(...)</code>	control the frame or packet size
<code>view.setHorizontalRange(...)</code>	set horizontal axis range
<code>view.setVerticalRange(...)</code>	set vertical axis range
<code>view.setUnits(...)</code>	set axis units (Horizontal, Vertical)
<code>view.setAxisLabels(...)</code>	set axis label text (Horizontal, Vertical)
<code>view.setTitle(...)</code>	set the sketch title shown by the app
<code>view.setPlotTitle(...)</code>	set the plot title for the active plot
<code>view.setDisplayMode(...)</code>	choose None, Persistence, Spectrogram, or Gradient
<code>view.addData(...)</code>	buffer data for a trace
<code>view.send()</code>	send buffered data and any needed config updates
<code>view.sendInfo(...)</code>	send a message to the desktop app serial monitor

Mapping Library Calls To What You See

Sketch Call	Desktop Result
<code>view.setPlotType(...)</code>	the canvas switches to Cartesian, Scatter, or Polar behavior
<code>view.setHorizontalRange(...)</code> and <code>view.setVerticalRange(...)</code>	axes start with those ranges
<code>view.setUnits(...)</code> and <code>view.setAxisLabels(...)</code>	labels and units appear in the plot UI
<code>view.setDisplayMode(...)</code>	plot rendering changes to the selected mode
<code>view.addData(...)</code>	one or more traces appear and update live
<code>view.setNumberOfPlots(...)</code> and <code>view.plot(index)</code>	the app can show and configure multiple plots
<code>view.sendInfo(...)</code>	a message will appear in the serial monitor

Example Sketches by Use Case

This is a curated set of examples, not a full list. Use it to decide what to run first or as the starting format for your own sketches.

ViewSauce**Best for:** Broad feature tour

Best single sketch for rapidly touching many plot types, modes, and display styles. Useful when you want broad exploratory coverage instead of one focused task. Tested on Teensy 4.1 and Arduino Giga.

HelloPlotter**Best for:** First end-to-end smoke test

Minimum Cartesian sketch. Best first check that the app, port selection, and basic streaming all work. Good with a floating input, potentiometer, or simple sensor on A0.

HelloScatter**Best for:** First XY / scatter workflow

Simplest paired-value example. Confirms Scatter mode and `addData(label, x, y)` usage. Samples A0 and A1.

HelloPolar**Best for:** First polar workflow

Simplest Polar example. Good first check that magnitude-plus-angle plotting makes sense. Samples A0.

MultiSensorDashboard**Best for:** Multi-plot monitoring dashboard

Shows multiple plots with different units, scales, titles, and thresholds. Good for industrial or telemetry-style monitoring use cases.

SpectrumAnalyzer

Best for: FFT / spectrum workflow

Good entry point for Frames mode, FFT-style plotting, and testing app-side display modes. Requires a Teensy 4.1. Uses simulated or generated data rather than an external signal source. Great for testing display modes and analysis features.

SignalAnalyzer_A0

Best for: Real signal inspection with both time and frequency views

Best choice when you want to inspect a real analog input in both oscilloscope and FFT views at the same time. Best used with a function generator or other clean 0–3.3V signal source on A0. Time domain can optionally be turned off.

AliasingExplorer

Best for: Sampling theory and aliasing demo

Excellent teaching and debugging example for Nyquist, aliasing, and `sendInfo(...)`. Useful when evaluating Frames mode and dual-plot explanations.

WindowFunctionExplorer

Best for: FFT window comparison / DSP teaching

Useful for comparing how window choice changes spectral leakage, peak shape, and measurement tradeoffs. Good when testing how clearly the app supports educational DSP workflows.

VibrationAnalysisXY

Best for: Rotating machinery orbit plots

Good Scatter example for phase-space and orbit-style analysis with persistence. Useful for machinery telemetry analysis.

WindRoseChart

Best for: Directional histogram / weather-style polar data

Good Polar example for directional distributions rather than continuously connected curves. Useful for wind, heading, compass, or directional event-count data.

Recipes

These recipes show small clusters of VoidLoop ViewPoint™ library calls used together for a real visual goal. The API reference teaches syntax; these recipes teach intent: what to use when you want a recognizable plot, domain look, or display behavior.

Each recipe is built around a desktop screenshot. Configuration-only sketches still call `view.send()` in `loop()` so the configuration is transmitted to the app.

1. Labels

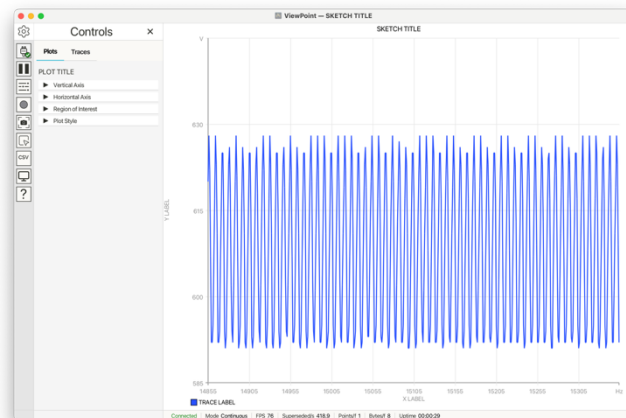
ViewPoint has separate text fields for the sketch title, plot title, axis labels, axis units, and trace labels. This recipe sets each one to a deliberately obvious value so you can match the call in the sketch to the text that appears in the desktop app. [See Plot Labels](#).

Recipe 1 - labeling on a single Cartesian plot

```
#include <ViewPoint.h>

void setup() {
  view.begin();
  view.setTitle("SKETCH TITLE");
  view.setPlotTitle("PLOT TITLE");
  view.setAxisLabels("X LABEL", "Y LABEL");
  view.setUnits("Hz", "V");
}

void loop() {
  view.addData("TRACE LABEL", analogRead(A0));
  view.send();
}
```

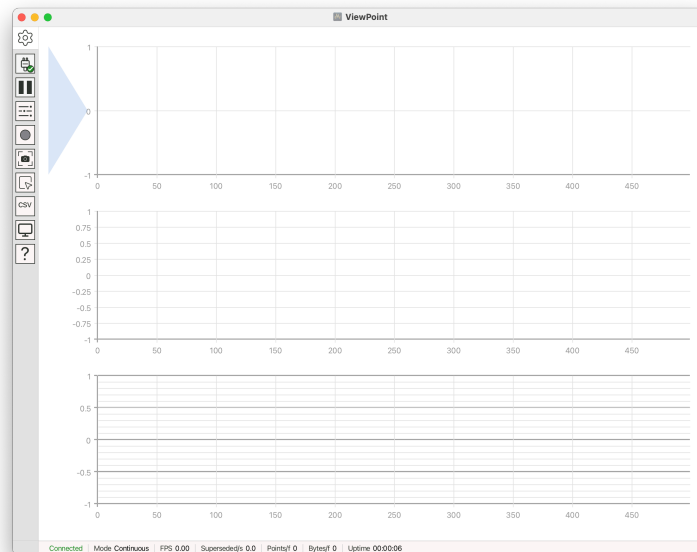


What the functions do

- **setTitle()** sets the sketch-level title, drawn above the plot region and in the desktop window title bar.
- **setPlotTitle()** sets the title inside the plot panel.
- **setAxisLabels()** sets axis text that is placed alongside each respective axis.
- **setUnits()** adds units that are used in analysis features and take the place of the max gridline label.
- **addData("TRACE LABEL", ...)** creates a trace by label; the label appears in the legend.

2. Axis ranges and grid spacing

Axis ranges can be configured three ways: bounds only with derived divisions, bounds with a specified division count, or bounds with explicit minor and major step sizes. This recipe compares the three forms so you can choose between automatic readable spacing, requesting a fixed number of divisions, or matching a domain-specific grid exactly. See [Axis Configuration](#).



Recipe 2 - three range overloads compared on stacked plots

```
#include <ViewPoint.h>

void setup() {
  view.begin();
  view.setNumberOfPlots(3);

  view.plot(0).setTitle("min, max");
  view.plot(0).setVerticalRange(-1.0, 1.0);

  view.plot(1).setTitle("min, max, divisions");
  view.plot(1).setVerticalRange(-1.0, 1.0, 8);

  view.plot(2).setTitle("min, max, minor, major");
  view.plot(2).setVerticalRange(-1.0, 1.0, 0.1, 0.5);
}

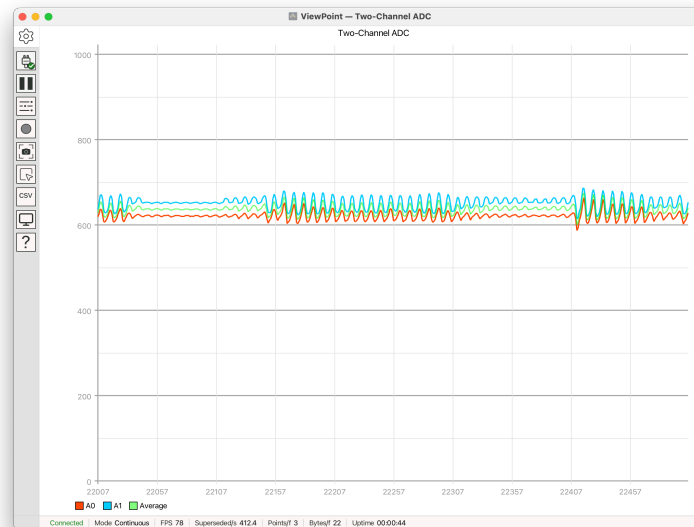
void loop() {
  view.send();
}
```

What the functions do

- **setVerticalRange(min, max)** sets bounds only; ViewPoint chooses readable grid spacing.
- **setVerticalRange(min, max, divisions)** splits the axis into a fixed number of equal segments.
- **setVerticalRange(min, max, minor, major)** sets explicit small and large grid spacing in data units.
- **setTitle(...)** Per plot titles appear in the **Controls** panel.

3. Trace colors and the legend

Trace labels control what appears in the legend, and trace colors make multi-channel plots readable at a glance. This recipe shows both color forms: named colors from the colors:: palette, raw RGB/ARGB literals, and the helper color(...) functions. See [Trace Configuration](#).



