Plaquette Documentation

Release 0.4.2

Plaquette

Nov 27, 2022

GUIDE

Plaquette is an object-oriented, user-friendly, signal-centric programming framework for **creative physical computing**. It promotes **expressiveness** over technical details while remaining fully compatible with Arduino, thus allowing **both beginner and advanced** creative practitioners to design meaningful physical computing systems in an intuitive fashion.

Plaquette allows you to:

- React to multiple sensors and actuators in real-time without interruption.
- Automatically calibrate sensors to generate stable interactions in changing environments.
- Design complex interactive behaviors by seamlessly combining powerful effects.

Quick links:

- *Discover the features*
- *Get started*
- *Filter your signals*

CHAPTER

ONE

TABLE OF CONTENTS

1.1 Why Plaquette?

1.1.1 Rationale

Media creators such as artists, interactive designers, and electronic musicians work with real-time sensory signals all the time. However, few of them (especially beginners) really "know their signals" and how to extract high-level informations from them (such as debouncing, smoothing, normalizing, detecting peaks and regularities).

Consider the following case of learning how to work with a simple photoresistor sensor plugged into an Arduino board on analog pin 0. The code reads as follow:

```
int value = analogRead(A0);
```

The value thus read is the raw 10-bits value returned by the Arduino board's Analog to Digital Converter (ADC) and thus reads as an integer between 0 and 1023. But how is that really useful for an artist who wants to use this value creatively.

For example, what if one wants to react to a flash of light? Well, one solution is to look at the kind of values we get and set a threshold.

```
if (value > 716)
  // do something
```

Nice. But there are two problems with this. First, while it might work under certain light conditions, it will likely stop working if these conditions change, forcing us to make adjustments by hand to the threshold value.

Second, and perhaps more importantly, this piece of code does not really *expresses* what we are after. As creative practitioners, we don't care whether the light signal is above 716 or 456 or whatnot: what we really want to know is whether it is *significantly high compared to ambient light*.

What this example shows is that the way we are teaching and learning about sensor data is ineffective for creative applications. In other words: **raw digital data lacks expressiveness**.

Continuing with our example, consider how one would take the input value and directly reroute it to an analog (PWM) output on pin 9:

```
analogWrite(9, value / 4);
```

Why do we need to perform that division by 4? That's because while the ADC gives us 10-bit values (1024 possibilities), the PWM only supports 8 bits (256 possibilities) forcing us to divide the incoming value by 4 (2 bits). But again, why is this important to know for an artist, designer, or musician? And what exactly does it have to do with our (expressive) intention?

1.1.2 A new standard

As a way to address these issues, we propose to create a general-purpose standard interface for simple, real-time signal processing for media artists. The objectives are as follow:

1. **Allow creators to concentrate on the creative dimensions of their work** rather than on irrelevant numerical questions, hence also facilitating their learning.
2. **Provide creators practitioners with accessible tools** that grasp high-level concepts such as "normalizing" and "detecting peaks" (rather than specific, arcane techniques on "how" to extract these informations such as "FFT", "zero-crossing" or "Chebyshev filtering").
3. **Facilitate team work and interoperability** between applications by favouring an easily understandable, cross-platform way of thinking about real-time signals (for example by keeping all signals "in check" between 0 and 1).

Plaquette responds to these challenges by adopting the following characteristics:

- **Easy to learn** by provide carefully-chosen functionalities that respond to common problems faced by creators ie. limited to only a few core functionalities that will solve 95% of your problems.
- **Real-time** by allowing responsive interaction without interruptions.
- **Focused on signals** rather than on numerical values such as 255, 1024, 716, etc.)
- **Robust** by tolerating changes in the sensory context without breaking down, because interactive works are often presented in environments that are difficult to fully control.
- **Interoperable and extensible** by adopting an object-oriented architecture fully compatible with Arduino.

1.2 Features

Plaquette is an *object-oriented*, *user-friendly*, *signal-centric* framework that facilitates *signal filtering* in *real-time*. It is fully *compatible with Arduino*.

1.2.1 Object-oriented

Plaquette is designed using input, output, and filtering units that can be easily interchanged in a plug-and-play fashion. Units are created using expressive code.

For example, the code `DigitalOut led` creates a new digital output object that can be used to control and LED.

Arduino	Plaquette
Create digital output to control an LED:	
`pinMode(12, OUTPUT);`	`DigitalOut led(12);`
Create digital input push-button:	
`pinMode(2, INPUT_PULLUP);`	`DigitalIn button(2);`

1.2.2 User-friendly

Plaquette allows users to quickly design interactive systems using an expressive language that abstracts away low-level functions. This allows both beginners and experts to create truly expressive code. For example, switching our LED object can be achieved by calling: `led.on()`. Find out more about Plaquette's base units by following *this link*.

Arduino	Plaquette
Turn LED on:	
`digitalWrite(12, HIGH);`	`led.on();`
Check if button is pushed:	
`if (digitalRead(2) == LOW)`	`if (button.isOn())`

1.2.3 Signal-centric

Plaquette helps designers manipulate real-time signals from inputs to outputs. In Plaquette, signals are represented as either `true/false` conditions in the case of digital binary signals such as those coming from a button or switch, or as floating-point numbers typically in the [0..1] range (ie. 0% to 100%) in the case of analog signals such as those emitted by a light sensor, microphone, or potentiometer. No more need to perform counter-intuitive conversions on integer values.

Arduino	Plaquette
Check if button is released:	
`if (digitalRead(2) != LOW)`	`if (!button)`
Check if sensor value is higher than 70%:	
`if (analogRead(A0) >= 716)`	`if (sensor >= 0.7)`

1.2.4 Signal Filtering

Plaquette provides simple yet powerful data filtering tools for debouncing, smoothing, and normalizing data. Removing noise in input signals can be as simple as calling a function such as `debounce()` or `smooth()`. Rather than guessing by trial-and-error the right threshold to trigger an event based on an input sensor, one can use auto-normalizing *filters* such as *MinMaxScaler* and *Normalizer*.

Signals in Plaquette can easily flow between units, in a similar fashion as modern data-flow softwares such as Max, Pure Data, and TouchDesigner. While this can be achieved using function calls, Plaquette further provides a special **piping operator** (>>) which allows the sending of data from one unit to another.

Arduino	Plaquette
Set LED to ON when button is pressed:	
`digitalWrite(12, digitalRead(2));`	`button >> led;`
Set LED to ON when input sensor is high:	
`digitalWrite(12, (analogRead(A0) >= 716 ? HIGH : LOW));`	`(sensor >= 0.7) >> led;`

Read *Regularizing Signals* to see how you can take full advantage of Plaquette's signal filtering features.

1.2.5 Real-time

Plaquette avoids blocking processes such as Arduino's (in)famous `delay()` by providing a set of *timing units* as well as time-based *signal generators*. The processing loop is thus never interrupted, allowing interactive and generative processes to flow smoothly.

Plaquette forbids the use of blocking functions such as Arduino's delay() and delayMicroseconds(). Rather, it invites programmers to adopt a frame-by-frame approach to coding similar to Processing.

Compare how a naive attempt to make an LED blink when pressing a button results in a slowly responding behavior in Arduino, versus Plaquette's real-time approach:

<table>
<tr><th>Arduino</th><th>Plaquette</th></tr>
<tr><td>

```
int buttonPin = 2;
int ledPin = 12;

void setup() {
  pinMode(buttonPin, INPUT_PULLUP);
  pinMode(ledPin, OUTPUT);
}

void loop() {
  // Button is checked only one per␣
↪second.
  if (digitalRead(buttonPin) == LOW) {
    digitalWrite(ledPin, HIGH);
    delay(500); // do nothing for 500ms
    digitalWrite(ledPin, LOW);
    delay(500); // do nothing for 500ms
  }
}
```

</td><td>

```
DigitalIn button(2);
DigitalOut led(12);

// Square wave with period of 1 second.
SquareOsc oscillator(1.0);

void begin() {}

void step() {
  // Button is checked all the time.
  if (button)
    oscillator >> led;
}
```

</td></tr>
</table>

1.2.6 Arduino compatible

Plaquette is installed as an Arduino library and provides a replacement for the core Arduino functionalities while remaining fully compatible with Arduino code. Seasoned Arduino users should consult the *Advanced Usage* section for some tips on how to integrate Plaquette into their existing code.

```
if (Serial.read() == 'T')
  led.toggle();
```

Warning: Plaquette is still at an experimental stage of development. If you have any issues or questions, please contact the developers or file a bug in our issue tracker.

1.3 Getting started

1.3.1 Step 1: Install Plaquette

If you do not have Arduino installed on your machine you need to download and install the Arduino IDE for your platform.

Once Arduino is installed, please install Plaquette as an Arduino library following these instructions.

1.3.2 Step 2: Your first Plaquette program

We will begin by creating a simple program that will make the built-in LED blink.

Create a new sketch

Create a new empty sketch by selecting **File > New**.

IMPORTANT: New Arduino sketches are initialized with some "slug" starting code. Make sure to erase the content of the sketch before beginning. You can use **Edit > Select All** and then click **Del** or **Backspace**.

Include library

Include the Plaquette library by typing:

```
#include <Plaquette.h>
```

Create an output unit

Now, we will create a new unit that will allow us to control the built-in LED:

```
DigitalOut myLed(13);
```

In this statement, `DigitalOut` is the **type** of unit that we are creating. There exist other types of units as we will soon see. `DigitalOut` is a type of unit that is attached to one of the many digital outputs on the Arduino board which can be set to one of two states ("on/off", "high/low", "1/0").

The word `myLed` is a **name** for the object we are creating.

Finally, `13` is a **parameter** of the object `myLed` that specifies the *pin* it corresponds to on the board.

In English, the statement would thus read as: "Create a unit named `myLed` of type `DigitalOut` on pin `13`."

Create an input unit

We will now create another unit that will generate a signal which will be sent to the LED to make it blink. To this effect, we will use the `SquareOsc` unit type which generates a square wave oscillating between "on/high/one" and "off/low/zero" at a regular period of 2.0 seconds and a duty-cycle of 50%:

```
SquareOsc myOsc(2.0, 0.5);
```

Create the begin() function

Each Plaquette sketch necessitates the declaration of two functions: `begin()` and `step()`.

Function `begin()` is called only once at the beginning of the sketch (just like the setup() function in Arduino). In our case, we do not to perform any special configuration at startup so we will leave the `begin()` function empty:

```
void begin() {}
```

Create the step() function

The `step()` function is called repetitively and indefinitely during the course of the program (like the loop() function in Arduino).

Here, we need to send the signal generated by the `myOsc` input unit to the `myLed` output unit. We will do this using Plaquette's special >> operator:

```
void step() {
  myOsc >> myLed;
}
```

In plain English, the statement `myOsc >> myLed` reads as: "Take the value generated by `myOsc` and put it in `myLed`."

Upload sketch

Upload your sketch to the Arduino board. You should see the LED on the board blinking once every two seconds at a regular pace.

Et voilà!

Full code

```
#include <Plaquette.h>

DigitalOut myLed(13);

SquareOsc myOsc(2.0, 0.5);

void begin() {}

void step() {
  myOsc >> myLed;
}
```

1.3.3 Step 3 : Experiment!

Period and duty cycle

Try changing the *period* and/or *duty-cycle* parameters in the square wave unit construction:

```
SquareOsc myOsc(<period>, <duty-cycle>);
```

- <period> can be any positive number representing the period of oscillation (in seconds)
- <duty-cycle> can be any number between 0.0 (0%) and 1.0 (100%) and represents the proportion of the period during which the signal is "high" (ie. "on duty")

What happens?

Adding and multiplying

Add another oscillator with a different period and duty cycle: multiply their values and send the result to the LED.

```
SquareOsc myOsc2(<period>, <duty-cycle>);
// ...
void step() {
  (myOsc * myOsc2) >> myLed;
}
```

Try adding their values instead: what do you see?

