Plaquette Documentation

Release 0.6.0

Plaquette

Jan 08, 2025

GUIDE

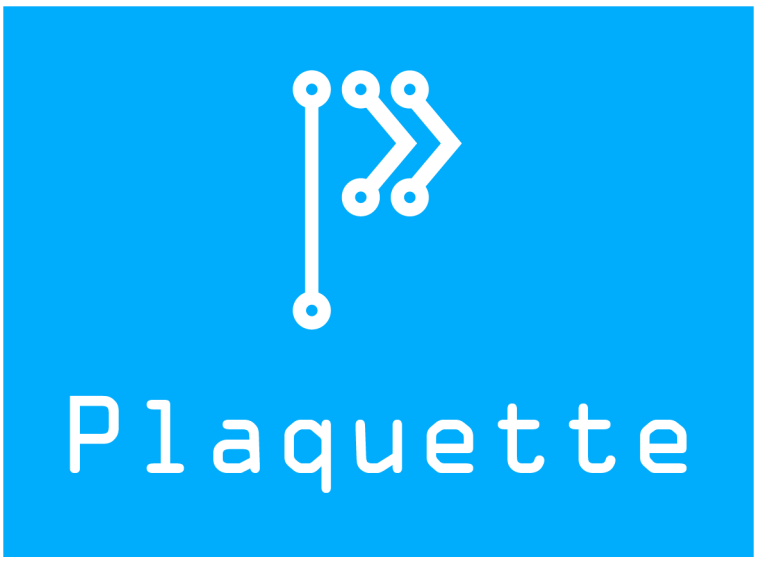
Plaquette is an object-oriented, user-friendly, signal-centric programming framework for **creative physical computing**. It promotes **expressiveness** over technical details while remaining fully compatible with Arduino, thus allowing **both beginner and advanced** creative practitioners to design 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*
Watch video tutorials	*Filter your signals*



CHAPTER

ONE

TABLE OF CONTENTS

1.1 Why Plaquette?

1.1.1 Rationale

Media creators such as media artists, interactive designers, digital luthiers, and electronic musicians work with real-time signals all the time. However, when working with tangible computing systems such as embedded sensors, robotics, connected objects, and electronic music instruments, available tools are often very low-level and lack expressivity. Creative practitionners thus struggle to design interesting works directly using such platforms.

Consider the following case of learning how to work with a simple lignt sensor (eg. photoresistor) connected to an Arduino board on analog pin 0. The code reads as follows:

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

The value that is read is a raw 10-bit value returned by the Arduino board's Analog to Digital Converter (ADC), an integer between 0 and 1023. But how is this value intuitively 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 would be to look at the value and compare it to a threshold:

```
if (value > 716) {
  ...
}
```

There are two problems with this approach.

Firstly, while it might work under certain lighting conditions, it will likely stop working if these conditions change, forcing us to make adjustments to the threshold value by hand.

Secondly, and perhaps more importantly, this piece of code does not really *express* 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 to detect a flash of light is whether the light signal is *significantly high compared to ambient light*.

What this example shows is that the way we are teaching and learning about sensor data is inefficient 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 detail 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 creative practitioners with accessible tools** that grasp high-level concepts such as "normalizing" and "detecting peaks" (rather than specific, arcane techniques on "how" to extract this information such as "FFT", "zero-crossing" or "Chebyshev filtering").
3. **Facilitate teamwork 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 are easily interchangeable 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 an LED.

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

1.2.2 User-friendly

Plaquette allows users to quickly design interactive systems using an expressive language that abstracts low-level functions. This allows both beginners and experts to create truly expressive code. For example, switching our LED object on or off 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) == HIGH)`	`if (button.isOn())`

1.2.3 Signal-centric

Plaquette helps designers manipulate real-time signals from inputs to outputs. In Plaquette, signals are represented either as `true/false` conditions (in the case of digital binary signals such as those coming from a button or switch), or as floating-point numbers 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.) Because of this, there is no more need for users to perform counter-intuitive conversions on integer values.

Arduino	Plaquette
Check if button is released:	
`if (digitalRead(2) != HIGH)`	`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 the right threshold for triggering an event based on input sensor input, one can use auto-normalizing *filters* such as *MinMaxScaler* and *Normalizer*.

Signals in Plaquette can easily flow between units, in a similar fashion to modern data-flow software such as Max, Pure Data, and TouchDesigner. While this can be achieved using function calls, Plaquette provides a special **piping operator** (>>) which allows data to be sent 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*. As such, the processing loop is 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 the following attempt to make an LED blink when pressing a button in Arduino, versus Plaquette's real-time approach:

Arduino	Plaquette

`int buttonPin = 2;`
`int ledPin = 12;`
 	
`void setup() {`
`  pinMode(buttonPin, OUTPUT);`
`  pinMode(ledPin, OUTPUT);`
`}`
 	
`void loop() {`
`  // Button is checked once per second.`
`  if (digitalRead(buttonPin) == HIGH) {`
`  digitalWrite(ledPin, HIGH);`
`  delay(500); // do nothing for 500ms`
`  digitalWrite(ledPin, LOW);`
`  delay(500); // do nothing for 500ms`
`  }`
`}`
	`DigitalIn button(2);`
	`DigitalOut led(12);`
	
	`// Square wave 1 second period.`
	`SquareWave oscillator(1.0);`
	
	`void begin() {}`
	
	`void step() {`
	`  // Button is checked at all time.`
	`  if (button)`
	`  oscillator >> led;`
	`}`

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.