Recipe 3 - named traces with RGB and translucent ARGB colors

```
#include <ViewPoint.h>
using namespace viewpoint;

void setup() {
  view.begin();
  view.setTitle("Two-Channel ADC");
  view.setVerticalRange(0, 1023, 100, 200);
  view.trace("A0").setColor(colors::OrangeRed);
  view.trace("A1").setColor(0x00CCFF); // RGB Literal
  view.trace("Average").setColor(color(128, 0, 255, 0)); // ARGB helper
}

void loop() {
  float a0 = analogRead(A0);
  float a1 = analogRead(A1);
  view.addData("A0", a0);
  view.addData("A1", a1);
  view.addData("Average", (a0 + a1) * 0.5f);
  view.send();
}
```

What the functions do

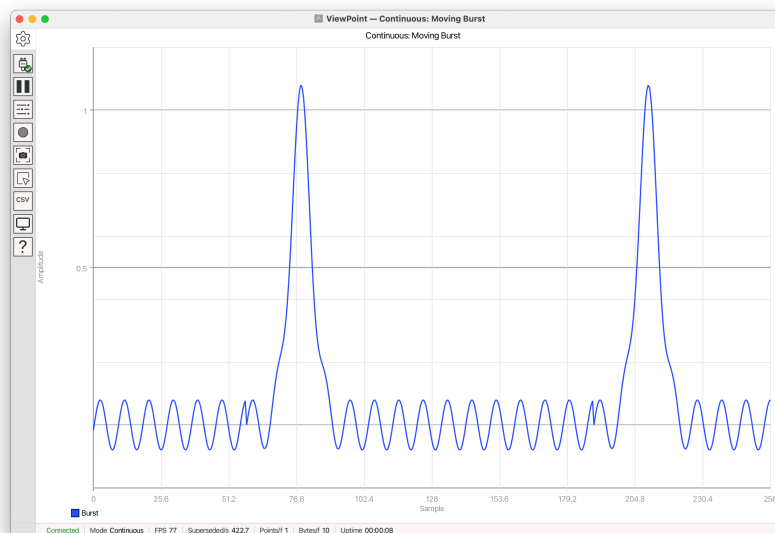
- **view.trace("A0")** gets or creates a trace by label.
- **.setColor(colors::OrangeRed)** chooses a named palette color.
- **.setColor(0x00CCFF)** chooses a raw RGB color in 0xRRGGBB form. Fully opaque alpha is assumed.
- **.setColor(color(128, 0, 255, 0))** uses the ARGB helper form: alpha, red, green, blue. Alpha values run from 0 transparent to 255 opaque.
- Trace creation order controls legend order and default automapping to plots.

4. Continuous vs Frames Modes

Continuous mode appends samples to a scrolling history, while Frames mode replaces the visible packet with the latest complete frame. In ViewPoint™, continuous draws new data with each point, while frames draws only after the full data set has been received – defining the frame rate.

This recipe uses the same burst-shaped data in both sketches. The burst center stays fixed, so the screenshots emphasize update behavior rather than motion.

Continuous mode



Recipe 4A - Continuous mode draws point by point and does NOT define FPS

```
#include <ViewPoint.h>
#include <math.h>

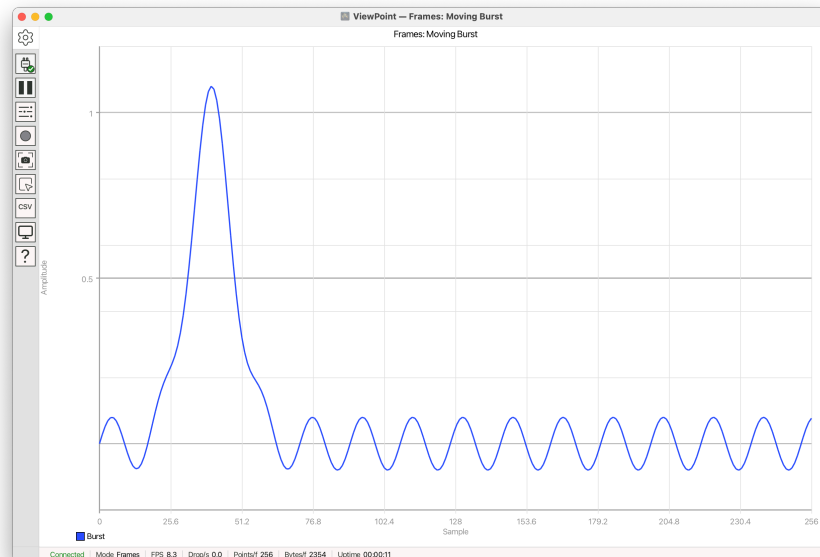
const int PACKET_SIZE = 256;

float burstCenter = 40.0f;

float burstValue(int i, float center) {
    float d = i - center;
    float pulse = expf(-(d * d) / (2.0f * 8.0f * 8.0f));
    float ripple = 0.08f * sinf(i * 0.35f);
    return pulse + ripple;
}

void setup() {
    view.begin();
    view.setDelay(2);
    view.setTitle("Continuous: point-by-point drawing");
    view.setPlotTitle("Moving Burst");
    view.setHorizontalRange(0, PACKET_SIZE);
    view.setVerticalRange(-0.2, 1.2, 0.2, 0.5);
    view.setAxisLabels("Sample", "Amplitude");
}

void loop() {
    for (int i = 0; i < PACKET_SIZE; i++) {
        view.addData("Burst", burstValue(i, burstCenter));
        view.send();
    }
}
```

Recipe 4B - Frames mode replaces the complete packet and defines the frame rate

```
#include <ViewPoint.h>
#include <math.h>

const int PACKET_SIZE = 256;

float burstCenter = 40.0f;

float burstValue(int i, float center) {
    float d = i - center;
    float pulse = expf(-(d * d) / (2.0f * 8.0f * 8.0f));
    float ripple = 0.08f * sinf(i * 0.35f);
    return pulse + ripple;
}

void setup() {
    view.begin(frames, PACKET_SIZE);
    view.setDelay(120);
    view.setTitle("Frames: full-packet replacement");
    view.setPlotTitle("Moving Burst");
    view.setHorizontalRange(0, PACKET_SIZE);
    view.setVerticalRange(-0.2, 1.2, 0.2, 0.5);
    view.setAxisLabels("Sample", "Amplitude");
}

void loop() {
    for (int i = 0; i < PACKET_SIZE; i++) {
        view.addData("Burst", burstValue(i, burstCenter));
    }

    view.send();
}
```

What changes

- **Continuous:** `view.send()` is called after every point. VoidLoop ViewPoint™ receives and draws the burst one sample at a time, so the plot behaves like a scrolling stream.

- **Frames:** `view.send()` is called once after all `PACKET_SIZE` samples have been added. VoidLoop ViewPoint™ receives a complete packet and replaces the previous frame in one update.
- **Same data, different timing:** both sketches generate the same burst samples. The visible difference comes from when `send()` is called and which update mode was selected in `view.begin(...)`.

When to use this

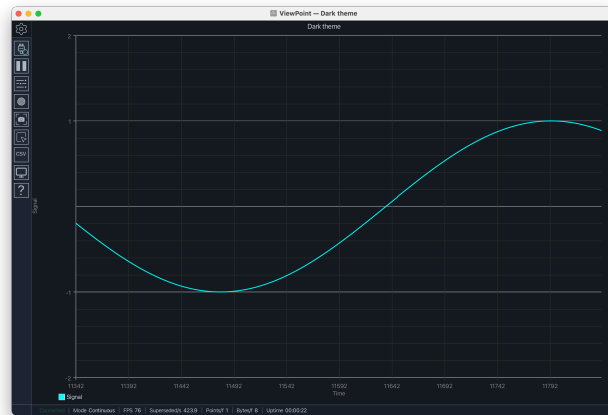
- Use Continuous for telemetry, sensor streams, and values that are meaningful as history.
- Use Frames for FFTs, oscilloscope captures, histograms, display modes, and data where packet boundaries are meaningful.

See also

- [examples/Cartesian/Quadrature/Quadrature.ino](#)

5. Custom grid colors and themes

Grid colors are part of the plot's meaning, not just decoration. This recipe shows how to set label, minor-grid, and major-grid colors so a plot can match a dark lab environment, domain convention, dashboard theme, or presentation capture. See [Plot Styles](#).



Recipe 5 - dark-theme grid with pale labels

```
#include <ViewPoint.h>
using namespace viewpoint;

void setup() {
  view.begin();
  view.setTitle("Dark Theme");
  view.setVerticalRange(-2.0, 2.0, 0.2, 1.0);
  view.setAxisLabels("Time", "Signal");
  view.setGridColors(colors::LightGray, 0x2A2A2A, 0x4A4A4A);
  view.trace("Signal").setColor(colors::Cyan);
}

void loop() {
  view.addData("Signal", sin(millis() / 200.0f));
  view.send();
}
```

What the functions do

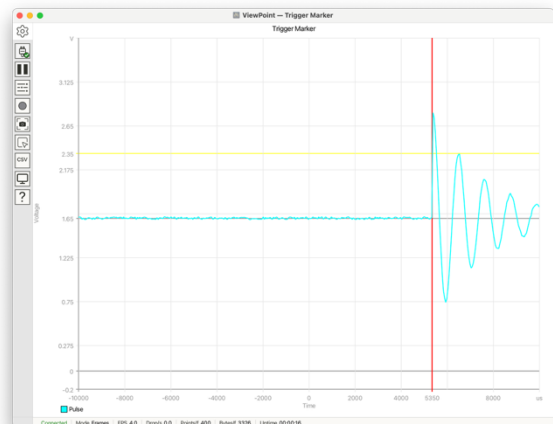
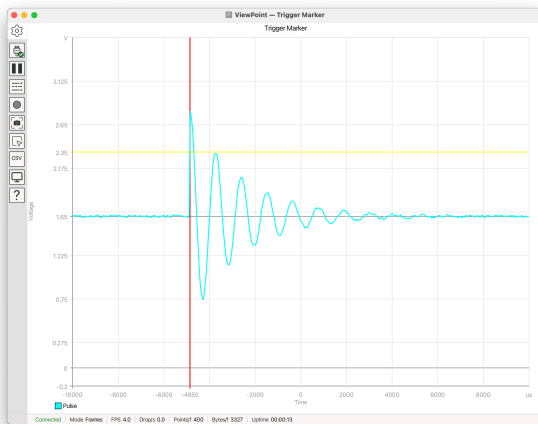
- **setGridColors(labels, minor, major)** sets axis-label color, minor gridline color, and major gridline color.
- The two-argument overload, **setGridColors(minor, major)**, changes gridlines without changing label color.

6. Reference lines: trigger threshold and event marker

Reference lines can mark static values, event positions, or moving cursors. This recipe combines horizontal threshold lines with vertical event markers, including an updated marker that moves in `loop()` by calling `updateVerticalReferenceLine()`.

A synthetic single-supply pulse crosses a trigger threshold, then rings down toward mid-rail. Horizontal reference lines mark ground, mid-rail, and the trigger level; a vertical marker moves to the trigger time for each frame.

See [Reference Lines](#) for managing reference lines and bands via the [Analysis](#) panel.