Use a conditional

Add a third oscillator that will "switch" between the two oscillators every 5 seconds using an if…else statement.

```
// TIP: omitting the duty-cycle parameter results in default value (0.5)
SquareOsc mySwitcher(5.0);
// ...
void step() {
  if (mySwitcher)
    myOsc >> myLed;
  else
    myOsc2 >> myLed;
}
```

ADVANCED: You can rewrite this expression in a more compact way using the ? : conditional operator:

```
void step() {
  (mySwitcher ? myOsc : myOsc2) >> myLed;
}
```

More examples

You will find more examples in **File > Examples > Plaquette** including:

- Using a button
- Using an analog input such as a photocell or potentiometer
- Using an analog output
- Basic filtering (smoothing, re-scaling)
- Serial input and output

1.4 Regularizing Signals

Plaquette provides expressive, automated, and robust ways to deal with signals for interactive design using **regularization filters** such as smoothing, min-max scaling, and normalization.

Here is a simple Arduino code that allows one to change the value of an output LED using an input photocell:

```
// The photocell analog pin.
int photoCellPin = A0;

// The output analog LED pin.
int ledPin = 9;

void setup() {
  // Initialize pins.
  pinMode(photoCellPin, INPUT);
  pinMode(ledPin, OUTPUT);
}

void loop() {
  // Read value from photocell (between 0..1023).
  int value = analogRead(photoCellPin);

  // Write value to LED (between 0..255).
  analogWrite(ledPin, value / 4);
}
```

As explained in *Why Plaquette?*, this simple code is complicated by the fact that the programmer needs to remember low-level information concerning the ranges of raw number values (1023, 255, ...) Furthermore, it fails to adapt to changing conditions such as the range of the ambient light.

Let's see how Plaquette can help us creating more expressive code using inputs and outputs signals rather than meaningless raw numbers.

1.4.1 Step 1 : Direct Input-to-Output

To begin, we will reimplement the example above using more expressive code.

First, let's define our input photocell on pin `A0` using an *AnalogIn* unit:

```
AnalogIn photoCell(A0);
```

Then, let's add an output analog LED on pin *9* using an *AnalogOut* unit:

```
AnalogOut led(9);
```

If we want to directly control the value of the LED from the value of the photocell, all we need to do is to send the photocell's value to the led. The easiest way to do so is using the `>>` operator:

```
photoCell >> led;
```

The complete Plaquette code will look like this:

```
#include <Plaquette.h> // include the Plaquette library

// Create input unit for photocell.
AnalogIn photoCell(A0);

// Create output unit for LED.
AnalogOut led(9);

// Initialize everything.
void begin() {
}

// Define frame-by-frame operations.
void step() {
  // Just send photo-cell value to LED.
  photoCell >> led;
}
```

1.4.2 Step 2 : Getting the Full Range of Signal

If we run this program, we will likely notice that the LED brightness will not span the full range from 0% to 100%. That's because depending on ambient lighting conditions, the photocell's values will not move across the full spectrum of possibilities. For instance, in the dark, the photocell might range from 10% to 50%, while in full daylight, it might range between 70% and 95%.

In order to resolve that issue, we need to **regularize** the photocell's signal. We can do so using a filtering unit such as a *MinMaxScaler*. This unit automatically keeps track of the minimum and maximum values taken by the incoming signal over time (for example, 10% and 50%) and remaps them into a new interval of [0, 1] (ie., 0% to 100%).

To do so, we will simply create the unit:

```
MinMaxScaler regularizer;
```

. . . and then *insert it* in the pipeline between the incoming photocell signal and the output LED:

```
photoCell >> regularizer >> led;
```

The above expression will do the following, in order:

1. Read the raw photocell value using the `photoCell` unit.
2. Send that raw value from the `photoCell` unit to the `regularizer` unit.
3. The `regularizer` unit updates itself if the value is a new extreme value (minimum or maximum).
4. The `regularizer` then remaps the raw photocell value to the full range of [0, 1] and sends it to the `led` unit.
5. The `led` unit takes the input value in [0, 1] and applies it to the intensity of the LED.

1.4.3 Step 3 : Reacting to Signal Changes

Remember our example from *ealier* where we were trying to detect high-valued signals using arbitary numbers?

```
if (value > 716)
  // do something
```

Suppose that instead of directly controlling the LED value based on the photocell's value, we instead want to react to abrupt changes in the photocell's value by triggering the LED? In other words, we would like to detect **peaks** in the incoming signal (such as when someone points a light source towards the photocell).

One first way to do so would be to pick a threshold in the regularized signal above which we would react to the light source. Let's say that we will react when the signal goes above 70%. The code of the `step()` function now becomes:

```
void step() {
  photoCell >> regularizer;
  if (regularizer > 0.7)
    1 >> led;
  else
    0 >> led;
}
```

. . . which can be more compactly rewritten by sending directly the conditional expression `(regularizer > 0.7)` to the output LED:

```
void step() {
  photoCell >> regularizer;
  (regularizer > 0.7) >> led;
}
```

1.4.4 Step 4 : Adapting to Changing Conditions

So far so good. The number 0.7 is still a hand-picked number but it makes more sense than 716 because it refers to a more human-understandable concept (70% instead of 716 / 1023). However, it will still be sensitive to changes in the ambient light and behave differently under different light conditions. In other words, it might work as expected in the morning, but might start working less well in the afternoon.

One first thing we could do would be to make sure our regularization unit adapts to changing conditions. In order to do this, rather than having our MinMaxScaler remaps values depending on every single incoming values ever seen, we can have it adapt over a **time window**. This will allow our regularizer to slowly forget what it has learnt and reprogram itself after a certain amount of time has passed.

This can be accomplished by calling the `timeWindow(seconds)` function inside the `begin()` function:

```
void begin() {
  // Allow regularizer to adapt over an approximate period of 1 hour (3600 s).
  regularizer.timeWindow(3600.0f);
}
```

1.4.5 Step 5 : Detecting Outliers

The MinMaxScaler is a very useful unit for making sure signals stay within a [0, 1] range. However, it is not always the best for signal detection since it only accounts for extreme values (minimum and maximum) which makes it sensitive to rare events, and someone switching the lights off and on again might completely ruin the show.

A better alternative is the *Normalizer* unit, which regularizes incoming signals by normalizing them around a target **mean** by taking into account **standard deviation**. Once the data is normalized, extreme **outlier** values can be more easily and robustly detected based on how much they diverge from the mean.

Let's replace our MinMaxScaler by a Normalizer unit:

```
Normalizer regularizer;
```

… and let's use the `isHighOutlier()` function to find values that are higher than usual:

```
void step() {
  photoCell >> regularizer;
  regularizer.isHighOutlier(photoCell) >> led;
}
```

Note: By default, the `isHighOutlier()` function detects values that are more than 1.5 deviations from the mean. The function can be made more or less sensitive by adjusting the number of deviations (typically between 1.0 and 3.0). For example, `isHighOutlier(value, 1.2)` will be more sensitive, `isHighOutlier(value, 2.5)` will be less sensitive, and `isHighOutlier(value, 3.0)` will only respond to rarely-occuring extremes. While these numbers (1.2, 1.5, 2.5, etc.) still need to be hand-picked, they are much more robust than our 716 and even to our 0.7 number from earlier.

Here is a complete version of the code:

```
#include <Plaquette.h> // include the Plaquette library

// Create input unit for photocell.
AnalogIn photoCell(A0);

// Create output unit for LED.
AnalogOut led(9);

// Create regularization object.
Normalizer regularizer;

// Initialize everything.
void begin() {
  // Allow regularizer to adapt over an approximate period of 1 hour (3600 s).
  regularizer.timeWindow(3600.0f);
```

(continues on next page)

(continued from previous page)

```
}

// Define frame-by-frame operations.
void step() {
  // Update regularizer with raw signal value.
  photoCell >> regularizer;

  // Detect outliers and send the value (1=true=outlier, 0=false=no outlier)
  // directly to the LED.
  regularizer.isHighOutlier(photoCell) >> led;
}
```

1.4.6 Step 6 : Detecting Peaks

The outlier detection method is useful to find extreme values. However, it also comes with an important limitation. The `isHighOutlier()` and `isOutlierLow()` methods return `true` *as long as* the received value is considered to be an outlier, making these methods unsuitable to trigger instantanous events such as toggling the status of an LED, starting a sound event, activating a motor, etc.

The *PeakDetector* unit addresses this limitation. It is best used in combination with a Normalizer unit. We will use the default mode of the PeakDetector (`PEAK_MAX`): for a peak to be detected, the signal will need to (1) cross a *trigger threshold* value (`triggerThreshold`); (2) reach its *apex* (max); and (3) *fall back* by a certain proportion (%) between the threshold and the apex (controlled by the `fallbackTolerance` parameter).

Building on the previous section for outlier detection, we will assign the PeakDetector's `triggerThreshold` to the value above which a value is considered to be a high outler, which can be obtained by calling the Normalizer's function `highOutlierThreshold()`:

```
PeakDetector detector(normalizer.highOutlierThreshold());
```

Note: As for the `isHighOutlier()` function, the `highOutlierThreshold()` function is set to return, by default, a threshold that is 1.5 standard deviations from the mean. The function can be made more or less sensitive by adjusting the number of deviations. For example, `highOutlierThreshold(1.2)` will be more sensitive, while `highOutlierThreshold(2.5)` will be less sensitive.

Finally, let's rewrite the `step()` function with our new peak detector, so that only when a **peak** is detected will the LED change state:

```
void step() {
  // Signal is normalized and sent to peak detector.
  sensor >> normalizer >> detector;

  // Toggle LED when peak detector triggers.
  if (detector)
    led.toggle();
}
```

The PeakDetector unit offers many options to fine-tune the peak detection process. Please read the *full documentation of the unit* for details.

1.5 Advanced Usage

1.5.1 Avoiding Plaquette Style

If you don't want to use Plaquette's >> operator, or Plaquette's auto-conversion of units to values (eg., `if (input)`, `input >> output`), you can avoid these features by simply using Plaquette units's `get()` and `put()` methods.

The `get()` method returns the current value of the unit:

```
float get()
```

The `put()` method sends a value to the unit and then returns the current value of the unit (the same that would be returned by `get()`):

```
float put(float value)
```

Additionally, digital input units such as *DigitalIn*, *Metro*, and *Timer*, have a `boolean isOn()` method that work for boolean `true/false` values. Additionally, digital output units such as *DigitalOut* have a `boolean putOn(boolean value)` method.

Here are some examples of how to adopt a classic object-oriented functions style instead of the Plaquette style.

Plaquette Style	Object-Oriented Style
`input >> output;`	`output.put(input.get());`
`digitalInput >> digitalOutput;`	`digitalOutput.putOn(digitalInput.isOn());`
`(2 * input) >> output;`	`output.put(2 * input.get());`
`!digitalInput >> digitalOutput;`	`digitalOutput.putOn(!digitalInput.isOn());`
`if (digitalInput)`	`if (digitalInput.isOn())`
`if (input < 0.4)`	`if (input.get() < 0.4)`
`input >> filter >> output;`	`output.put(filter.put(input.get()));`

1.5.2 Using Plaquette as an External Library

Seasoned Arduino coders might want to avoid rewriting their code using Plaquette's builtin `begin()` and `step()` functions, or they may want to include Plaquette's self-updating loop in a timer interrupt function. It is possible to do so by including the file `PlaquetteLib.h` instead of `Plaquette.h`.

One is then responsible for calling `Plaquette.begin()` at the beginning of the `setup()` function and to call `Plaquette.step()` at the beginning of the `loop()` function or inside the interrupt.

Here is an example of our blinking code rewritten using this feature:

```
#include <PlaquetteLib.h>

using namespace pq;

DigitalOut myLed(13);

SquareOsc myOsc(2.0, 0.5);

void setup() {
  Plaquette.begin();
}
```

(continues on next page)

(continued from previous page)

```
void loop() {
  Plaquette.step();
  myOsc >> myLed;
}
```

1.6 Credits

Developers:

- Sofian Audry • Website • GitHub
- Thomas Ouellet Fredericks • Website • GitHub

Contributors:

- Logo: Ian Donnelly • Website
- Code: Matthew Loewen • Website <https://www.mloewen.com/>`__`GitHub

Plaquette's base source code was produced as part of a research project at labXmodal. A special thanks to Chris Salter for his support.

Plaquette borrows ideas from the Arduino, ChucK, mbed, Processing, and Pure Data.

1.7 License

Plaquette is distributed under the Gnu General Public License v 3.0.

The text of the Plaquette documentation is licensed under a Creative Commons Attribution-ShareAlike 3.0 License. Parts of the text was copied and/or adapted from the Arduino documentation. Code samples in the guide are released into the public domain.

The Plaquette documentation is licensed under a Creative Commons Attribution-Share Alike 3.0 License. Parts of the documentation has been borrowed and/or adapted from the Arduino Reference and from the Processing Reference texts.

1.8 Base Units

Basic input-output units.

1.8.1 AnalogIn

An analog (ie. continuous) input unit that returns values between 0 and 1 (ie. 0% and 100%).

The unit is assigned to a specific `pin` on the board.

The `mode` specifies the behavior of the component attached to the pin:

- in `ANALOG_DEFAULT` mode (default) the value is expressed as a percentage of the reference voltage (Vref, typically 5V)
- in `ANALOG_INVERTED` mode the value is inverted (ie. 0V corresponds to 100% while 2.5V corresponds to 50%).

Example

Control an LED using a potentiometer.

```
#include <Plaquette.h>

AnalogIn potentiometer(A0);

AnalogOut led(9);

SineOsc oscillator;

void begin() {}

void step() {
  // The analog input controls the frequency of the LED's oscillation.
  oscillator.frequency(potentiometer.mapTo(2.0, 10.0));
  oscillator >> led;
}
```

Reference

class **AnalogIn** : public Node, public PinUnit, public Smoothable

A generic class representing a simple analog input.

Public Functions

AnalogIn(uint8_t pin = A0, uint8_t mode = ANALOG_DEFAULT)

Constructor.

Parameters

- **pin** – the pin number
- **mode** – the mode (ANALOG_DEFAULT or ANALOG_INVERTED)

inline virtual float **get**()

Returns value in [0, 1].

virtual float **mapTo**(float toLow, float toHigh)

Maps value to new range.

inline uint8_t **pin**() const

Returns the pin this component is attached to.

inline uint8_t **mode**() const

Returns the mode of the component.

inline virtual void **smooth**(float smoothTime = PLAQUETTE_DEFAULT_SMOOTH_WINDOW)

Apply smoothing to object.

inline virtual void **noSmooth**()

Remove smoothing.

inline virtual void **cutoff**(float hz)

Changes the smoothing window cutoff frequency (expressed in Hz).

inline float **cutoff**() const
Returns the smoothing window cutoff frequency (expressed in Hz).

> **Warning:** If the analog input pin is not connected to anything, the value returned by `get()` will fluctuate based on a number of factors (e.g. the values of the other analog inputs, how close your hand is to the board, etc.).

See Also

- *AnalogOut*
- *DigitalIn*

1.8.2 AnalogOut

An analog (ie. continuous) output unit that takes a value between 0 and 1 (ie. 0% and 100%).

The unit is assigned to a specific `pin` on the board.

The `mode` specifies the behavior of the component attached to the pin:

- in `SOURCE` mode (default) the pin acts as the source of current and the value is expressed as a percentage of the maximum voltage (Vcc, typically 5V)
- in `SINK` mode the component the source of current is external and should be equal to Vcc

Example

```
AnalogOut led(9);

void begin() {
  led.put(0.5);
}

void step() {
  // The LED value is changed randomly by a tiny amount (random walk).
  // Mutliplying by samplePeriod() makes sure the rate of change stays stable.
  (led + randomFloat(-0.1, 0.1) * samplePeriod()) >> led;
}
```

Reference

class **AnalogOut** : public AnalogSource, public PinUnit
A generic class representing a simple PWM output.

Public Functions

AnalogOut(uint8_t pin = 9, uint8_t mode = SOURCE)
Constructor.

Parameters

- **pin** – the pin number
- **mode** – the mode (SOURCE or SINK)

virtual float **put**(float value)
Pushes value into the component and returns its (possibly filtered) value.

inline virtual void **invert**()
Inverts value by calling `put(1-get())` (eg. 0.2 becomes 0.8).

inline virtual float **get**()
Returns value in [0, 1].

inline uint8_t **pin**() const
Returns the pin this component is attached to.

inline uint8_t **mode**() const
Returns the mode of the component.

Note: On most Arduino boards analog outputs rely on Pulse Width Modulation (PWM). After a call to `put(value)`, the pin will generate a steady square wave of the specified duty cycle until the next call to `put()` on the same pin. The frequency of the PWM signal on most pins is approximately 490 Hz. On the Uno and similar boards, pins 5 and 6 have a frequency of approximately 980 Hz.

Note: On most Arduino boards (those with the ATmega168 or ATmega328P), this functionality works on pins 3, 5, 6, 9, 10, and 11. On the Arduino Mega, it works on pins 2 - 13 and 44 - 46. Older Arduino boards with an ATmega8 only support `AnalogOut` on pins 9, 10, and 11. The Arduino DUE supports analog output on pins 2 through 13, plus pins DAC0 and DAC1. Unlike the PWM pins, DAC0 and DAC1 are Digital to Analog converters, and act as true analog outputs.

See Also

- *AnalogIn*
- *DigitalOut*

1.8.3 DigitalIn

A digital (ie. binary) input unit that can be either "on" or "off".

The unit is assigned to a specific `pin` on the board.

The `mode` specifies the behavior of the component attached to the pin:

- in `INTERNAL_PULLUP` mode (default) the internal 20K pullup resistor is used
- in `EXTERNAL_PULLUP` mode you need to use an external pullup resistor connected to Vcc
- in `EXTERNAL_PULLDOWN` mode you need to use an external pulldown resistor connected to GND

Debouncing

Some digital inputs such as push-buttons often generate spurious open/close transitions when pressed, due to mechanical and physical issues: these transitions called "bouncing" may be read as multiple presses in a very short time, fooling the program.

The DigitalIn object features debouncing capabilities which can prevent this kind of problems. Debouncing can be achieved using different modes: default (`DEBOUNCE_DEFAULT`), lock-out (`DEBOUNCE_LOCK_OUT`) and prompt-detect (`DEBOUNCE_PROMPT_DETECT`). For more information please refer to the documentation of the Bounce2 Arduino Library.

Example

Turns on and off a light emitting diode (LED) connected to digital pin 13, when pressing a pushbutton attached to digital pin 2.

```
#include <Plaquette.h>

DigitalIn button(2);

DigitalOut led(13);

void begin() {
  button.debounce(); // debounce button
}

void step() {
  // Toggle the LED each time the button is pressed.
  if (button.rose())
    led.toggle();
}
```

Reference

class **`DigitalIn`** : public DigitalSource, public PinUnit, public Debounceable

A generic class representing a simple digital input.

Public Functions

`DigitalIn`(uint8_t pin = 0, uint8_t mode = INTERNAL_PULLUP)

Constructor.

Parameters

- **`pin`** – the pin number
- **`mode`** – the mode (INTERNAL_PULLUP, EXTERNAL_PULLUP, or EXTERNAL_PULLDOWN)

inline virtual bool **`isOn`**()

Returns true iff the input is "on".

inline virtual bool **`rose`**()

Returns true if the value rose.

inline virtual bool **fell**()
    Returns true if the value fell.

inline virtual bool **changed**()
    Returns true if the value changed.

inline virtual int8_t **changeState**()
    Difference between current and previous value of the unit.

inline virtual bool **isOff**()
    Returns true iff the input is "off".

inline virtual int **getInt**()
    Returns value as integer (0 or 1).

inline virtual float **get**()
    Returns value as float (either 0.0 or 1.0).

inline uint8_t **pin**() const
    Returns the pin this component is attached to.

inline uint8_t **mode**() const
    Returns the mode of the component.

inline virtual void **debounce**(float debounceTime = PLAQUETTE_DEFAULT_DEBOUNCE_WINDOW)
    Apply smoothing to object.

inline virtual void **noDebounce**()
    Remove smoothing.

inline uint8_t **debounceMode**() const
    Returns the debounce mode.

inline void **debounceMode**(uint8_t mode)
    Sets debounce mode.

Parameters **mode** – the debounce mode (DEBOUNCE_DEFAULT, DEBOUNCE_LOCK_OUT or DEBOUNCE_PROMPT_DETECT)

See Also

- *AnalogIn*
- *DigitalOut*
- Bounce2 Arduino Library

1.8.4 DigitalOut

A digital (ie. binary) output unit that can be switched "on" or "off".

The unit is assigned to a specific `pin` on the board.

The `mode` specifies the behavior of the component attached to the pin:

- in `SOURCE` mode (default) the pin acts as the source of current and the component is "on" when the pin is "high" (Vcc)
- in `SINK` mode the source of current is external and the component is "on" when the pin is "low" (GND)

Example

Switches off an LED connected in "sink" mode after a timeout.

```
#include <Plaquette.h>

DigitalOut led(13, SINK);

void begin() {
  led.on();
}

void step() {
  // Switch the LED off after 5 seconds.
  if (seconds() > 5)
    led.off();
}
```

Reference

class **DigitalOut** : public DigitalSource, public PinUnit

A generic class representing a simple digital output.

Public Functions

DigitalOut(uint8_t pin = LED_BUILTIN, uint8_t mode = SOURCE)

Constructor.

Parameters

- **pin** – the pin number
- **mode** – the mode (SOURCE or SINK)

inline virtual bool **isOn**()

Returns true iff the input is "on".

inline virtual bool **toggle**()

Switches between on and off.

inline virtual bool **isOff**()

Returns true iff the input is "off".

inline virtual int **getInt**()

Returns value as integer (0 or 1).

inline virtual float **get**()

Returns value as float (either 0.0 or 1.0).

inline virtual bool **on**()

Sets output to "on" (ie. false, 0).

inline virtual bool **off**()

Sets output to "off" (ie. true, 1).

inline virtual float **put**(float value)

Pushes value into the unit.

Parameters `value` – the value sent to the unit

Returns the new value of the unit

inline uint8_t **pin**() const
Returns the pin this component is attached to.

inline uint8_t **mode**() const
Returns the mode of the component.

See Also

- *AnalogOut*
- *DigitalIn*

1.8.5 StreamIn

An input unit that can receive values transmitted through a stream – for example, the Arduino serial line. Values are sent in clear text and separated by newlines and/or carriage returns.

Example

Controls the value of a LED using serial. Try opening the serial monitor and sending values between 0 and 1.

```
#include <Plaquette.h>

StreamIn serialIn(Serial);

AnalogOut led(9);

void begin() {}

void step() {
  serialIn >> led;
}
```

To run this example:

1. Upload the code.
2. In the Arduino software open the serial monitor: **Tools > Serial Monitor**.
3. Make sure the default baudrate of **9600** bps is selected.
4. Make sure one of the options “Newline”, “Carriage return”, or “Both NL + CR” is selected.
5. Write a number between 0.0 and 1.0 and press “Enter”. This should allow you to set the LED intensity.
6. Try different values.

Reference

class **StreamIn** : public AnalogSource

Stream/serial input. Reads float values using Arduino built-in parseFloat().

Public Functions

StreamIn(Stream &stream = Serial)

Constructor.

Parameters **stream** – a reference to a Stream object

inline virtual float **get**()

Returns value in [0, 1].

See Also

- *AnalogIn*
- *DigitalIn*
- *StreamOut*
- Arduino serial
- Arduino streams

1.8.6 StreamOut

An output unit that transmits values through a stream – for example, the Arduino serial line. Values are sent in clear text and separated by newlines and/or carriage returns.

Example

Outputs the number of seconds to serial.

```
#include <Plaquette.h>

StreamOut serialOut(Serial);

void begin() {}

void step() {
  // Output the number of seconds
  seconds() >> serialOut;
}
```

To run this example:

1. Upload the code.
2. In the Arduino software open the serial monitor: **Tools > Serial Monitor**.
3. Make sure the default baudrate of **9600** bps is selected.

4. You should see the seconds increase.
5. Close the monitor and open serial plotter: **Tools > Serial Plotter**.
6. You should see a graphical representation of the seconds.
7. Replace the line in `step()` by: `sin(seconds()) >> serialOut` and upload. You should now see a sine wave signal in the serial plotter.

Reference

class **`StreamOut`** : public AnalogSource

Stream/serial output. Number of digits of precision is configurable.