Declarations and helpers

```
#include <ViewPoint.h>
#include <math.h>
using namespace viewpoint;

const float VREF = 3.3f;
const float TRIGGER_LEVEL = 2.35f;
const uint32_t SAMPLE_RATE = 20000; // 20 kSPS
const int DISPLAY_SAMPLES = 400;
const uint32_t FRAME_DELAY_MS = 250;

int triggerMarker = -1;
int triggerIndex = 120;
int triggerStep = 17;
unsigned long lastFrameMs = 0;

float sampleAt(int i, int trigger) {
    float baseline = VREF / 2.0f;
    float noise = random(-15, 16) * 0.001f;
    float t = i - trigger;

    if (t < 0) {
        return baseline + noise;
    }

    float pulse = 1.15f * expf(-t / 45.0f) * cosf(t * 0.28f);
    return baseline + pulse + noise;
}
```

Configuration

```
void setup() {
    view.begin(frames, DISPLAY_SAMPLES);
}
```

```

view.setDelay(0);
view.setTitle("Trigger Marker");

float time_span_us = DISPLAY_SAMPLES * (1.0f / SAMPLE_RATE) * 1e6f;
float half_span = time_span_us / 2.0f;

view.setPlotTitle("Triggered Pulse");
view.setHorizontalRange(-half_span, half_span, 10);
view.setVerticalRange(-0.2, 3.6, 8);
view.setAxisLabels("Time", "Voltage");
view.setUnits("us", "V");

view.addHorizontalReferenceLine(0.0f, colors::DimGray);
view.addHorizontalReferenceLine(VREF / 2.0f, colors::DimGray);
view.addHorizontalReferenceLine(TRIGGER_LEVEL, colors::Yellow, 1.5f);

float triggerTimeUs = (triggerIndex - DISPLAY_SAMPLES / 2) *
    (1.0f / SAMPLE_RATE) * 1e6f;
triggerMarker = view.addVerticalReferenceLine(triggerTimeUs,
    colors::Red,
    2.0f);

view.trace("Pulse").setColor(colors::Cyan);
}

```

Loop

```

void loop() {
    unsigned long now = millis();
    if (now - lastFrameMs < FRAME_DELAY_MS) return;
    lastFrameMs = now;

    float triggerTimeUs = (triggerIndex - DISPLAY_SAMPLES / 2) *
        (1.0f / SAMPLE_RATE) * 1e6f;
    view.updateVerticalReferenceLine(triggerMarker, triggerTimeUs);

    for (int i = 0; i < DISPLAY_SAMPLES; i++) {
        view.addData("Pulse", sampleAt(i, triggerIndex));
    }
    view.send();

    triggerIndex += triggerStep;
    if (triggerIndex > DISPLAY_SAMPLES - 80 || triggerIndex < 80) {
        triggerStep = -triggerStep;
        triggerIndex += triggerStep;
    }
}

```

What's happening

- **Horizontal lines** identify meaningful voltage levels: ground, mid-rail, and the trigger threshold.
- **The vertical marker** identifies when the event occurred in the captured frame.
- **updateVerticalReferenceLine()** keeps the marker tied to the event as the synthetic trigger position moves from frame to frame.

What the functions do

- **addHorizontalReferenceLine(value, color, stroke)** draws a flat threshold at a y value.
- **addVerticalReferenceLine(value, color, stroke)** draws an event marker at an x value and returns its reference-line index.
- **updateVerticalReferenceLine(index, value)** moves an existing marker without creating a new line.

7. Oscilloscope-style centered view

An oscilloscope-style view needs a stable time window, predictable sample timing, and reference lines that make the input range easy to read. This recipe configures a single time-domain plot in microseconds with visible ground and mid-rail references.

```
#include <ViewPoint.h>
using namespace viewpoint;

const float VREF = 3.3f;
const uint32_t SAMPLE_RATE = 10000; // 10 kSPS
const int DISPLAY_SAMPLES = 500;
const uint32_t SAMPLE_INTERVAL_US =
    1000000UL / SAMPLE_RATE;

uint32_t lastSampleUs = 0;

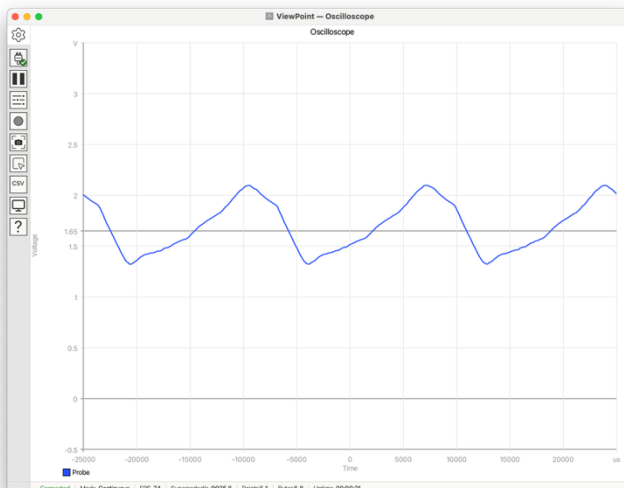
void setup() {
    view.begin();
    view.setDelay(0);
    view.setTitle("Oscilloscope");

    float time_span_us = DISPLAY_SAMPLES * (1.0f /
        SAMPLE_RATE) * 1e6f;
    float half_span = time_span_us / 2.0f;

    view.setPlotTitle("Oscilloscope");
    view.setVerticalRange(-0.5, 3.5, 8);
    view.setHorizontalRange(-half_span, half_span, 10);
    view.setAxisLabels("Time", "Voltage");
    view.setUnits("us", "V");
    view.addHorizontalReferenceLine(VREF / 2.0f, colors::DimGray);
    view.addHorizontalReferenceLine(0.0f, colors::DimGray);
}

void loop() {
    uint32_t now = micros();
    if (now - lastSampleUs < SAMPLE_INTERVAL_US) return;
    lastSampleUs += SAMPLE_INTERVAL_US;

    float volts = analogRead(A0) * VREF / 1023.0f;
    view.addData("Probe", volts);
    view.send();
}
```



What the functions do

- **setDelay(0)** disables the library's built-in delay so the micros() gate controls sample cadence.
- **setHorizontalRange(-half_span, half_span, 10)** gives the time-domain view a stable left/right span around zero.
- **setVerticalRange(-0.5, 3.5, 8)** frames a 0-3.3 V ADC signal with headroom above and below the rails.
- **Reference lines at VREF / 2.0f and 0.0f** mark mid-rail and ground, common scope landmarks for single-supply signals.

See also

- [examples/Cartesian/SignalAnalyzer_A0/SignalAnalyzer_A0.ino](#)

8. Electrocardiogram-style strip chart

An ECG-style strip chart is a good example of domain-specific plot configuration: the grid spacing, colors, units, baseline, and scroll rate all work together. This recipe shows how those calls combine to produce a recognizable medical-paper display. See [Plot Styles](#).

Recipe 8 - pink-grid ECG strip chart with a baseline

```
#include <ViewPoint.h>
#include <math.h>

float phase = 0.0f;

float ecgSample(float p) {
    if (p < 0.10f) return 0.15f * sinf((p / 0.10f) * PI);
    if (p < 0.16f) return 0.0f;
    if (p < 0.18f) return -0.10f * sinf(((p - 0.16f) / 0.02f) * PI);
    if (p < 0.22f) return 1.20f * sinf(((p - 0.18f) / 0.04f) * PI);
    if (p < 0.24f) return -0.20f * sinf(((p - 0.22f) / 0.02f) * PI);
    if (p < 0.40f) return 0.05f;
    if (p < 0.56f) return 0.30f * sinf(((p - 0.40f) / 0.16f) * PI);
    return 0.0f;
}

void setup() {
    view.begin();
    view.setDelay(4);
    view.setTitle("Heartbeat ECG");
    view.setPlotTitle("Lead II");
    view.setAxisLabels("Time", "Voltage");
    view.setUnits("s", "mV");
    view.setVerticalRange(-0.5, 1.5, 0.1, 0.5);
    view.setGridColors(0xFFCCCC, 0xFF6666);
    view.addHorizontalReferenceLine(0.0f);
    view.trace("Lead II").setColor(0x00AA00);
}

void loop() {
    view.addData("Lead II", ecgSample(phase));
    view.send();
    phase += 0.0048f;
    if (phase >= 1.0f) phase -= 1.0f;
}
```



What the functions do

- `setVerticalRange(-0.5, 1.5, 0.1, 0.5)` creates the small and large divisions expected on ECG plots.
- `setGridColors(0xFFCCCC, 0xFF6666)` gives the plot the medical paper look.
- `addHorizontalReferenceLine(0.0f)` marks the isoelectric baseline.

When to use this

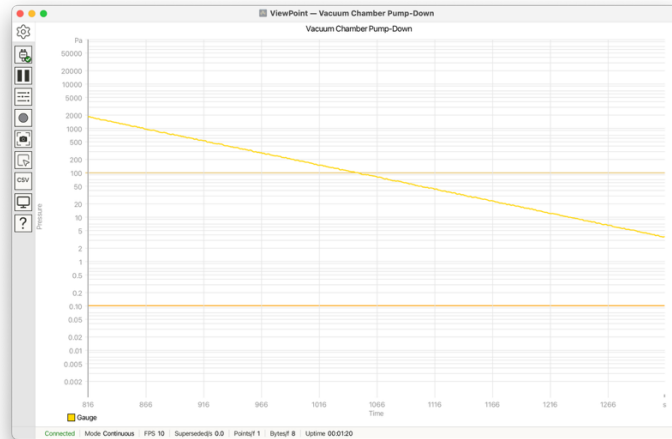
- Adjust the vertical range when your signal is not in millivolts.
- Change `setDelay()` and the phase increment together if you want a faster or slower sweep.