Public Functions

`StreamOut`(Stream &stream = Serial)

Constructor.

Parameters **`stream`** – a reference to a Stream object

virtual float **`put`**(float value)

Pushes value into the unit.

Parameters **`value`** – the value sent to the unit

Returns the new value of the unit

virtual void **`precision`**(uint8_t digits)

Sets precision of the output.

Parameters **`digits`** – the number of digits to show after decimal point

inline virtual float **`get`**()

Returns value in [0, 1].

See Also

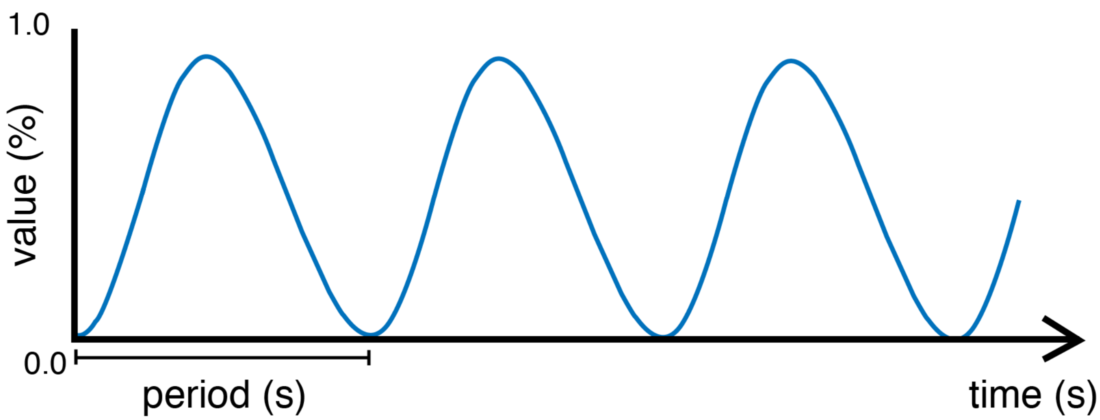
- *AnalogOut*
- *DigitalOut*
- *StreamIn*
- Arduino serial
- Arduino streams

1.9 Generators

Source units that generate different kinds of signals.

1.9.1 SineOsc

A source unit that can generate a sinusoid or sine wave. The signal is remapped to oscillate between 0 and 1 (rather than -1 and 1 as the traditional sine wave).



Example

Pulses an LED.

```
#include <Plaquette.h>

AnalogOut led(9);

SineOsc osc;

void begin() {
  osc.frequency(5.0); // frequency of 5 Hz
}

void step() {
  osc >> led;
}
```

class **SineOsc** : public Osc
Sine oscillator. Phase is expressed as % of period.

Public Functions

`SineOsc``(float period = 1.0f)`
Constructor.

Parameters **`period`** – the period of oscillation (in seconds)

virtual Node &**`period`**`(float period)`
Sets the period (in seconds).

Parameters **`period`** – the period of oscillation (in seconds)

Returns the unit itself

virtual Node &**`frequency`**`(float frequency)`
Sets the frequency (in Hz).

Parameters **`frequency`** – the frequency of oscillation (in Hz)

Returns the unit itself

virtual Node &**`amplitude`**`(float amplitude)`
Sets the amplitude of the wave.

Parameters **`amplitude`** – a value in [0, 1] that determines the amplitude of the wave (centered at 0.5).

Returns the unit itself

virtual Node &**`phase`**`(float phase)`
Sets the phase (ie.

the offset, in % of period).

Parameters **`phase`** – the phase (in % of period)

Returns the unit itself

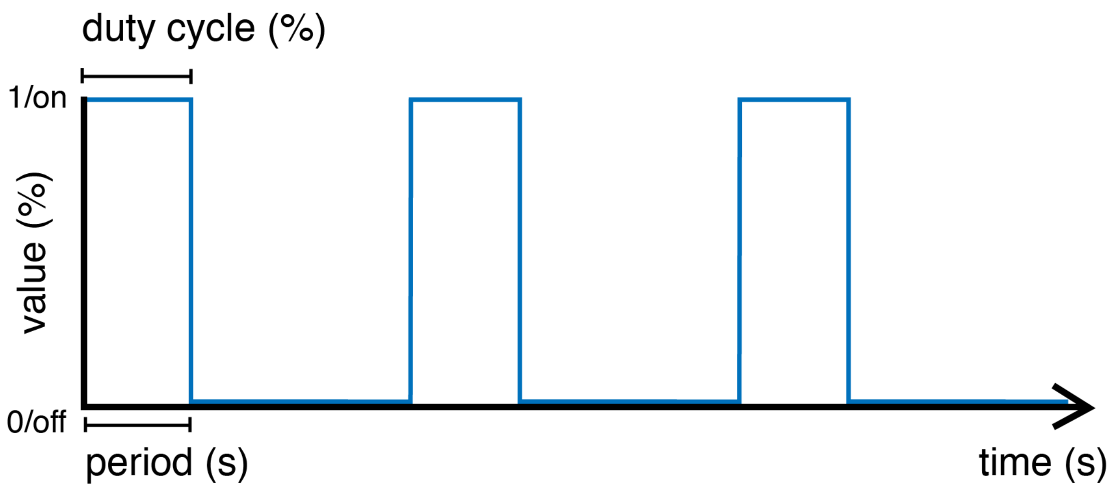
inline virtual float **`get`**`()`
Returns value in [0, 1].

See Also

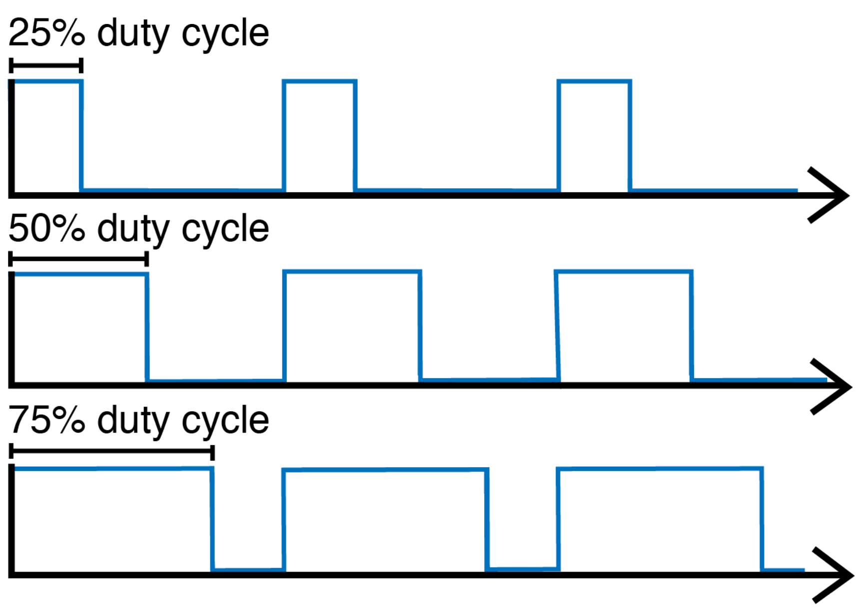
- *SquareOsc*
- *TriOsc*

1.9.2 SquareOsc

A source unit that generates a square wave signal. The signal can be tuned by changing the `period` and/or `frequency` of the oscillation, as well as the `duty cycle`.



The duty cycle represents the proportion of time (expressed as a percentage) in each cycle (period) during which the wave is "on".



Example

Makes the built-in LED blink with a period of 4 seconds. Because the duty cycle is set to 25%, the LED will stay on for 1 second and then off for 3 seconds.

```
#include <Plaquette.h>

DigitalOut led(13);

SquareOsc blinkOsc(4.0);

void begin() {
  blinkOsc.dutyCycle(0.25); // Sets the duty cycle to 25%
}

void step() {
  blinkOsc >> led;
}
```

class **SquareOsc** : public Osc

Square oscillator. Duty cycle is expressed as % of period.

Public Functions

SquareOsc(float period = 1.0f, float dutyCycle = 0.5f)

Constructor.

Parameters

- **period** – the period of oscillation (in seconds)
- **dutyCycle** – the duty-cycle as a value in [0, 1]

virtual *SquareOsc* &**dutyCycle**(float dutyCycle)

Sets the duty-cycle (ie.

the proportion of time during which the signal is on).

Parameters **dutyCycle** – the duty-cycle as a value in [0, 1]

Returns the unit itself

virtual Node &**period**(float period)

Sets the period (in seconds).

Parameters **period** – the period of oscillation (in seconds)

Returns the unit itself

virtual Node &**frequency**(float frequency)

Sets the frequency (in Hz).

Parameters **frequency** – the frequency of oscillation (in Hz)

Returns the unit itself

virtual Node &**amplitude**(float amplitude)

Sets the amplitude of the wave.

Parameters **`amplitude`** – a value in [0, 1] that determines the amplitude of the wave (centered at 0.5).

Returns the unit itself

virtual Node &**`phase`**(float phase)

Sets the phase (ie.

the offset, in % of period).

Parameters **`phase`** – the phase (in % of period)

Returns the unit itself

inline virtual float **`get`**()

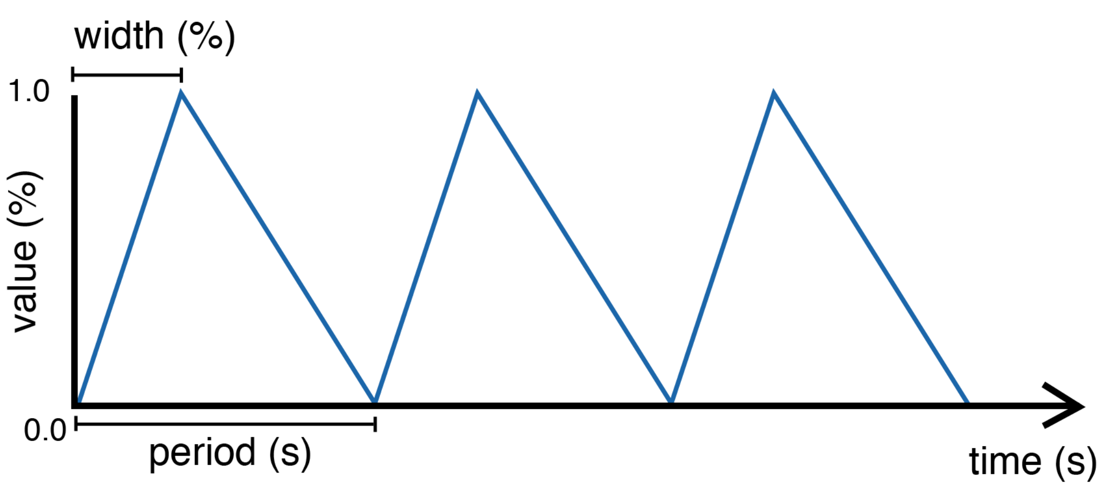
Returns value in [0, 1].

See Also

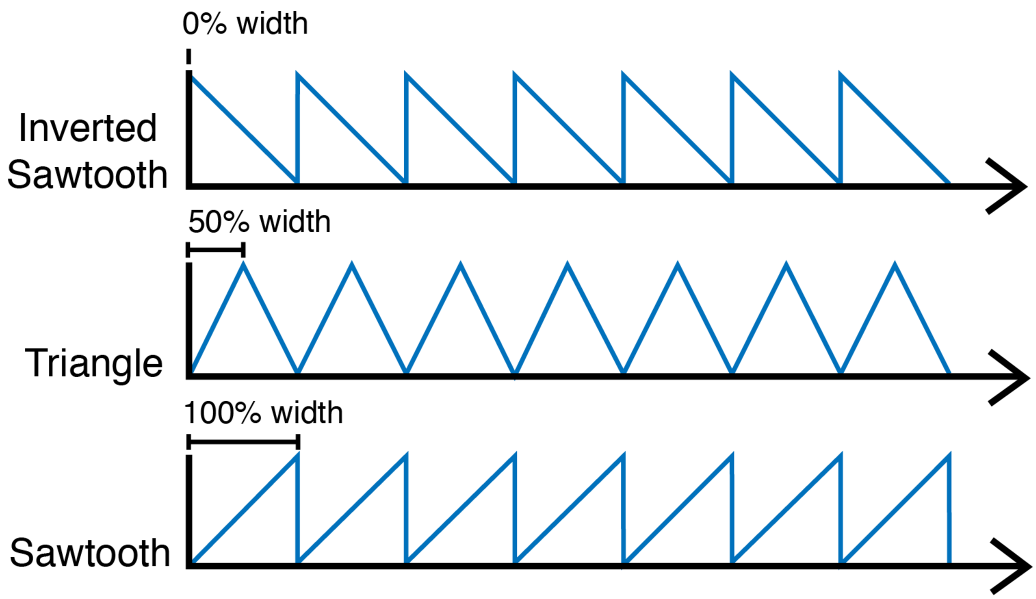
- *SineOsc*
- *TriOsc*

1.9.3 TriOsc

A source unit that can generate a range of triangle-shaped signals such as the triangle wave and the sawtooth wave. The signal can be adjusted by changing the `period` and/or `frequency` of the oscillation.