See also

- `examples/Cartesian/HeartbeatECG/HeartbeatECG.ino`

9. Large-dynamic-range sensor on a log axis

Some sensors produce values that span several orders of magnitude, where a linear axis hides the low end or lets the high end dominate. This recipe shows a logarithmic vertical axis with positive data, fixed decade bounds, and reference lines that remain readable across the full range. See [Axis Configuration - Scale](#).



```
#include <ViewPoint.h>
using namespace viewpoint;

void setup() {
  view.begin();
  View.setDelay(100);
  view.setTitle("Vacuum Chamber Pump-Down");
  view.setPlotTitle("Chamber Pressure");
  view.setVerticalRange(1e-3, 1e5);
  view.enableLogarithmicScale();
  view.setAxisLabels("Time", "Pressure");
  view.setUnits("s", "Pa");
  view.addHorizontalReferenceLine(100, colors::Goldenrod, 1.0f);
  view.addHorizontalReferenceLine(0.1, colors::Orange, 1.5f);
  view.trace("Guage").setColor(colors::Gold);
}

void loop() {
  // Note: full simulation available at docs.voidloop.com
  view.addData("Guage", lux);
  view.send();
}
```

What the functions do

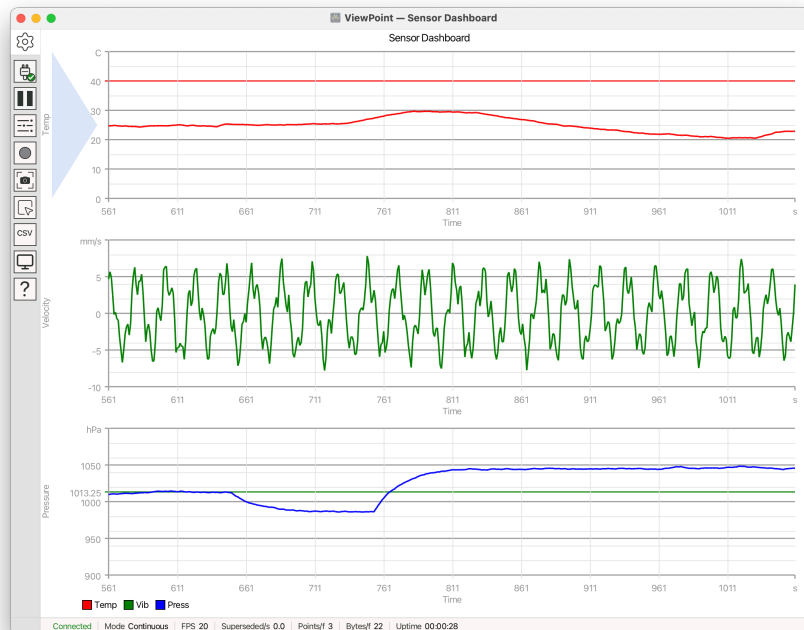
- **enableLogarithmicScale()** maps positive linear values onto a log vertical axis.
- **setVerticalRange(1, 100000)** spans five decades; the minimum must be greater than zero.
- **Reference lines** keep meaningful environmental levels visible across the compressed scale.

When to use this

- Use **enableLogarithmicScale(false)** when the data is already in log space, such as dB.
- Clamp or filter zero/negative values before plotting on a log axis.

10. Multi-plot stacked monitor

Multi-plot layouts let unrelated signals share one session without forcing them onto the same scale. This recipe builds a stacked dashboard where each plot has its own title, range, units, reference lines, and trace color.



Recipe 10 - three-plot stacked sensor dashboard

Declarations and helpers

```
#include <ViewPoint.h>
using namespace viewpoint;

float temperature = 25.0f;
float tempTarget = 25.0f;
float vibPhase = 0.0f;
float pressure = 1013.0f;
float pressureTarget = 1013.0f;
```

Configuration

```
void setup() {
    view.begin();
    view.setDelay(50);
    view.setTitle("Sensor Dashboard");
    view.setNumberOfPlots(3);

    view.plot(0).setTitle("Temperature");
    view.plot(0).setVerticalRange(0, 50, 5, 10);
    view.plot(0).setAxisLabels("Time", "Temp");
    view.plot(0).setUnits("s", "C");
    view.plot(0).addHorizontalReferenceLine(40.0f, colors::Red, 1.5f);

    view.plot(1).setTitle("Vibration");
    view.plot(1).setVerticalRange(-10, 10, 2, 5);
    view.plot(1).setAxisLabels("Time", "Velocity");
    view.plot(1).setUnits("s", "mm/s");
```

```

view.plot(2).setTitle("Pressure");
view.plot(2).setVerticalRange(900, 1100, 20, 50);
view.plot(2).setAxisLabels("Time", "Pressure");
view.plot(2).setUnits("s", "hPa");
view.plot(2).addHorizontalReferenceLine(1013.25f, colors::Green, 1.5f);

view.trace("Temp").setColor(colors::Red);
view.trace("Vib").setColor(colors::Green);
view.trace("Press").setColor(colors::Blue);
}

```

Loop

```

void loop() {
    if (random(100) < 2) tempTarget = 20.0f + random(20);
    temperature += (tempTarget - temperature) * 0.01f;
    temperature += random(-10, 10) / 100.0f;

    vibPhase += 0.3f;
    float vib = 5.0f * sin(vibPhase) + 2.0f * sin(vibPhase * 3.7f);
    vib += random(-10, 10) / 10.0f;

    if (random(100) < 1) pressureTarget = 950.0f + random(100);
    pressure += (pressureTarget - pressure) * 0.05f;
    pressure += random(-5, 5) / 10.0f;

    view.addData("Temp", temperature);
    view.addData("Vib", vib);
    view.addData("Press", pressure);
    view.send();
}

```

What the functions do

- **setNumberOfPlots(3)** creates three stacked plot panels.
- **view.plot(i)** configures one panel's title, axis range, units, labels, and reference lines.
- **Trace auto-map** pairs traces with plots in creation order when no explicit assignment is made.

When to use this

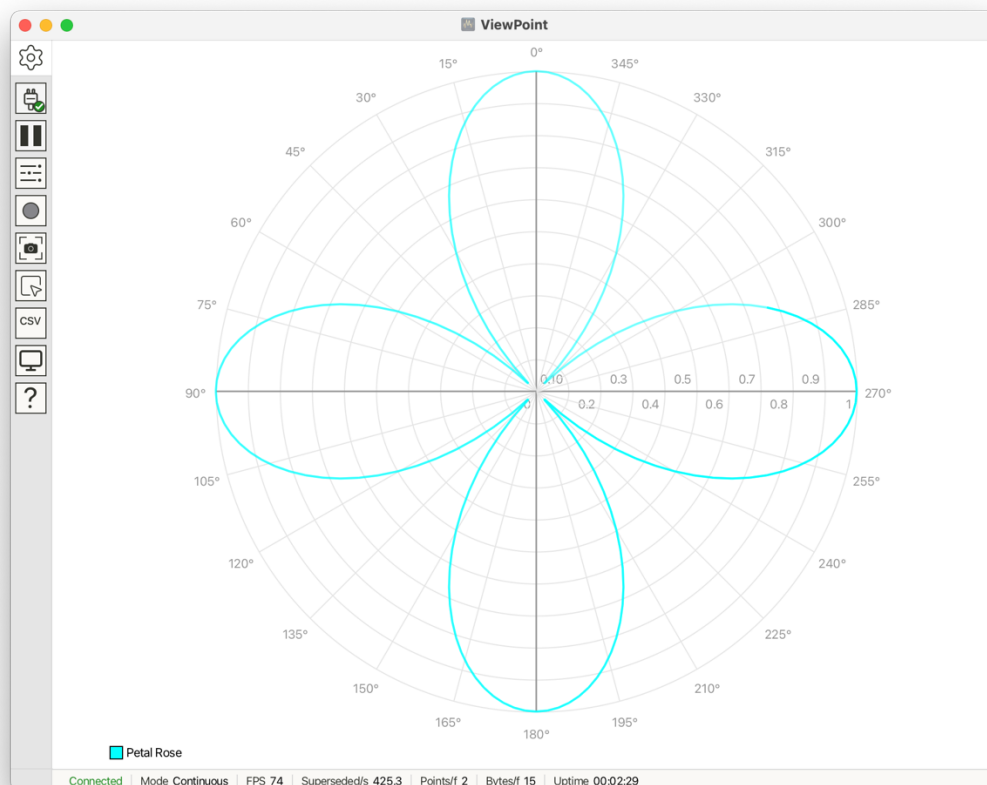
- Call **view.plot(i).addTrace(traceId)** when auto-map is not the desired trace-to-plot assignment.
- Use separate scales whenever signals have different units or ranges.

See [examples/Cartesian/MultiSensorDashboard/MultiSensorDashboard.ino](#)

11. Polar angular conventions

Polar plots are only useful when the angular convention matches the problem. This recipe compares the two common conventions side by side: compass-style degrees with 0 at the top, and mathematical radians with 0 at the right. See [Polar Properties](#).

Compass convention: degrees, 0 at top



Recipe 11A - compass-convention polar plot, 0 degrees at top

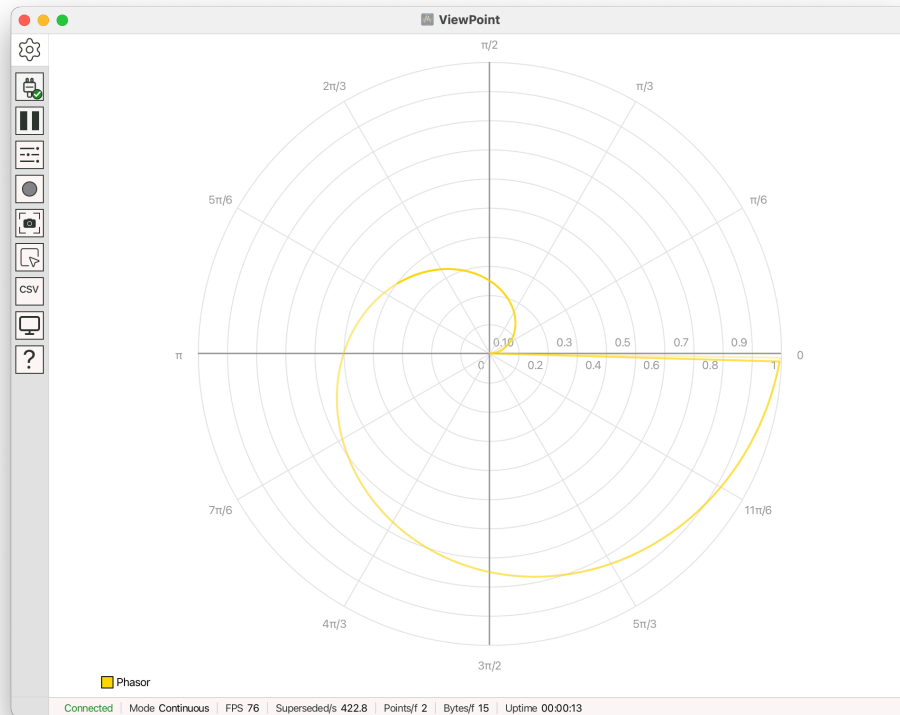
```
#include <ViewPoint.h>
using namespace viewpoint;

float theta = 0.0f;

void setup() {
  view.begin(polar);
  view.setAngularUnits(AngularUnit::Degrees);
  view.setAngularOffset(90);
  view.setAngularStep(15);
  view.setRadialRange(0, 1);
  view.trace("Petal Rose").setColor(colors::Cyan);
}

void loop() {
  view.addData("Petal Rose", fabs(cos(2 * theta * DEG_TO_RAD)), theta);
  view.send();
  theta += 2.0f;
  if (theta >= 360.0f) theta -= 360.0f;
}
```

Math convention: radians, 0 at right



Recipe 11B - math-convention polar plot, 0 radians at right

```
#include <ViewPoint.h>
using namespace viewpoint;

float theta = 0.0f;

void setup() {
  view.begin(polar);
  view.setRadians();
  view.setAngularStep(30);
  view.setRadialRange(0, 1);
  view.trace("Phasor").setColor(colors::Gold);
}

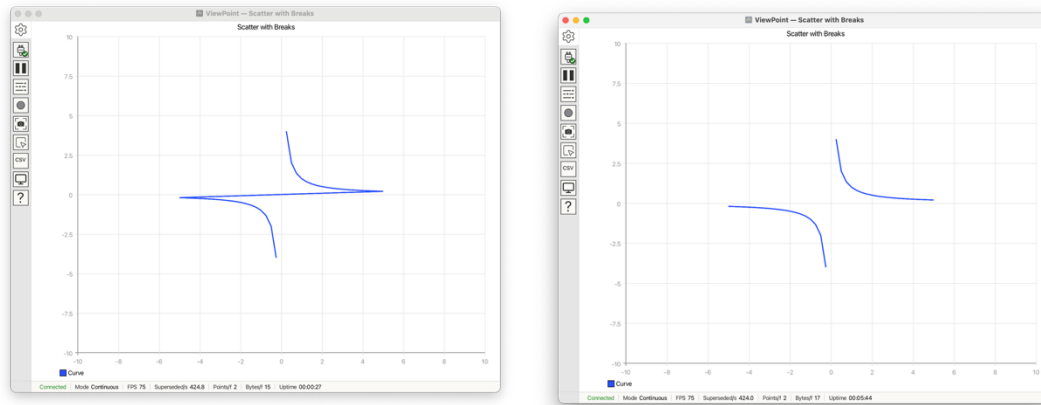
void loop() {
  view.addData("Phasor", theta / TWO_PI, theta);
  view.send();
  theta += 0.03f;
  if (theta >= TWO_PI) theta -= TWO_PI;
}
```

What the functions do

- **setAngularOffset(90)** rotates the compass plot so 0 degrees is at the top.
- **setRadians()** switches incoming angle values and angular offset to radians.
- **setAngularStep()** controls grid spacing. It currently accepts whole degrees even when the angular unit is radians.
- **setRadialRange(0, 1)** creates a unit-radius plot suitable for a normalized bearing or phasor.

12. Scatter with breaks

Scatter and Polar traces normally connect consecutive points. This recipe shows how `addBreak()` inserts an intentional discontinuity so a single trace can draw separate strokes, paths, or phasor segments without unwanted bridge lines.



Recipe 12 – Left shows the sketch without `addBreak` while right shows the sketch with `addBreak`

```
#include <ViewPoint.h>

void setup() {
  view.begin(scatter, frames, 40);
  view.setTitle("Scatter with Breaks");
  view.setHorizontalRange(-10, 10);
  view.setVerticalRange(-10, 10);
}

void loop() {
  view.addBreak("Path");
  for (float x = -5.0f; x <= 5.0f; x += 0.25) {
    view.addData("Curve", 1.0f / x);
  }
  view.send();
}
```

What the functions do

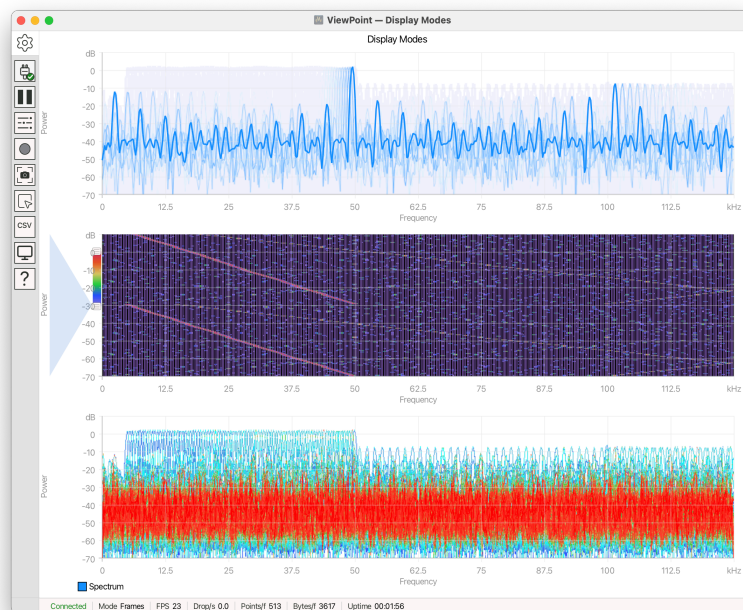
- **`addData("Path", x, y)`** appends paired points to a Scatter trace.
- **`addBreak("Curve")`** lifts the pen so the next point starts a new stroke, essential when one trace spans a discontinuity (an asymptote, a sensor dropout, or separate phasor segments) and you don't want the gap bridged by a connecting line.
- **`view.begin(scatter, frames, 40)`** selects the plot type where arbitrary paired (x, y) data is meaningful; sets the update mode to Frames; sets the number of points per frame to 40.

13. Multiple display modes from the same frame stream

Display modes are easiest to understand when the same frame stream is rendered three different ways. This recipe uses VPDSP to generate a sweeping spectrum, then shows Persistence, Spectrogram, and Gradient on stacked plots so their different uses are directly comparable.

Note that all three plots share the same base configuration, this is easily applied by using the base plot. Individual (display mode) configuration is applied to each plot independently.

See [Display Modes](#) under the [Controls](#) panel.



Recipe 13 - Persistence, Spectrogram, and Gradient from one frame stream

Declarations and helpers

```
#include <ViewPoint.h>
#include <VPDSP.h>

using namespace viewpoint;

#define SAMPLE_RATE 25000
#define FFT_SIZE 1024
#define FFT_OUTPUT_SIZE (FFT_SIZE / 2 + 1)
#define CARRIER_START 5000
#define CARRIER_END 50000
#define CARRIER_STEP 500

signal_generator sig(SAMPLE_RATE);
FFT<(DSP_SIZE)FFT_SIZE> fft;
float samples[FFT_SIZE];
uint32_t carrier = CARRIER_START;
```

Configuration

```

void setup() {
    view.begin(frames, FFT_OUTPUT_SIZE);
    view.setDelay(40);
    view.setTitle("Display Modes");
    view.setNumberOfPlots(3);

    view.trace("Spectrum").setColor(colors::DodgerBlue);

    // Apply base configuration to each plot
    view.setHorizontalRange(0, SAMPLE_RATE / 2000, 10);
    view.setVerticalRange(-70, 10, 8);
    view.setAxisLabels("Frequency", "Power");
    view.setUnits("kHz", "dB");

    // Apply individual configuration to each plot
    view.plot(0).setTitle("Persistence");
    view.plot(0).setDisplayMode(persistence, 0);

    view.plot(1).setTitle("Spectrogram");
    view.plot(1).setDisplayMode(spectrogram, 0);

    view.plot(2).setTitle("Gradient");
    view.plot(2).setDisplayMode(gradient, 0);

    sig.set_carrier(shape_square, carrier);
    sig.set_AM(shape_sine, 2, 0.01);
    sig.set_noise(3.0f);
    fft.set_window_type(WIN_BLACKMAN_HARRIS);
}

```

Loop

```

void loop() {
    for (int i = 0; i < FFT_SIZE; i++) {
        samples[i] = 1.65f * sig.next_sample(0);
    }

    fft.calculate(samples);
    fft.post_scale_dB(-120, 1.65f);

    view.addData("Spectrum", (float*)fft, FFT_OUTPUT_SIZE);
    view.send();

    carrier += CARRIER_STEP;
    if (carrier > CARRIER_END) carrier = CARRIER_START;
    sig.set_carrier(shape_square, carrier);
}

```

What the functions do

- **view.begin(frames, FFT_OUTPUT_SIZE)** makes each FFT output one complete frame.
- **setDisplayMode(persistence, 0)** keeps previous frames visible as fading trails.
- **setDisplayMode(spectrogram, 0)** turns each frame into a waterfall column.
- **setDisplayMode(gradient, 0)** accumulates repeated hits into a heatmap.

When to use this

- Use **Persistence** for transient capture and repeated overlays.
- Use **Spectrogram** to view amplitude as color over time.
- Use **Gradient** when you want to see where frame data accumulates most often.
- Keep **axis ranges** fixed; auto-scaling makes display-mode comparisons harder to read.

See `examples/Cartesian/SpectrumAnalyzer/SpectrumAnalyzer.ino`

Quick Troubleshooting

Symptom	Likely Cause	What To Try
Port does not appear in Settings	Board not enumerated, charge-only cable, wrong USB port, or board still rebooting after upload	Unplug and reconnect the board, try a data-capable USB cable, wait a few seconds after upload, reopen the port list, and confirm the board appears in Arduino IDE.
ViewPoint cannot connect	Another app owns the serial port	Close Arduino Serial Monitor, Arduino Serial Plotter, PlatformIO monitor, terminal serial tools, or any other serial app, then reconnect.
Connects, but no plot appears	Sketch is not sending ViewPoint data, the wrong sketch is running, or <code>view.begin()</code> / <code>view.send()</code> is missing	Upload HelloPlotter, confirm the sketch includes <code><ViewPoint.h></code> , calls <code>view.begin()</code> in <code>setup()</code> , and calls <code>view.addData(...)</code> plus <code>view.send()</code> in <code>loop()</code> .
Plot appears, but labels or configuration do not apply	Data is arriving, but ViewPoint protocol configuration is not being negotiated	Make sure the sketch uses the VoidLoop ViewPoint™ library, <code>setLegacyMode(true)</code> is not enabled, and the app was connected before or shortly after the sketch started.
Data appears, but scaling is wrong	Axis bounds, auto-scale, log scale, or trace auto-scaling settings do not match the data	Open Controls > Axis Configuration . Re-enable Auto-Scale, check manual Min/Max bounds, disable Logarithmic scale for zero or negative data, and confirm the trace has Affects Auto-Scaling enabled.
Trace is visible but changes the scale too much	A noisy or outlier trace is included in auto-scaling	Open Controls > Traces and turn off Affects Auto-Scaling for that trace, or use a fixed axis range.
Display modes are unavailable or do nothing	Display modes require the right plot type and update mode	Use Cartesian data in Frames mode. In sketches, call <code>view.begin(frames, packetSize)</code> or <code>view.setMode(frames)</code> , then send a full frame before <code>view.send()</code> .