The `width` parameter represents the "turning point" during the period at which the signals reaches its maximum and starts going down again. Changing the width allows to generate different kinds of triangular-shaped waves. For example, by setting width to 1.0 (100%) one obtains a *sawtooth* wave; by setting it to 0.0 (0%) an *inverted sawtooth* is created; anything in between generates different flavors of *triangle* waves.



Example

Controls a set of traffic lights that go: red, yellow, green, red, yellow, green, and so on. It uses a sawtooth to iterate through these three states.

```
#include <Plaquette.h>

DigitalOut green(10);
DigitalOut yellow(11);
DigitalOut red(12);

TriOsc osc(10.0);

void begin() {
  osc.width(1.0); // sawtooth wave
}

void step() {
  // Shut down all lights.
  0 >> led >> yellow >> green;
  // Switch appropriate LED.
  if (osc < 0.4)
    green.on();
  else if (osc < 0.6)
    yellow.on();
  else
    red.on();
}
```

class **TriOsc** : public Osc

Triangle/sawtooth oscillator.

Public Functions

TriOsc(float period = 1.0f, float width = 0.5f)

Constructor.

Parameters

- **period** – the period of oscillation (in seconds)
- **width** – a value in [0, 1] that determines the point at which the wave reaches its maximum point (expressed as a fraction of the period)

virtual *TriOsc* &**width**(float width)

Sets the width of the wave.

Parameters **width** – a value in [0, 1] that determines the point at which the wave reaches its maximum point (expressed as a fraction of the period)

Returns the unit itself

virtual Node &**period**(float period)

Sets the period (in seconds).

Parameters **period** – the period of oscillation (in seconds)

Returns the unit itself

virtual Node &**frequency**(float frequency)

Sets the frequency (in Hz).

Parameters **frequency** – the frequency of oscillation (in Hz)

Returns the unit itself

virtual Node &**amplitude**(float amplitude)

Sets the amplitude of the wave.

Parameters **amplitude** – a value in [0, 1] that determines the amplitude of the wave (centered at 0.5).

Returns the unit itself

virtual Node &**phase**(float phase)

Sets the phase (ie.

the offset, in % of period).

Parameters **phase** – the phase (in % of period)

Returns the unit itself

inline virtual float **get**()

Returns value in [0, 1].

See Also

- *Ramp*
- *SineOsc*
- *SquareOsc*

1.10 Timing

Time-management source units.

1.10.1 Alarm

An alarm clock digital source unit. Counts time and becomes "on" when time is up. The alarm can be started, stopped, and resumed.

When started, the alarm stays "off" until it reaches its timeout duration, after which it becomes "on".

Example

Uses an alarm to change the state of built-in LED at random periods of time.

```
#include <Plaquette.h>

Alarm myAlarm(2.0); // an alarm with 2 seconds duration

DigitalOut led(13);

void begin() {
  myAlarm.start(); // start alarm
}

void step() {
  if (myAlarm) // the alarm will stay "on" until it is stopped or restarted
  {
    // Change LED state.
    led.toggle();

    // Restarts the timer with a random duration between 1 and 5 seconds.
    myAlarm.duration(randomFloat(1.0, 5.0));
    myAlarm.start();
  }
}
```

Reference

class **Alarm** : public DigitalNode, public AbstractTimer
Chronometer class which becomes "on" after a given duration.

Public Functions

virtual bool **isOn**()
True when time is up.

inline virtual bool **isOff**()
Returns true iff the input is "off".

inline virtual int **getInt**()
Returns value as integer (0 or 1).

inline virtual float **get**()
Returns value as float (either 0.0 or 1.0).

virtual void **start**()
Starts/restarts the chronometer.

virtual void **start**(float duration)
Starts/restarts the chronometer with specific duration.

virtual float **progress**() const
The progress of the timer process (in %).

virtual void **stop**()
Interrupts the chronometer.

virtual void **resume**()
Resumes process.

inline virtual float **elapsed**() const
The time currently elapsed by the chronometer (in seconds).

inline bool **isStarted**() const
Returns true iff the chronometer is currently running.

See Also

- *Metro*
- *Ramp*
- *Timer*
- *SquareOsc*

1.10.2 Metro

A metronome digital source unit. Emits an "on" signal at a regular pace.

Example

```
#include <Plaquette.h>

Metro myMetro(0.5); // a metronome with a half-second duration

DigitalOut led(13);

void begin() {
}

void step() {
  if (myMetro)
  {
    // Change LED state.
    led.toggle();
  }
}
```

Reference

class **Metro** : public DigitalSource
    Chronometer digital unit which emits 1/true/"on" for one frame, at a regular pace.

Public Functions

Metro(float period = 1.0f)
    Constructor.

Parameters **period** – the period of oscillation (in seconds)

virtual *Metro* &**period**(float period)
    Sets the period (in seconds).

Parameters **period** – the period of oscillation (in seconds)

Returns the unit itself

virtual *Metro* &**frequency**(float frequency)
    Sets the frequency (in Hz).

Parameters **frequency** – the frequency of oscillation (in Hz)

Returns the unit itself

virtual *Metro* &**phase**(float phase)
    Sets the phase (ie.

the offset, in % of period).

Parameters **phase** – the phase (in % of period)

Returns the unit itself

inline virtual bool **isOn**()
Returns true iff the input is "on".

inline virtual bool **rose**()
Returns true if the value rose.

inline virtual bool **fell**()
Returns true if the value fell.

inline virtual bool **changed**()
Returns true if the value changed.

inline virtual int8_t **changeState**()
Difference between current and previous value of the unit.

inline virtual bool **isOff**()
Returns true iff the input is "off".

inline virtual int **getInt**()
Returns value as integer (0 or 1).

inline virtual float **get**()
Returns value as float (either 0.0 or 1.0).

See Also

- *Ramp*
- *SquareOsc*
- *Timer*

1.10.3 Ramp

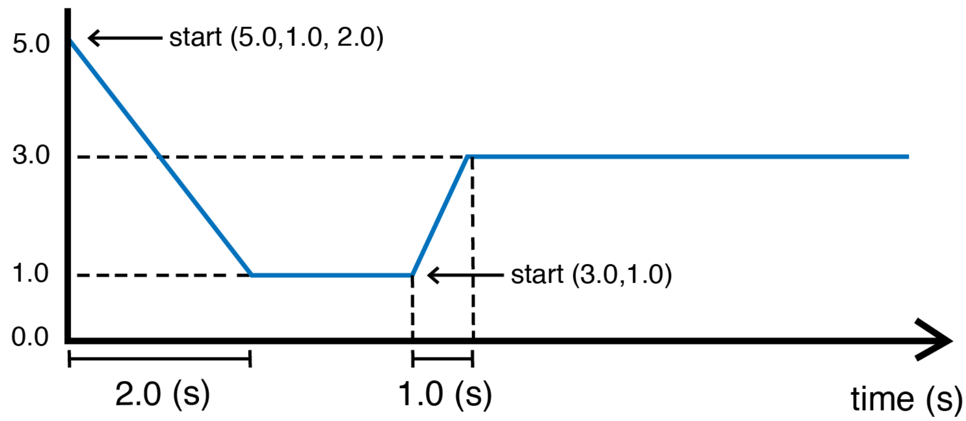
A source unit that generates a smooth transition between two values. The unit can be triggered to start transitioning to a target value for a certain duration.

There are two ways to start the ramp.

By calling `start(from, to, duration)` the ramp will transition from value `from` to value `to` in `duration` seconds.

Alternatively, calling `start(to, duration)` will start a transition from the ramp's current value to `to` in `duration` seconds.

The following diagram shows what happens to the ramp signal if `start(5.0, 1.0, 2.0)` is called, followed later by `start(3.0, 1.0)`:



Note: Ramps also support the use of easing functions in order to create different kinds of expressive effects with signals. An easing function can optionally be specified at the end of a `start()` command or by calling the `easing()` function.

Please refer to *this page* for a full list of available easing functions.

Example

Sequentially ramps through different values.

```
#include <Plaquette.h>

Ramp myRamp(0.0); // the ramp is initalized at zero (0)

StreamOut serialOut(Serial);

void begin() {
  // Apply an easing function (optional).
  myRamp.easing(easeOutSine);
}

void step() {
  if (myRamp.isComplete())
  {
    // Restarts the ramp going from current value to a random value in [-10, +10] in 2␣
↪seconds
    myRamp.start(randomFloat(-10, 10), 2.0);
  }
```

(continues on next page)

(continued from previous page)

```
  myRamp >> serialOut;
}
```

Reference

class **Ramp** : public Node, public AbstractTimer

Provides a ramping / tweening mechanism that allows smooth transitions between two values.

Public Functions

Ramp(float from = 0.0f)

Basic constructor.

Use one of the start(…) functions to launch ramps with specific parameters.

Parameters **from** – the value the ramp starts with

Ramp(float from, float to, float duration, easing_function easing = easeNone)

Basic constructor.

Use one of the start(…) functions to launch ramps with specific parameters.

Parameters

- **from** – the initial value
- **to** – the final value
- **duration** – the duration of the ramp (in seconds)
- **easing** – the easing function to apply (default: no easing)

inline virtual float **get**()

Returns value of ramp.

void **easing**(easing_function easing)

Sets easing function to apply to ramp.

Parameters **easing** – the easing function

inline void **noEasing**()

Remove easing function (linear/no easing).

virtual void **to**(float to)

Assign final value of the ramp starting from current value.

Parameters **to** – the final value

virtual void **fromTo**(float from, float to)

Assign initial and final values of the ramp.

Parameters

- **from** – the initial value
- **to** – the final value

virtual void **start**()

Starts/restarts the ramp. Will repeat the last ramp.

virtual void **`start`**(float to, float duration, easing_function easing = 0)
Starts a new ramp, starting from current value.

Parameters

- **`to`** – the final value
- **`duration`** – the duration of the ramp (in seconds)
- **`easing`** – the easing function (optional)

virtual void **`start`**(float from, float to, float duration, easing_function easing = 0)
Starts a new ramp.

Parameters

- **`from`** – the initial value
- **`to`** – the final value
- **`duration`** – the duration of the ramp (in seconds)
- **`easing`** – the easing function (optional).

virtual void **`start`**(float duration)
Starts/restarts the chronometer with specific duration.

virtual float **`progress`**() const
The progress of the timer process (in %).

inline virtual bool **`isComplete`**() const
Returns true iff the chronometer has completed its process.

virtual void **`stop`**()
Interrupts the chronometer.

virtual void **`resume`**()
Resumes process.

inline virtual float **`elapsed`**() const
The time currently elapsed by the chronometer (in seconds).

inline bool **`isStarted`**() const
Returns true iff the chronometer is currently running.

See Also

- *Alarm*
- *Easings*
- *Metro*
- *Timer*
- *TriOsc*

1.10.4 Timer

A timer digital source unit that counts time. The timer can be started, stopped, and resumed.

When started, the timer goes from 0 to 1 through its duration.