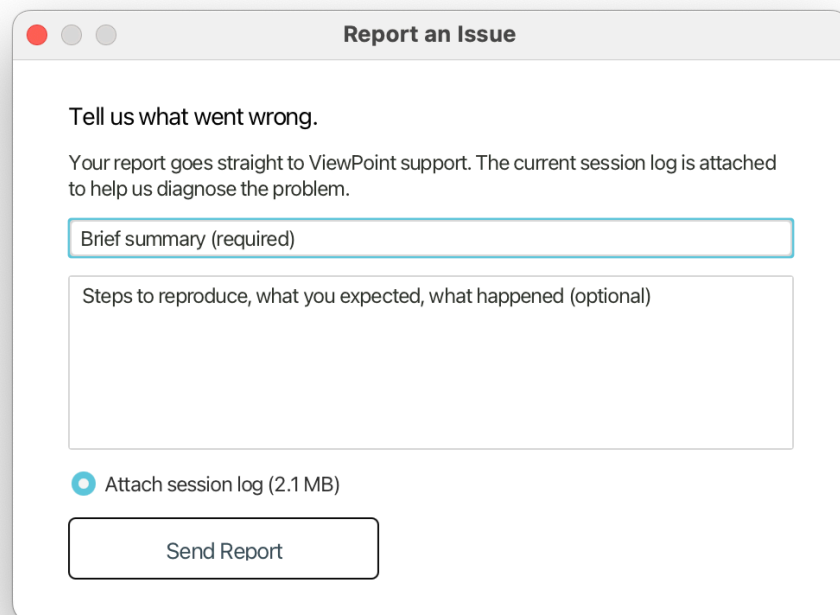
Symptom	Likely Cause	What To Try
Frames mode looks wrong	send() is being called at the wrong time, or packet size does not match the frame	In Frames mode, call addData(...) repeatedly to fill the frame, then call view.send() once per complete frame. Check that packetSize matches the number of points sent.
CSV capture is missing timestamps	The sketch is not sending source timestamps, or CSV settings are using a different timestamp source	VoidLoop ViewPoint™ can auto-supply timestamps, however, to send sketch timestamps, pass a timestamp to view.send(timestamp). Also check Settings > CSV Capture timestamp options.
CSV capture is missing expected traces	Trace labels changed, traces were hidden, or capture started before data and configuration arrived	Start capture after the plot is configured and visible. Confirm trace names in the legend and CSV settings .
Screen recording is unavailable or low quality	Recording backend is missing or limited	JCodec works without setup but is limited to 10 FPS. Install FFmpeg for higher frame rates, better quality, and audio support, then confirm VoidLoop ViewPoint™ detects it in Settings > Screen Capture .
Serial Monitor shows data but the plot is wrong	Raw serial format does not match ViewPoint's expected data or command protocol	If using the library, compare against HelloPlotter. If writing a custom sender, see Appendix B: Serial Command Reference .
Arduino Serial Plotter works, but ViewPoint labels or configuration do not	Sketch may be in legacy/data-only mode	Disable view.setLegacyMode(true) when targeting ViewPoint. Legacy mode is for Arduino Serial Plotter compatibility and skips VoidLoop ViewPoint™ configuration commands.
After uploading firmware, connection drops	The board disconnects and reconnects during upload	Wait for the board to reappear, then reconnect. Enable Auto-Reconnect in Settings if you frequently upload while VoidLoop ViewPoint™ is open.

Help and Feedback

Join us on the VoidLoop ViewPoint™ Forum!

For additional support please email our team at support@voidloop.com or for returns and license issues email license@voidloop.com

Report an Issue from within ViewPoint



Report an Issue

Tell us what went wrong.

Your report goes straight to ViewPoint support. The current session log is attached to help us diagnose the problem.

Brief summary (required)

Steps to reproduce, what you expected, what happened (optional)

☒ Attach session log (2.1 MB)

Send Report

To report an issue from within VoidLoop ViewPoint™, use the *Help* → *Report an Issue...* dialog. The application logs from your most recent session will be automatically attached. A reference number will be returned to you and someone from our team will be in touch as soon as possible. Email supporting screen captures to support@voidloop.com with the reference number in the subject or body of the email.

Please include as much relevant information as possible in the Description field. Helpful information to include: the steps taken that resulted in the issue, the result you were hoping for versus what happened, and any follow up steps you took to resolve it. The platform, app version, and similar fields will be included automatically.

Appendix A: API Reference

Every public method on `view` (the `Plotter` instance), grouped by purpose. Defaults shown in parentheses. The "Desktop result" column describes the visible effect once the desktop app is connected and negotiated.

Lifecycle

Method	Behavior	Desktop result
<code>begin(unsigned long baud = 115200, PlotType type = Cartesian, Mode mode = Continuous, int packetSize = VIEWPOINT_DEFAULT_PACKET_SIZE)</code>	Initialize default <code>Serial</code> at baud, set the trailing plot type, mode, and packet size, start protocol negotiation	App opens the session and reports connection on the next [viewpoint]version=exchange
<code>begin(Stream& serial, PlotType type = Cartesian, Mode mode = Continuous, int packetSize = default)</code>	Use a caller-initialized <code>Stream</code> (the caller owns <code>serial.begin()</code>); same trailing defaults	Same handshake, on the supplied stream
<code>begin(PlotType type, Mode mode = Continuous, int packetSize = default)</code>	Default <code>Serial</code> at 115200, leading with plot type	App opens with the chosen plot type
<code>begin(Mode mode, int packetSize = default)</code>	Default <code>Serial</code> at 115200, Cartesian by default, leading with mode	App opens in the chosen mode
<code>send(const char* timestamp = "")</code>	Process incoming protocol bytes, send any pending configuration, drain buffered trace data, optionally emit a timestamp, send the frame-complete marker in Frames mode, clear trace buffers	App receives the configured frame or batch of points
<code>reset()</code>	Free library-owned traces, clear plot config, restore type/mode/packet defaults; triggers a full handshake on next <code>send()</code>	App clears all state on the next [viewpoint]version=...
<code>removeTrace(int id)</code>	Free the trace at <code>id</code> , drop any derived traces that referenced it, force a full handshake	App removes the trace and its legend entry
<code>clearTraces()</code>	Free every trace and derived trace; force a full handshake	App clears all traces
<code>clearPlots()</code>	Reset every plot's title, axes, grid, display mode, and trace list	App resets all plot configuration

The four `begin(...)` overloads cover the four natural starting shapes: default `Serial` with optional baud and trailing defaults, a caller-provided `Stream`, plot-type-first, or mode-first. Default arguments fill in the rest, so a sketch can write `view.begin()`, `view.begin(scatter)`, `view.begin(frames, 512)`, or `view.begin(Serial1, polar, frames, 360)` without juggling argument order beyond the leading type.

Plot Type, Mode, and Delay

Method	Behavior	Desktop result
<code>setPlotType(PlotType)</code>	Set Cartesian, Scatter, or Polar; switching to Polar forces packet size to 360	App switches to the chosen layout
<code>setMode(Mode)</code>	Set Continuous or Frames; sets a sensible default <code>delayMs_</code> for each (2 ms / ~16.7 ms) and enables TX buffering in Frames mode	App treats the stream as live points or bounded frames
<code>setPacketSize(int size)</code>	Set the points-per-frame target in Frames mode	App expects this many points per [frames]complete=true
<code>setDelay(float ms)</code>	Override the inter-sample (Continuous) or inter-frame (Frames) delay in milliseconds	None directly — slows or speeds the stream rate
<code>setLegacyMode(bool)</code>	Bypass the [viewpoint] handshake, suppress frame-complete markers, clamp <code>send()</code> to 500 points per call	None — legacy mode targets non-ViewPoint receivers

Default Plot Configuration

These calls configure the **default plot** — the implicit plot every sketch starts with. In a multi-plot sketch, the same setters are available on `view.plot(i)` for per-plot configuration.

Method	Behavior	Desktop result
<code>setVerticalRange(min, max)</code>	Set the y-axis range	App draws the y-axis with these bounds
<code>setVerticalRange(min, max, int divisions)</code>	Range plus division count	Axis gridlines split into divisions segments
<code>setVerticalRange(min, max, float minor, float major)</code>	Range plus explicit minor and major step	Axis gridlines at the given spacings
<code>setHorizontalRange(min, max)</code> and overloads	Same idea for the x-axis	App draws the x-axis with these bounds
<code>setUnits(const char* h, const char* v)</code>	Set horizontal and vertical unit strings	Units appear in the axis labels
<code>setRadialRange(min, max)</code> and overloads	Polar radial limits and divisions; mirrors the vertical-range API	App draws the radial axis from min to max
<code>setAngularOffset(float)</code>	Rotate the polar angle reference	Polar grid rotates by this offset
<code>setAngularStep(int)</code>	Set the angular grid spacing	Polar grid lines step by this amount
<code>setAngularUnits(AngularUnit)</code>	Switch between degrees and radians	App interprets incoming angle values accordingly
<code>setRadians()</code>	Shortcut for <code>setAngularUnits(Radians)</code>	Same as above
<code>setAxisLabels(const char* h, const char* v)</code>	Set the axis label text (independent of units)	App shows the labels next to each axis
<code>setTitle(const char* title)</code>	Set the sketch-level title, shown above the plot region and in the window title	App displays the sketch title
<code>setPlotTitle(const char* title)</code>	Set the title of the default plot	App displays the plot title

Logarithmic Scale

Method	Behavior	Desktop result
<code>enableLogarithmicScale(bool mapData = true)</code>	Enable log scaling on the default plot's vertical axis. <code>mapData=true</code> means trace data is linear and the app maps it; <code>false</code> means data is already in log-space (e.g., dB) and is plotted linearly against log-spaced axis bounds	App renders the y-axis with log gridlines and treats incoming data as configured
<code>disableLogarithmicScale()</code>	Disable log scaling on the vertical axis	App reverts to linear y-axis

For axis-specific control, use `view.plot(i).enableLogarithmicScale(axis, mapData)`.

Grid and Reference Lines

Method	Behavior	Desktop result
<code>setGridColors(uint32_t minor, uint32_t major)</code>	Set the minor and major gridline colors	Grid redraws with the chosen colors
<code>setGridColors(uint32_t labels, uint32_t minor, uint32_t major)</code>	Also sets axis-label color	Labels and gridlines redraw with the chosen colors
<code>addHorizontalReferenceLine(float value, bool isMajor = false)</code>	Add a <i>horizontal line</i> on the vertical axis at value. The line is drawn flat across the plot at constant y	App overlays a horizontal line at the given y-value
<code>addHorizontalReferenceLine(float value, uint32_t color, float stroke = 1.0f)</code>	Same with explicit color and stroke	Same, in the requested color and weight
<code>addVerticalReferenceLine(float value, bool isMajor = false)</code>	Add a <i>vertical line</i> on the horizontal axis at value. The line is drawn straight up at constant x	App overlays a vertical line at the given x-value
<code>addVerticalReferenceLine(float value, uint32_t color, float stroke = 1.0f)</code>	Same with explicit color and stroke	Same, in the requested color and weight

Display Modes

Method	Behavior	Desktop result
<code>setDisplayMode(DisplayMode mode)</code>	Apply a display mode to the default plot (Cartesian + Frames only)	App switches the plot's renderer to the chosen mode
<code>setDisplayMode(DisplayMode mode, int traceId)</code>	Same, scoped to one trace	App applies the display mode to that trace's drawing only

Multi-Plot

Method	Behavior	Desktop result
<code>setNumberOfPlots(int count)</code>	Allocate count plot panels (clamped to <code>VIEWPOINT_MAX_PLOTS</code>)	App splits the canvas into the requested number of stacked plots
<code>plot(int index)</code>	Return a <code>Plot&</code> for the plot at index. The returned object exposes the same axis, title, units, display-mode, and reference-line API as view itself	None directly — subsequent setters on the returned <code>Plot</code> configure that panel

Trace Management

Method	Behavior	Desktop result
<code>createTrace(int id)</code>	Allocate a library-owned trace at <code>id</code> with capacity <code>packetSize</code>	A new trace is added to the legend on the next handshake
<code>createTrace(int id, size_t capacity)</code>	Same with explicit capacity	Same
<code>createTrace(int id, float* buffer, size_t size)</code>	Wrap a caller-provided single-value buffer (no allocation, no growth)	Same
<code>createTrace(int id, float* x, float* y, size_t size)</code>	Wrap caller-provided paired buffers	Same
<code>trace(int id)</code>	Return the existing trace at <code>id</code> , or create a library-owned one if missing	Same as <code>createTrace</code> if a trace is created
<code>trace(const char* label)</code>	Look up a trace by label; create one if there is no match	A new trace is added with the supplied label
<code>addData(int id, float value)</code>	Append a single value to the trace at <code>id</code>	Trace updates on next <code>send()</code>
<code>addData(const char* label, float value)</code>	Append by label (creates the trace if missing)	Same
<code>addData(id, float x, float y), addData(label, x, y)</code>	Append a paired value	Trace updates on next <code>send()</code>
<code>addData(id, const float* data, size_t count)</code> and label variant	Append an array of values in one call	Trace updates on next <code>send()</code>
<code>addData(id, const float* x, const float* y, size_t count)</code>	Append paired arrays	Trace updates on next <code>send()</code>
<code>addBreak(int id), addBreak(const char* label)</code>	Insert a discontinuity in a Scatter or Polar trace	App starts a new line segment without connecting back to the previous point
<code>sendData(int id, float value)</code>	Send one value for a single trace channel immediately, without buffering through Trace	App receives one out-of-band data line
<code>sendData(int id, float x, float y)</code>	Same for paired values	Same

Messaging

Method	Behavior	Desktop result
<code>sendInfo(const char* fmt, ...)</code>	<code>printf</code> -style formatted info message (up to ~500 chars)	App shows a [message] notification
<code>sendInfo(const char* title, const char* msg)</code>	Titled info message	App shows the titled message
<code>sendError(const char* fmt, ...)</code>	<code>printf</code> -style formatted error message	App shows an [error] notification
<code>sendError(const char* title, const char* msg)</code>	Titled error message	App shows the titled error

State Queries

Method	Returns
<code>isReady()</code>	true once the protocol is in <code>Negotiated</code> or <code>DataOnly</code> state, or whenever legacy mode is enabled
<code>isNegotiated()</code>	true only after the <code>[viewpoint]version=...</code> handshake completed
<code>packetSize()</code>	Current packet size (Frames mode target)
<code>traceCount()</code>	Number of active traces
<code>plotType()</code>	Current <code>PlotType</code>
<code>mode()</code>	Current Mode
<code>isLegacyMode()</code>	Whether <code>setLegacyMode(true)</code> is in effect

Advanced

Method	Behavior
<code>update()</code>	Run the protocol state machine and drain incoming bytes without sending data. Useful when a sketch polls a slow sensor and may not call <code>send()</code> for seconds at a time
<code>protocol()</code>	Return the <code>Protocol</code> for advanced use: version checks (<code>protocol().supports(major, revision)</code>), state inspection (<code>protocol().isNegotiated()</code>), and direct command building

Appendix B: Advanced Serial Command Reference

This reference covers the protocol-level command families visible in the Serial Monitor. Useful for understanding the sketch-to-app communication or debugging protocol issues or writing libraries to support additional platforms.

Command Family	What It Controls
[viewpoint]	handshake and protocol setup — resets serial configuration and traces
[cartesian], [scatter], [polar]	plot type and number of plots
[continuous], [frames]	update mode
[packet size]	frame or packet sizing
[axis]	ranges, steps, and divisions
[units]	units text
[plot labels] and [sketch]	plot and sketch titles
[trace labels] and [trace color]	trace naming and color
[trace filter]	derived or filtered traces
[display mode]	spectrogram, persistence, gradient, or none
[grid colors] and [reference line]	visual overlays and grid styling
[time]	timestamps
[message] and [error]	app messages and warnings

This appendix describes the human-readable V1 command protocol for sending configuration commands to VoidLoop ViewPoint™ over serial. It is intended for users who want to write their own microcontroller or host-side library.

Command Format

Commands are one line of text ending with a newline:

```
[command]key=value,key=value
```

The command token, including brackets, is case-sensitive and must match exactly. Argument names are case-insensitive because VoidLoop ViewPoint™ lowercases keys before lookup. Argument values are not escaped; avoid commas and equals signs inside text values.

Only lines that start with [are treated as V1 commands. Data lines are separate CSV-style numeric rows.

Session Lifecycle

A normal configuration sequence is:

```
[viewpoint]version=V1R1
[cartesian]plots=1
[continuous]
[packet size]size=500
[trace labels]labels=Trace 1
[viewpoint]completed=true
```

[viewpoint]version=V1R1 starts a new configuration session and resets the app-side serial plot configuration. Send plot setup commands after it. [viewpoint]completed=true ends the configuration block and notifies VoidLoop ViewPoint™ to use the configuration.

In frames mode, send [frames]complete=true after each complete frame of data so VoidLoop ViewPoint™ can determine when to commit and render that frame.

Common Values

Value type	Accepted values
Plot index	0, 1, 2, etc. Use -1 or omit plot where supported to target all plots.
Axis	x, y, r, t. r maps to the radial axis and t maps to theta for polar plots.
Boolean	true or false in lowercase.
Color	Hex RGB or ARGB: F00, FF0000, 80FF0000, #00FF00, or 0x00FF00. Six-digit RGB is treated as opaque.
Display mode	none, spectrogram, persistence, gradient.
Trace filter	none, average, maxhold.
Reference line style	solid, dashed, dotted, pattern.
Axis scaling strategy	none, expand_only, windowed, robust_percentile, peak_decay, debounced, symmetric_zero, baseline_zero.

Commands

[viewpoint]

Starts or completes a configuration block.

Argument	Required	Description
version	Start only	Protocol version. Use V1R1.
completed	Complete only	Use true to mark the configuration block complete.

Examples:

```
[viewpoint]version=V1R1  
[viewpoint]completed=true
```

[sketch]

Sets the sketch/session title.

Argument	Required	Description
title	Yes	Title text.

Example:

```
[sketch]title=Motor Test
```

[cartesian]

Selects Cartesian plotting and sets the number of plots.

Argument	Required	Description
plots	No	Number of plots. Defaults to 1.

Example:

```
[cartesian]plots=2
```

[scatter]

Selects scatter plotting and sets the number of plots.

Argument	Required	Description
plots	No	Number of plots. Defaults to 1.

Example:

```
[scatter]plots=1
```

[polar]

Selects polar plotting and configures the angular axis. Polar mode uses one plot.

Argument	Required	Description
degrees	No	true for degrees, false for radians. Defaults to true.
offset	No	Angular offset. Defaults to 0.
step	No	Angular segment size. Defaults to 15.

Example:

```
[polar]degrees=true,offset=0,step=15
```

[continuous]

Sets continuous scrolling mode. No arguments.

Example:

```
[continuous]
```

[frames]

Sets frames mode, or marks one frame complete.

Argument	Required	Description
complete	No	Send true after a full frame of data. Omit this argument when selecting frames mode.

Examples:

```
[frames]  
[frames]complete=true
```

[packet size]

Sets the display window size in samples.

Argument	Required	Description
size	Yes	Number of samples retained in the visible packet/window.

Example:

```
[packet size]size=1024
```

[trace labels]

Sets primary trace labels in channel order.

Argument	Required	Description
labels	Yes	Comma-separated labels. The parser treats the whole value after labels= as the label list.

Example:

```
[trace labels]labels=Temperature,Pressure,Status
```

[trace color]

Sets one trace color.

Argument	Required	Description
trace	Yes	Primary trace index.
color	Yes	Hex color.

Example:

```
[trace color]trace=0,color=FF0000
```

Send one [trace color] command per trace. The bulk form colors=... is not supported by the V1 app parser.

[trace filter]

Creates or removes an app-side derived trace for a primary source trace. Derived traces are currently useful for Cartesian frames-mode workflows.

Argument	Required	Description
trace	Yes	Source primary trace index.
type	Yes	average, maxhold, minhold, or none.
param	No	Filter parameter. Defaults to 0.3 for average, otherwise 0.95.
color	No	Hex color for the derived trace.
label	No	Label for the derived trace.

Examples:

```
[trace filter]trace=0,type=average,param=0.25,color=00AAFF,label=Average  
[trace filter]trace=0,type=none
```

type=none removes derived traces for the specified source trace.

[add traces]

Associates existing traces with a plot.

Argument	Required	Description
plot	No	Plot index. Defaults to 0.
traces	Yes	Comma-separated trace indices.

Example:

```
[add traces]plot=1,traces=2,3
```

[plot labels]

Sets a plot title and axis labels.

Argument	Required	Description
plot	No	Plot index, or omit/use -1 for all plots.
title	No	Plot title.
x	No	X-axis label.
y	No	Y-axis label.

Example:

```
[plot labels]plot=0,title=Temperature,x=Time,y=Voltage
```

[units]

Sets axis units.

Argument	Required	Description
plot	No	Plot index, or omit/use -1 for all plots.
x	No	X-axis unit.
y	No	Y-axis unit.

Example:

```
[units]plot=0,x=s,y=mV
```

[axis]

Configures an axis range and tick behavior.

Argument	Required	Description
plot	No	Plot index, or omit/use -1 for all plots.
axis	No	x, y, r, or t. Unknown or omitted axes are ignored by most app behavior.
min	No	Axis minimum.
max	No	Axis maximum.
auto	No	true or false. If min or max is sent and auto is omitted, ViewPoint treats auto-scaling as disabled for that axis.
strategy	No	Axis scaling strategy.
divisions	No	Number of tick divisions. Positive values select division-based ticks.
minor	No	Minor tick step. Positive minor or major selects step-based ticks.
major	No	Major tick step.

Examples:

```
[axis]plot=0,axis=y,min=-1,max=1,minor=0.1,major=0.5
[axis]plot=0,axis=y,auto=true,strategy=windowed
```

If min and max are both present and inverted, VoidLoop ViewPoint™ swaps them. If they are equal or non-finite, the command is ignored. divisions values above 1000 are clamped.

[log scale]

Enables or disables logarithmic scaling for an axis. Log scaling is supported on Cartesian Y, scatter X/Y, and polar radial axes. Unsupported combinations are ignored.