Example

Uses a timer to change the duty cycle of a blinking LED, then restarts with a new random duration.

```
#include <Plaquette.h>

Timer myTimer(2.0); // a timer with 2 seconds duration

DigitalOut led(13);

SquareOsc osc(3.0); // a square oscillator with a 3 seconds period

void begin() {
  myTimer.start(); // start timer
}

void step() {
  // Adjust oscillator's duty cycle according to current timer progress.
  osc.dutyCycle(myTimer);

  // Apply oscillator to LED state.
  osc >> led;

  if (myTimer.isComplete()) // if the timer has completed its course
  {
    // Restarts the timer with a random duration between 1 and 5 seconds.
    myTimer.start(randomFloat(1.0, 5.0));
  }
}
```

Reference

class **Timer** : public Node, public AbstractTimer

Chronometer class which ramps from 0 to 1 in a given duration.

Public Functions

virtual float **get**()

Returns progress in [0, 1].

virtual void **start**()

Starts/restarts the chronometer.

virtual void **start**(float duration)

Starts/restarts the chronometer with specific duration.

inline virtual bool **isComplete**() const
: Returns true iff the chronometer has completed its process.

virtual void **stop**()
: Interrupts the chronometer.

virtual void **resume**()
: Resumes process.

inline virtual float **elapsed**() const
: The time currently elapsed by the chronometer (in seconds).

inline bool **isStarted**() const
: Returns true iff the chronometer is currently running.

See Also

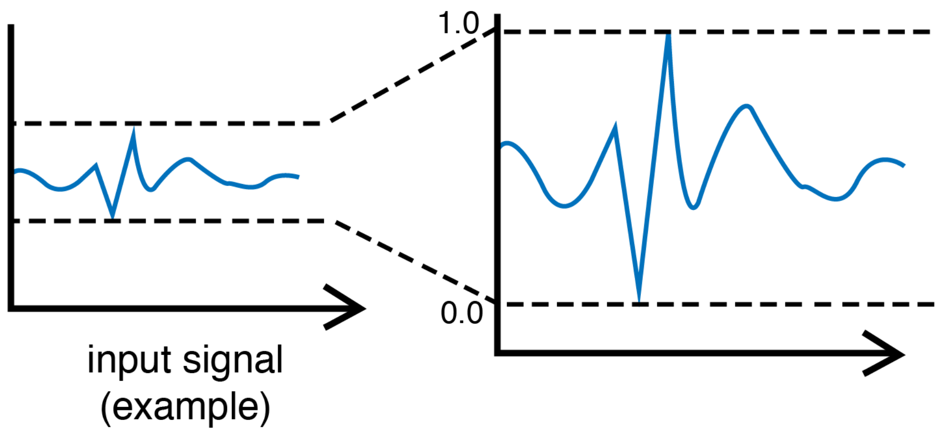
- *Alarm*
- *Metro*
- *Ramp*
- *TriOsc*

1.11 Filters

Filtering units for real-time signal processing.

1.11.1 MinMaxScaler

This filtering unit regularizes incoming signals by remapping them into a new interval of [0, 1]. It does so by keeping track of the minimum and the maximum values ever taken by the signal and rescales it such that the minimum value of the signal is mapped to 0 and the maximum value is mapped to 1.



In order to accomodate signals that might be changing through time, the user can specify a "decay time window" to control the rate of decay of the mininum and maximum boundaries. The principle is similar to the how the *Smoother* and the *Normalizer* make use of exponential moving average.

> **Warning:** This filtering unit works well as long as there are no "outliers" in the signal (ie. extreme values) that appear in rare conditions. Such values will replace the mininum or maximum value and greatly restrict the spread of the filtered values.
>
> There are three ways to prevent this:
>
> 1. Specifying a decay window using the `time(decayTime)` function.
> 2. Smoothing incoming values using the `smooth()` method or a *Smoother* unit before sending to the `MinMaxScaler`.
> 3. Using a regularization unit that is less prone to outliers such as the *Normalizer*.

Example

Reacts to high input values by activating an output LED. Scaler is used to automatically adapt to incoming sensor values.

```
#include <Plaquette.h>

AnalogIn sensor(A0);

MinMaxScaler scaler;

DigitalOut led(13);

void begin() {}
```

(continues on next page)

(continued from previous page)

```
void step() {
  // Rescale value.
  sensor >> scaler;

  // Light led on threshold of 80%.
  (scaler > 0.8) >> led;
}
```

Reference

class **MinMaxScaler** : public MovingFilter

Regularizes signal into [0,1] by rescaling it using the min and max values.

Public Functions

MinMaxScaler()

Constructor.

virtual void **infiniteTimeWindow**()

Sets time window to infinite.

virtual void **timeWindow**(float seconds)

Changes the time window (expressed in seconds).

virtual float **timeWindow**() const

Returns the time window (expressed in seconds).

virtual bool **timeWindowIsInfinite**() const

Returns true if time window is infinite.

virtual void **reset**()

Resets the moving filter.

virtual float **put**(float value)

Pushes value into the unit.

If *isStarted()* is false the filter will not be updated but will just return the filtered value.

Parameters **value** – the value sent to the unit

Returns the new value of the unit

virtual void **start**()

Starts calibration.

When calibration is started, calls to put(value) will return normalized value AND update the normalization statistics.

virtual void **stop**()

Stops calibration.

When calibration is stopped, calls to put(value) will return normalized value without updating the normalization statistics.

virtual bool **isStarted**() const

Returns true iff the statistics have already been started.

inline virtual float **get**()

Returns value in [0, 1].

See Also

- *Normalizer*
- *Smoother*

1.11.2 Normalizer

This filtering unit regularizes incoming signals by normalizing them around a target mean and standard deviation. It works by computing the normal distribution of the incoming data (mean and standard variation) and uses this information to re-normalize the data according to a different normal distribution (target mean and variance).

By default, the unit computes the mean and variance over all the data ever received. However, it can instead compute over a time window using an exponential moving average.

Example

Uses a normalizer to analyze input sensor values and detect extreme values.

```
#include <Plaquette.h>

// Analog sensor (eg. photocell or microphone).
AnalogIn sensor(A0);

// Creates a normalizer with mean 0 and standard deviation 1.
Normalizer normalizer(0, 1);

// Output indicator LED.
DigitalOut led(13);

void begin() {}

void step() {
  // Normalize value.
  sensor >> normalizer;

  // Light led if value differs from mean by more
  // than twice the standard deviation.
  (abs(normalizer) > 2.0) >> led;
}
```

Reference

class **Normalizer** : public MovingFilter, public MovingStats

Adaptive normalizer: normalizes values on-the-run using exponential moving averages over mean and standard deviation.

Public Functions

Normalizer()

Default constructor.

Will renormalize data around a mean of 0.5 and a standard deviation of 0.15.

Normalizer(float timeWindow)

Will renormalize data around a mean of 0.5 and a standard deviation of 0.15.

Parameters **smoothWindow** – specifies the approximate "time window" over which the normalization applies(in seconds)

Normalizer(float mean, float stdDev)

Constructor with infinite time window.

Parameters

- **mean** – the target mean
- **stdDev** – the target standard deviation
- **smoothWindow** – specifies the approximate "time window" over which the normalization applies(in seconds)

Normalizer(float mean, float stdDev, float timeWindow)

Constructor.

Parameters

- **mean** – the target mean
- **stdDev** – the target standard deviation
- **smoothWindow** – specifies the approximate "time window" over which the normalization applies(in seconds)

inline void **targetMean**(float mean)

Sets target mean of normalized values.

Parameters **mean** – the target mean

inline float **targetMean**() const

Returns target mean.

inline void **targetStdDev**(float stdDev)

Sets target standard deviation of normalized values.

Parameters **stdDev** – the target standard deviation

inline float **targetStdDev**() const

Returns target standard deviation.

virtual void **infiniteTimeWindow**()

Sets time window to infinite.

virtual void **timeWindow**(float seconds)
: Changes the time window (expressed in seconds).

virtual float **timeWindow**() const
: Returns the time window (expressed in seconds).

virtual bool **timeWindowIsInfinite**() const
: Returns true if time window is infinite.

virtual void **reset**()
: Resets the statistics.

virtual float **put**(float value)
: Pushes value into the unit.

  If *isStarted()* is false the filter will not be updated but will just return the filtered value.

  Parameters **value** – the value sent to the unit

  Returns the new value of the unit

virtual float **lowOutlierThreshold**(float nStdDev = 1.5f) const
: Returns value above which value is considered to be a low outler (below average).

  Parameters **nStdDev** – the number of standard deviations (typically between 1 and 3); low values = more sensitive

virtual float **highOutlierThreshold**(float nStdDev = 1.5f) const
: Returns value above which value is considered to be a high outler (above average).

  Parameters **nStdDev** – the number of standard deviations (typically between 1 and 3); low values = more sensitive

bool **isClamped**() const
: Return true iff the normalized value is clamped within reasonable range.

void **clamp**(float nStdDev = NORMALIZER_DEFAULT_CLAMP_STDDEV)
: Assign clamping value.

  Values will then be clamped between reasonable range (*targetMean()* +/- nStdDev * *targetStdDev()*).

  Parameters **nStdDev** – the number of standard deviations (default: 3.333333333)

void **noClamp**()
: Remove clamping.

virtual void **start**()
: Starts calibration.

  When calibration is started, calls to put(value) will return normalized value AND update the normalization statistics.

virtual void **stop**()
: Stops calibration.

  When calibration is stopped, calls to put(value) will return normalized value without updating the normalization statistics.

virtual bool **isStarted**() const
: Returns true iff the statistics have already been started.

inline virtual float **get**()
: Returns value in [0, 1].

virtual bool **`isOutlier`**(float value, float nStdDev = 1.5f) const
Returns true if the value is considered an outlier.

Parameters

- **`value`** – the raw value to be tested (non-normalized)
- **`nStdDev`** – the number of standard deviations (typically between 1 and 3); low values = more sensitive

Returns true if value is nStdDev number of standard deviations above or below mean

virtual bool **`isLowOutlier`**(float value, float nStdDev = 1.5f) const
Returns true if the value is considered a low outlier (below average).

Parameters

- **`value`** – the raw value to be tested (non-normalized)
- **`nStdDev`** – the number of standard deviations (typically between 1 and 3); low values = more sensitive

Returns true if value is nStdDev number of standard deviations below mean

virtual bool **`isHighOutlier`**(float value, float nStdDev = 1.5f) const
Returns true if the value is considered a high outlier (above average).

Parameters

- **`value`** – the raw value to be tested (non-normalized)
- **`nStdDev`** – the number of standard deviations (typically between 1 and 3); low values = more sensitive

Returns true if value is nStdDev number of standard deviations above mean

See Also

- *MinMaxScaler*
- *Smoother*

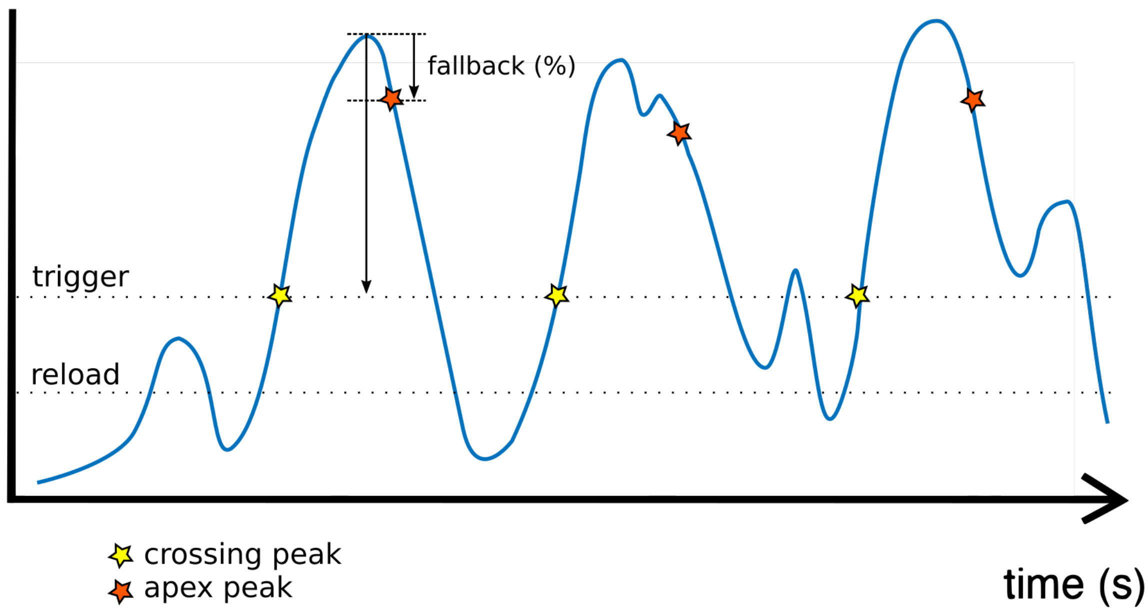
1.11.3 PeakDetector

This unit detects peaks (minima or maxima) in an incoming signal. Peaks are detected based on crossing a trigger threshold above (or below) which a peak is detected.

Two different ways are supported to do this:

- In **crossing** modes (`PEAK_RISING` and `PEAK_FALLING`) the peak is detected *as soon as the signal crosses* the `triggerThreshold`.
- In **apex** modes (`PEAK_MAX` and `PEAK_MIN`) the peak is detected after the signal crosses the `triggerThreshold`, reaches its apex, and then *falls back* by a certain proportion (%) between the threshold and the apex (controlled by the `fallbackTolerance` parameter).

In all cases, after a peak is detected, the detector will wait until the signal crosses back the `reloadThreshold` (which can be adjusted to control detection sensitivity) before it can be triggered again.



In summary, the four different modes available are:

- `PEAK_RISING` : peak detected as soon as `value >= triggerThreshold`, then wait until `value < reloadThreshold`
- `PEAK_FALLING` : peak detected as soon as `value <= triggerThreshold`, then wait until `value > reloadThreshold`
- `PEAK_MAX` : peak detected after `value >= triggerThreshold` and then *falls back* after peaking; then waits until `value < reloadThreshold`
- `PEAK_MIN` : peak detected after `value <= triggerThreshold` and then *falls back* after peaking; then waits until `value > reloadThreshold`

Note: Before sending a signal to a PeakDetector unit, it is recommended to normalize signals, preferably using the *Normalizer* unit. Furthermore, to avoid a noisy signal to generate false peaks, it is recommended to smooth the signal by calling the source unit's `smooth()` method or by using a *Smoother* unit.

Example

Uses a Normalizer and a PeakDetector to analyze input sensor values and detect peaks. Toggle and LED each time a peak is detected.

```
#include <Plaquette.h>

// Analog sensor (eg. photocell or microphone).
AnalogIn sensor(A0);

// Normalization unit to normalize values.
Normalizer normalizer;
```

(continues on next page)

(continued from previous page)

```
// Peak detector. Threshold is set at 1.5 standard deviations above normal.
PeakDetector detector(normalizer.highOutlierThreshold(1.5)); // default mode = PEAK_MAX
// NOTE: You can change mode using optional 2nd parameter, example:
// PeakDetector detector(1.5, PEAK_FALLING));

// Digital LED output.
DigitalOut led;

void begin() {
  // Adjust reload threshold to smaller value than reloadThreshold.
  detector.reloadThreshold(normalizer.highOutlierThreshold(1.0));

  // Adjust fallback tolerance as % between apex and trigger threshold.
  detector.fallbackTolerance(0.2); // 0.2 = 20% (default: 10%)

  // Smooth signal to avoid false peaks due to noise.
  sensor.smooth();

  // Set a time window of 1 minute (60 seconds) on normalizer.
  // This will allow the normalier to slowly readjust itself
  // if the lighting conditions change.
  normalizer.timeWindow(60.0f);
};

void step() {
  // Signal is normalized and sent to peak detector.
  sensor >> normalizer >> detector;

  // Toggle LED when peak detector triggers.
  if (detector)
    led.toggle();
}
```

Reference

class **PeakDetector** : public DigitalNode

Emits a signals when a signal peaks.

Public Functions

PeakDetector(float triggerThreshold, uint8_t mode = PEAK_MAX)

Constructor.

Possible modes are:

- PEAK_RISING : peak detected when value becomes >= triggerThreshold, then wait until it becomes < reloadThreshold (*)
- PEAK_FALLING : peak detected when value becomes <= triggerThreshold, then wait until it becomes > reloadThreshold (*)

- PEAK_MAX : peak detected after value becomes >= triggerThreshold and then falls back after peaking; then waits until it becomes < reloadThreshold (*)
- PEAK_MIN : peak detected after value becomes <= triggerThreshold and then rises back after peaking; then waits until it becomes > reloadThreshold (*)

Parameters

- **`triggerThreshold`** – value that triggers peak detection
- **`mode`** – peak detection mode

void **`triggerThreshold`**(float triggerThreshold)
Sets triggerThreshold.

inline float **`triggerThreshold`**() const
Returns triggerThreshold.

void **`reloadThreshold`**(float reloadThreshold)
Sets minimal threshold that "resets" peak detection in crossing (rising/falling) and peak (min/max) modes.

inline float **`reloadThreshold`**() const
Returns minimal value "drop" for reset.

void **`fallbackTolerance`**(float fallbackTolerance)
Sets minimal relative "drop" after peak to trigger detection in peak (min/max) modes, expressed as proportion (%) of peak minus triggerThreshold.

inline float **`fallbackTolerance`**() const
Returns minimal relative "drop" after peak to trigger detection in peak modes.

bool **`modeInverted`**() const
Returns true if mode is PEAK_FALLING or PEAK_MIN.

bool **`modeCrossing`**() const
Returns true if mode is PEAK_RISING or PEAK_FALLING.

void **`mode`**(uint8_t mode)
Sets mode.

inline uint8_t **`mode`**() const
Returns mode.

virtual float **`put`**(float value)
Pushes value into the unit.

Parameters **`value`** – the value sent to the unit

Returns the new value of the unit

inline virtual bool **`isOn`**()
Returns true iff the triggerThreshold is crossed.

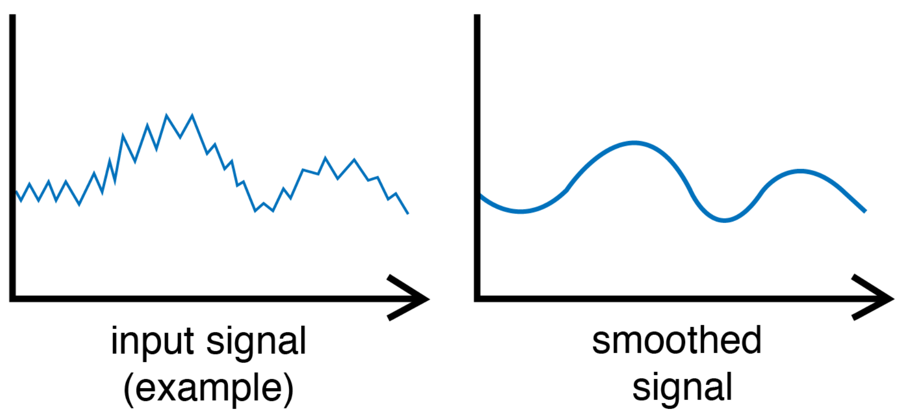
inline virtual float **`get`**()
Returns value as float (either 0.0 or 1.0).

See Also

- *Normalizer*
- *MinMaxScaler*
- *Smoother*

1.11.4 Smoother

Smooths the incoming signal by removing fast variations and noise (high frequencies).



Example

Smooth a sensor over time.

```
#include <Plaquette.h>

AnalogIn sensor(A0);

// Smooths over time window of 10 seconds.
Smoother smoother(10.0);

StreamOut serialOut(Serial);

void begin() {}

void step() {
  // Smooth value and send it to serial output.
```

(continues on next page)

(continued from previous page)

```
  sensor >> smoother >> serialOut;
}
```

Note: The filter uses an exponential moving average which corresponds to a form of low-pass filter.

Reference

class **Smoother** : public Node, public MovingAverage

Simple moving average transform filter.

Public Functions

Smoother(float smoothWindow = PLAQUETTE_DEFAULT_SMOOTH_WINDOW)

Constructor.

Parameters **factor** – a parameter in [0, 1] representing the importance of new values as opposed to old values (ie. lower smoothing factor means *more* smoothing)

virtual float **put**(float value)

Pushes value into the unit.

Parameters **value** – the value sent to the unit

Returns the new value of the unit

inline virtual float **get**()

Returns smoothed value.

void **timeWindow**(float seconds)

Changes the smoothing window (expressed in seconds).

inline float **timeWindow**() const

Returns the smoothing window (expressed in seconds).

void **cutoff**(float hz)

Changes the smoothing window cutoff frequency (expressed in Hz).

float **cutoff**() const

Returns the smoothing window cutoff frequency (expressed in Hz).

See Also

- *AnalogIn*
- *DigitalIn*

1.12 Functions

Standalone utility functions.

1.12.1 mapFloat()

Re-maps a number from one range to another. That is, a value of `fromLow` would get mapped to `toLow`, a value of `fromHigh` to `toHigh`, values in-between to values in-between, etc.

Does not constrain values to within the range, because out-of-range values are sometimes intended and useful. The `constrain()` function may be used either before or after this function, if limits to the ranges are desired.

Note that the "lower bounds" of either range may be larger or smaller than the "upper bounds" so the `mapFloat()` function may be used to reverse a range of numbers, for example

```
y = mapFloat(x, 1.0, 50.0, 50.0, 1.1);
```

The function also handles negative numbers well, so that this example

```
y = mapFloat(x, 1.0, 50.0, 50.0, -100.0);
```

is also valid and works well.

Unlike the Arduino map() function, `mapReal()` uses floating-point math and *will* generate fractions.

Example

```
#include <Plaquette.h>

SineOsc modulator(10.0);

SquareOsc oscillator(1.0);

DigitalOut led(13);

void begin() {
}

void step() {
  // Change frequency of oscillator between 2Hz and 15Hz.
  float freq = mapFloat(modulator, 0.0, 1.0, 2.0, 15.0);
  oscillator.frequency(freq);
  // Send to LED.
  oscillator >> led;
}
```

Reference

float pq::**mapFloat**(double value, double fromLow, double fromHigh, double toLow, double toHigh)

Re-maps a number from one range to another.

Parameters

- **value** – the number to map
- **fromLow** – the lower bound of the value's current range
- **fromHigh** – the upper bound of the value's current range
- **toLow** – the lower bound of the value's target range
- **toHigh** – the upper bound of the value's target range

Returns the mapped value

See Also

- *mapFrom01()*
- *mapTo01()*

1.12.2 mapFrom01()

Re-maps a number in the range [0, 1] to another range. That is, a value of 0 would get mapped to `toLow`, a value of 1 to `toHigh`, values in-between to values in-between, etc.

```
mapFrom01(x, toLow, toHigh)
```

is equivalent to:

```
y = mapFloat(x, 0, 1, toLow, toHigh)
```

See *mapFloat()* for more details.

Example

```
#include <Plaquette.h>

SineOsc modulator(10.0);

SquareOsc oscillator(1.0);

DigitalOut led(13);

void begin() {
}

void step() {
  // Change duty-cycle of oscillator in range [0.2, 0.8].
  oscillator.dutyCycle(mapFrom01(modulator, 0.2, 0.8));
  // Send to LED.
```

(continues on next page)

(continued from previous page)

```
  oscillator >> led;
}
```

Reference

float pq::**mapFrom01**(double value, double toLow, double toHigh)

Re-maps a number in range [0, 1] to a new range.

Parameters

- **value** – the number to map (in [0,1])
- **toLow** – the lower bound of the value's target range
- **toHigh** – the upper bound of the value's target range

Returns the mapped value in [toLow, toHigh]

See Also

- *mapFloat()*
- *mapTo01()*

1.12.3 mapTo01()

Re-maps a number to range [0, 1]. That is, a value of `fromLow` would get mapped to 0, a value of `fromHigh` to 1, values in-between to values in-between, etc.

```
mapTo01(x, fromLow, fromHigh)
```

is equivalent to:

```
y = mapFloat(x, fromLow, fromHigh, 0, 1)
```

See *mapFloat()* for more details.