Argument	Required	Description
plot	No	Plot index, or omit/use -1 for all plots.
axis	Yes	Axis to scale.
enabled	No	true or false. Omitted means enabled.
mapdata	No	true when incoming data is linear and the data should be mapped to log space. false when incoming data is already log-scaled.

Examples:

```
[log scale]plot=0,axis=y
[log scale]plot=0,axis=y,enabled=false
[log scale]plot=0,axis=y,mapdata=false
```

[\[reference line\]](#)

Adds or updates a custom reference line.

Argument	Required	Description
plot	No	Plot index, or omit/use -1 for all plots.
axis	Yes	Axis for the reference line.
value	Yes	Axis value where the line is drawn.
color	No	Hex color.
stroke	No	Line stroke width. Defaults to 1.
id	No	Stable sender-side ID. Reusing the same ID updates that line instead of adding another one.
style	No	solid, dashed, dotted, or pattern.
ismajor	No	Marks the line as major.

Example:

```
[reference line]plot=0,axis=y,value=2.5,color=FFAA00,stroke=1.5,id=0,style=dashed
```

[\[grid colors\]](#)

Sets plot grid and label colors.

Argument	Required	Description
plot	No	Plot index, or omit/use -1 for all plots.
labels	No	Axis label/tick text color.
major	No	Major gridline color.
minor	No	Minor gridline color.
stroke	No	Axis stroke width.

Example:

```
[grid colors]plot=0,labels=FFFFFF,major=666666,minor=333333,stroke=1
```

[display mode]

Sets a plot display mode.

Argument	Required	Description
plot	No	Plot index, or omit/use -1 for all plots.
mode	Yes	none, spectrogram, persistence, or gradient.
trace	No	Trace index used by the display mode. Defaults to 0; -1 means all traces.
colors	No	Comma-separated hex colors for spectrogram color maps.

Examples:

```
[display mode]plot=0,mode=persistence,trace=0
[display mode]plot=0,mode=spectrogram,trace=0,colors=000000,00AAFF,FFFFFF
[display mode]plot=0,mode=none
```

[time]

Provides a source timestamp for subsequent data rows, primarily for CSV capture alignment.

Argument	Required	Description
raw	Yes	Numeric timestamp. Values below 1000000000000000 are interpreted as microseconds; larger values are interpreted as nanoseconds.

Example:

```
[time]raw=1234567890
```

[message] and [error]

Accepted by the parser for sender-side status messages. In the current app command pipeline, they do not change plot configuration.

Command	Arguments
[message]	title optional, message optional
[error]	title optional, message optional

Examples:

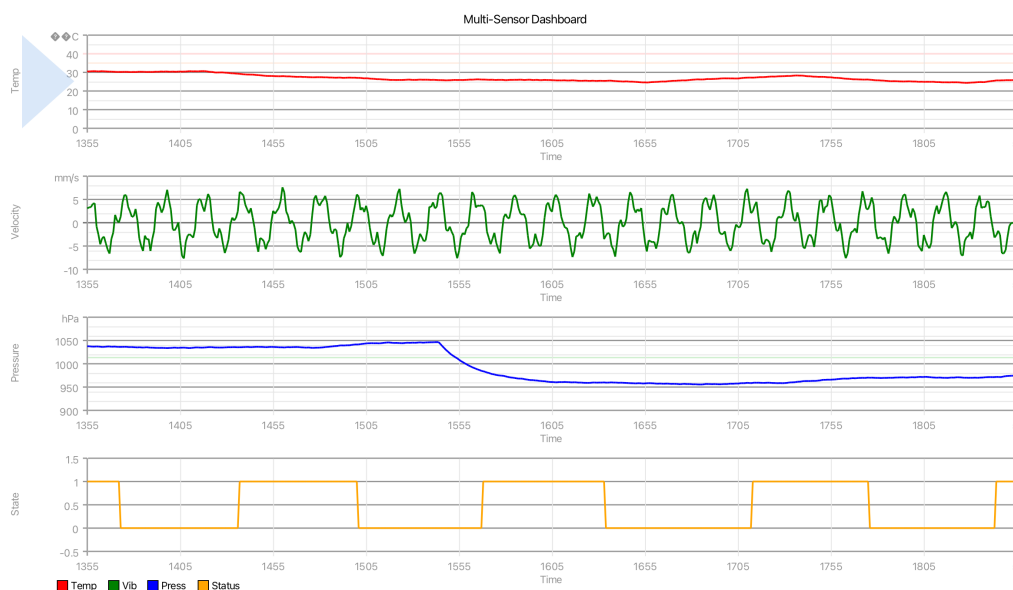
```
[message]title=Status,message=Starting capture
[error]message=Sensor disconnected
```


Complete Example

```

[viewpoint]version=V1R1
[sketch]title=Multi-Sensor Dashboard
[cartesian]plots=4
[continuous]
[packet size]size=500
[trace labels]labels=Temp,Vib,Press,Status
[trace color]trace=0,color=F00
[trace color]trace=1,color=008000
[trace color]trace=2,color=00F
[trace color]trace=3,color=FFA500
[plot labels]plot=0,title=Temperature,x=Time,y=Temp
[axis]plot=0,axis=y,min=0,max=50.00,minor=5.0000,major=10.00
[reference line]id=0,plot=0,axis=y,value=35.00,color=21FF6600,stroke=1.5000
[reference line]id=1,plot=0,axis=y,value=40.00,color=21FF0000,stroke=2.0000
[units]plot=0,x=s,y=°C
[plot labels]plot=1,title=Vibration,x=Time,y=Velocity
[axis]plot=1,axis=y,min=-10.00,max=10.00,minor=2.0000,major=5.0000
[reference line]id=0,plot=1,axis=y,value=0,color=666,stroke=1.0000
[units]plot=1,x=s,y=mm/s
[plot labels]plot=2,title=Pressure,x=Time,y=Pressure
[axis]plot=2,axis=y,min=900.00,max=1100.00,minor=20.00,major=50.00
[reference line]id=0,plot=2,axis=y,value=1013.25,color=3100AA00,stroke=1.5000
[units]plot=2,x=s,y=hPa
[plot labels]plot=3,title=System Status,x=Time,y=State
[axis]plot=3,axis=y,min=-0.5000,max=1.5000,minor=1.0000,major=0.5000
[units]plot=3,x=s
[viewpoint]completed=true

```



Appendix C: Library Color Reference

Pinks			Greens		Blues		Browns				
	Pink	#FFC0CB		LawnGreen	#7CFC00		PowderBlue	#B0E0E6		Cornsilk	#FFF8DC
	LightPink	#FFB6C1		Chartreuse	#7FFF00		LightBlue	#ADD8E6		BlanchedAlmond	#FFEBCD
	HotPink	#FF69B4		LimeGreen	#32CD32		LightSkyBlue	#87CEFA		Bisque	#FFE4C4
	DeepPink	#FF1493		Lime	#00FF00		SkyBlue	#87CEEB		NavajoWhite	#FFDEAD
	MediumVioletRed	#C71585		ForestGreen	#228B22		DeepSkyBlue	#00BFFF		Wheat	#F5DEB3
	PaleVioletRed	#DB7093		Green	#008000		LightSteelBlue	#B0C4DE		BurlyWood	#DEB887
Reds				DarkGreen	#006400		DodgerBlue	#1E90FF		Tan	#D2B48C
	LightSalmon	#FFA07A		GreenYellow	#ADFF2F		CornflowerBlue	#6495ED		RosyBrown	#BC8F8F
	Salmon	#FA8072		YellowGreen	#9ACD32		SteelBlue	#4682B4		SandyBrown	#F4A460
	DarkSalmon	#E9967A		SpringGreen	#00FF7F		RoyalBlue	#4169E1		Goldenrod	#DAA520
	LightCoral	#F08080		MediumSpringGreen	#00FA9A		Blue	#0000FF		DarkGoldenrod	#B8860B
	IndianRed	#CD5C5C		LightGreen	#90EE90		MediumBlue	#0000CD		Peru	#CD853F
	Crimson	#DC143C		PaleGreen	#98FB98		DarkBlue	#00008B		Chocolate	#D2691E
	FireBrick	#B22222		DarkSeaGreen	#8FBC8F		Navy	#000080		SaddleBrown	#8B4513
	Red	#FF0000		MediumSeaGreen	#3CB371		MidnightBlue	#191970		Sienna	#A0522D
	DarkRed	#8B0000		SeaGreen	#2E8B57		MediumSlateBlue	#7B68EE		Brown	#A52A2A
Oranges				Olive	#808000		SlateBlue	#6A5ACD		Maroon	#800000
	Coral	#FF7F50		DarkOliveGreen	#556B2F		DarkSlateBlue	#483D8B	Grayscale		
	Tomato	#FF6347		OliveDrab	#6B8E23	Purples				White	#FFFFFF
	OrangeRed	#FF4500		PhosphorGreen	#60C030		Lavender	#E6E6FA		Snow	#FFFAFA
	Gold	#FFD700	Cyans				Thistle	#D8BFD8		Ivory	#FFFFF0

Orange	#FFA500	LightCyan	#E0FFFF	Plum	#DDA0DD	WhiteSmoke	#F5F5F5
DarkOrange	#FF8C00	Cyan	#00FFFF	Violet	#EE82EE	Gainsboro	#DCDCDC
Yellows		Aqua	#00FFFF	Orchid	#DA70D6	LightGray	#D3D3D3
LightYellow	#FFFFE0	Aquamarine	#7FFFD4	Fuchsia	#FF00FF	Silver	#C0C0C0
LemonChiffon	#FFFACD	MediumAquamarine	#66CDAA	Magenta	#FF00FF	DarkGray	#A9A9A9
LightGoldenrodYellow	#FAFAD2	PaleTurquoise	#AFEEEE	MediumOrchid	#BA55D3	Gray	#808080
PapayaWhip	#FFEFD5	Turquoise	#40E0D0	MediumPurple	#9370DB	DimGray	#696969
Moccasin	#FFE4B5	MediumTurquoise	#48D1CC	BlueViolet	#8A2BE2	SlateGray	#708090
PeachPuff	#FFDAB9	DarkTurquoise	#00CED1	DarkViolet	#9400D3	DarkSlateGray	#2F4F4F
PaleGoldenrod	#EEE8AA	LightSeaGreen	#20B2AA	DarkOrchid	#9932CC	Black	#000000
Khaki	#F0E68C	CadetBlue	#5F9EA0	DarkMagenta	#8B008B		
DarkKhaki	#BDB76B	DarkCyan	#008B8B	Purple	#800080		
Yellow	#FFFF00	Teal	#008080	Indigo	#4B0082		
YellowOrange	#FFB343	Grid					
		GridBackground	#1E1E1E				
		GridMinor	#333333				
		GridMajor	#666666				
		GridLabels	#FFFFFF				