Example

```
#include <Plaquette.h>

AnalogOut led(9);

void begin() {
}

void step() {
  // Generate a sinusoidal values between -1 and 1.
  float x = sin(seconds());
  // Remap to [0, 1] and send to LED.
  mapTo01(x, -1, 1) >> led;
}
```

Reference

float pq::**mapTo01**(double value, double fromLow, double fromHigh)

Re-maps a number to the [0, 1] range.

Parameters

- **value** – the number to map
- **fromLow** – the lower bound of the value's current range
- **fromHigh** – the upper bound of the value's current range

Returns the mapped value in [0, 1]

See Also

- *mapFloat()*
- *mapFrom01()*

1.12.4 randomFloat()

This function returns a random real-valued number.

Example

```
#include <Plaquette.h>

DigitalOut led(13);

void begin() {
}

void step() {
  // 2% probability to toggle the LED
  if (randomFloat() < 0.02)
    led.toggle();
}
```

Reference

float pq::**randomFloat**()

Generates a uniform random number in the interval [0,1).

float pq::**randomFloat**(float max)

Generates a uniform random number in the interval [0,max).

float pq::**randomFloat**(float min, float max)

Generates a uniform random number in the interval [min,max) (b>a).

See Also

- random()

1.12.5 seconds()

This function returns the number of seconds since the program started.

Example

```
#include <Plaquette.h>

DigitalOut led(13, SOURCE);

void begin() {
  led.off();
}

void step() {
  // Switch the LED on after 10 seconds.
  if (seconds() > 10)
    led.on();
}
```

Reference

float pq::**seconds**(bool referenceTime = true)

Returns time in seconds.

Optional parameter allows to ask for reference time (default) which will yield the same value through one iteration of step(), or "real" time which Will return the current total running time.

Parameters **referenceTime** – determines whether the function returns the reference time or the real time

Returns the time in seconds

See Also

- micros()
- millis()

1.13 Structure

Core structural functions and operators.

1.13.1 begin()

The `begin()` function is called when a sketch starts. Use it to initialize units, start using libraries, etc. The `begin()` function will only run once, after each powerup or reset of the board.

Note: Function `begin()` is the Plaquette equivalent of Arduino's setup(). However, Plaquette takes care of many of the initialization calls that need to be done in Arduino such as `pinMode()`. Therefore in many cases it will contain only a few calls or even be left empty.

Example

```
#include <Plaquette.h>

SquareOsc oscillator;
AnalogIn input(A0);

void begin() {
  oscillator.period(1.0);
  oscillator.dutyCycle(0.75);
  input.smooth();
}

void step() {
  // ...
}
```

See Also

- *step()*

1.13.2 step()

After creating a `begin()` function, which initializes and sets the initial values, the `step()` function does precisely what its name suggests, and performs one processing step that loops indefinitely as fast as possible, allowing your program to change and respond. Use it to actively control the board.

Note: Function `step()` is the Plaquette equivalent of Arduino's loop(). However, it is highly recommended that this function executes as fast as possible. Hence, one should performing computationally-intensive processing or calling blocking functions such as `delay()`

Example

```
#include <Plaquette.h>

DigitalIn button(2);

DigitalOut led(13);

void begin() {
}

void step() {
  button >> led;
}
```

See Also

- *begin()*

1.13.3 . (dot)

Provides access to an object's methods and data. An object is one instance of a class and may contain both methods (object functions) and data (object variables and constants), as specified in the class definition. The dot operator directs the program to the information encapsulated within an object.

Example

Switches LED on every 4 seconds.

```
#include <Plaquette.h>

DigitalOut led(13);

void begin() {
  led.off();
}

void step() {
  if (round(seconds()) % 4 == 0)
    led.on();
 else
    led.off();
}
```

Syntax

```
object.method()
object.variable
```

1.13.4 >> (pipe)

Sends data across units from left to right. This operator is specific to Plaquette and can be used in a chained manner.

The operation uses the `get()` and `put()` methods of units in such a way that:

```
input >> output;
```

is equivalent to:

```
output.put(input.get());
```

Numerical and boolean values can also be used:

```
12 >> output;
0.8 >> output;
true >> output;
```

Example

```
#include <Plaquette.h>

AnalogIn sensor(A0);

MinMaxScaler scaler;

AnalogOut led(9);

void begin() {}

void step() {
  // Rescale value and send the result to LED.
  sensor >> scaler >> led;
}
```

Syntax

```
input >> output
input >> filter >> output
```

1.14 Extra

Extra units and functions.

1.14.1 Easings

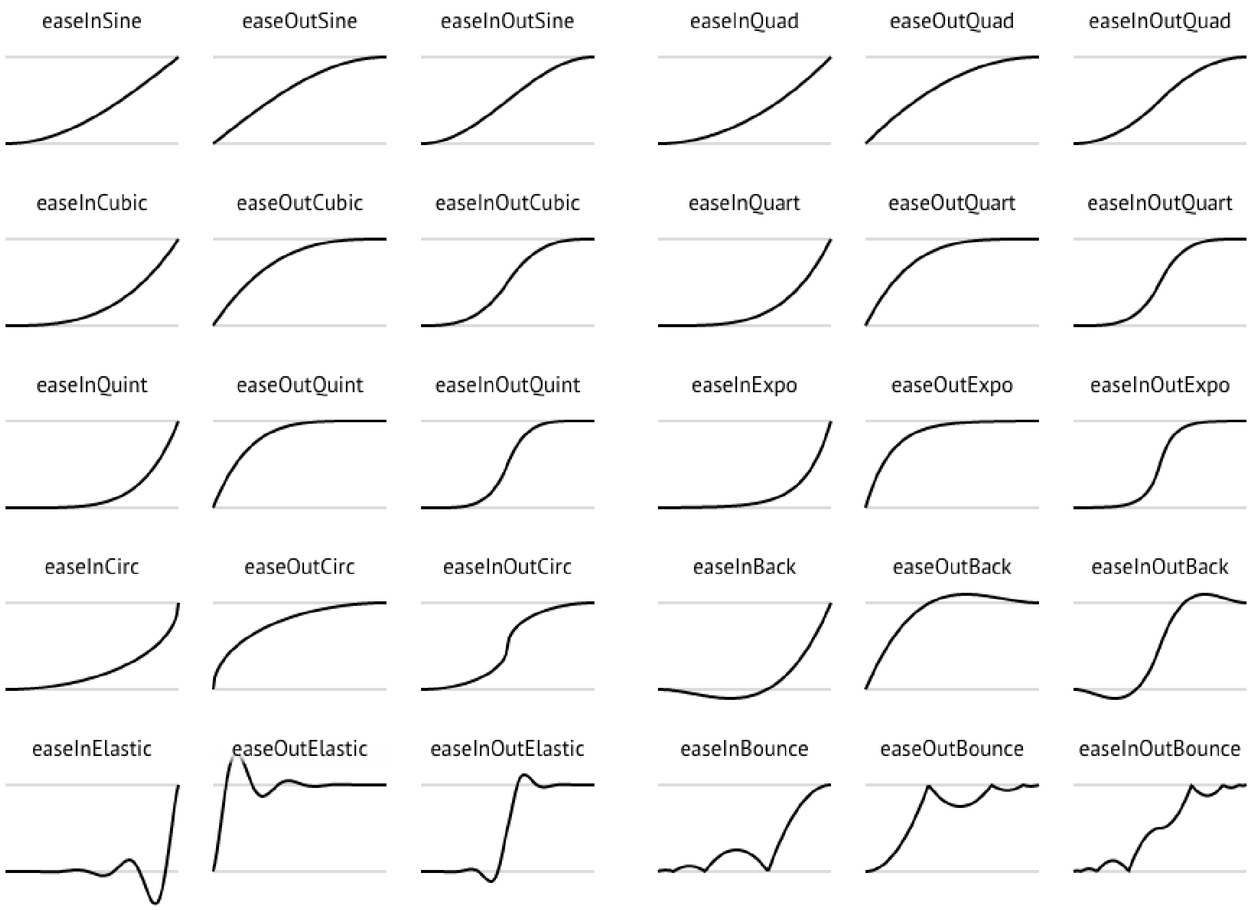
Easing functions apply non-linear effects to changing values, in order to create expressive real-time outputs. Plaquette provides users with a wide range of such functions, typically used with a Ramp unit.

All easing functions have the same signature:

```
float easeFunction(float t)
```

Value `x` should be in range [0, 1] and returned value is also in [0, 1].

This is the list of all easing functions (source: http://easings.net):



See Also

- *Ramp*

1.14.2 ContinuousServoOut

A source unit that controls a continous rotation servo-motor. A continuous servo-motor can move indefinitely forward or backwards.

Servo motors have three wires: power, ground, and signal. The power wire is typically red, and should be connected to the 5V pin on the Arduino board. The ground wire is typically black or brown and should be connected to a ground pin on the Arduino board. The signal pin is typically yellow, orange or white and should be connected to a digital pin on the Arduino board. Note that servos draw considerable power, so if you need to drive more than one or two, you'll probably need to power them from a separate supply (i.e. not the +5V pin on your Arduino). Be sure to connect the grounds of the Arduino and external power supply together.

Example

Everytime a button is pushed, the motor is stopped. Then upon button release it starts moving in the opposite direction.

```
#include <Plaquette.h>
#include <PqServo.h>

// The servo-motor output on pin 9.
ContinuousServoOut servo(9);

// The push-button.
DigitalIn button(2);

// Preserves the servo last speed value.
float lastValue = 0;

void begin() {
  // Debounce button.
  button.debounce();
  // Starts the servo.
  servo.put(1.0);
}

void step() {
  if (button) {
    // Save speed.
    lastValue = servo.get();
    // Stop servo.
    servo.stop();
  }
  else if (button.fell()) {
    // Reset speed.
    servo.put(lastValue);
    // ... then invert it.
    servo.reverse();
  }
}
```

class `ContinuousServoOut` : public AbstractServoOut
Continuous servo-motor.

Public Functions

`ContinuousServoOut`(uint8_t pin = 9)
Constructor for a continuous rotation servo-motor.

Parameters `pin` – the pin number

virtual void **stop**()
Stops the servo-motor.

virtual void **reverse**()
Sends servo-motor in reverse mode.

virtual float **put**(float value)
Pushes value into the unit.

Parameters `value` – the value sent to the unit

Returns the new value of the unit

inline uint8_t **pin**() const
Returns the pin this servomotor is attached to.

inline virtual float **get**()
Returns value in [0, 1].

See Also

- *AnalogOut*
- *ServoOut*

1.14.3 ServoOut

A source unit that controls a standard servo-motor.

Servo motors have three wires: power, ground, and signal. The power wire is typically red, and should be connected to the 5V pin on the Arduino board. The ground wire is typically black or brown and should be connected to a ground pin on the Arduino board. The signal pin is typically yellow, orange or white and should be connected to a digital pin on the Arduino board. Note that servos draw considerable power, so if you need to drive more than one or two, you'll probably need to power them from a separate supply (i.e. not the +5V pin on your Arduino). Be sure to connect the grounds of the Arduino and external power supply together.

Example

Sweeps the shaft of a servo motor back and forth across 180 degrees.

```
#include <Plaquette.h>
#include <PqServo.h>

// The servo-motor output on pin 9.
ServoOut servo(9);

// Oscillator to make the servo sweep.
SineOsc oscillator(2.0);

void begin() {
  // Position the servo in center.
  servo.center();
}

void step() {
  // Updates the value and send it back as output.
  oscillator >> servo;
}
```

class **ServoOut** : public AbstractServoOut
Standard servo-motor (angular).

Public Functions

ServoOut(uint8_t pin = 9)
Constructor for a standard servo-motor.

Parameters **pin** – the pin number

virtual float **putAngle**(float angle)
Sets the servomotor position to a specific angle between 0 and 180 degrees.

Parameters **angle** – the angle in degrees

Returns the current angle

virtual float **getAngle**()
Return the current angular angle in [0, 180].

inline virtual void **center**()
Re-centers the servo-motor.

virtual float **put**(float value)
Pushes value into the unit.

Parameters **value** – the value sent to the unit

Returns the new value of the unit

inline uint8_t **pin**() const
Returns the pin this servomotor is attached to.

inline virtual float **get**()
Returns value in [0, 1].

See Also

- *AnalogOut*
- *ContinuousServoOut*

INDEX

T