The following example uses Arduino's constrain() function and Serial object to control a blinking LED that slows down with each button push. keeping the oscillation period within a certain range.

```
#include <Plaquette.h>

DigitalIn button(2);

DigitalOut led(LED_BUILTIN);

SquareWave oscillator(1.0);

int countButtonPushes = 0;

void begin() {}

void step() {

  if (Serial.read() == 'R') {
    countButtonPushes = 0;
  }

  if (button.rose()) { // true when value rises (ie. button is pushed)
    countButtonPushes++;
    oscillator.period( constrain(countButtonPushes, 2, 10) );
  }
```

(continues on next page)

(continued from previous page)

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

Warning

Plaquette needs the main processing loop to run continuously without interruption to work correctly. Users should thus **avoid using blocking processes** such as Arduino's delay() and delayMicroseconds() and functions in their code when using Plaquette.

Warning

Many of the core Arduino functions work with integer types such as `int` or `long` rather than floating-point types such as `float`. Plaquette provides alternative *functions* which should be used instead.

In particular, please use:

- *mapFloat()* instead of map()
- *randomFloat()* instead of random()
- *seconds()* instead of millis()

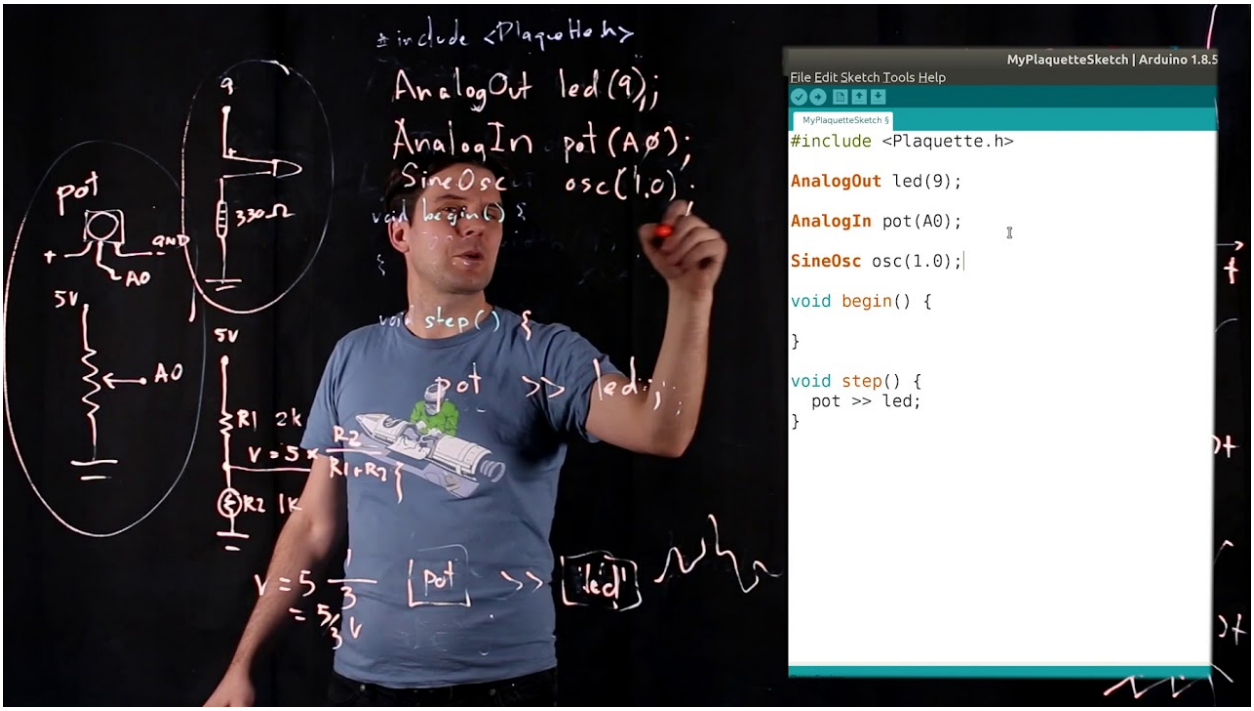
Warning

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

This short introduction will guide you through the first steps of using Plaquette.

We also recommend watching our introductory video tutorial series.



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 on your microcontroller 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 also exists other types of units, which will be described later. `DigitalOut` is a type of software unit that can represent one of the many hardware pins for

digital output on the Arduino board. One way to think about this is that the `DigitalOut` is a "virtual" version of the Arduino pin. These 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 hardware *pin* that 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`."

> **Note**
>
> Most Arduino boards have a pin connected to an on-board LED in series with a resistor and on most boards, this LED is connected to digital pin `13`. The constant `LED_BUILTIN` is the number of the pin to which the on-board LED is connected.

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 `SquareWave` unit type which generates a square wave oscillating between "on/high/one" and "off/low/zero" at a regular period of 2.0 seconds:

```
SquareWave myWave(2.0);
```

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). For our first program, we do not need 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 will send the signal generated by the `myWave` input unit to the `myLed` output unit. We will do this by using Plaquette's special >> operator:

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

In plain English, the statement `myWave >> myLed` reads as: "Take the value generated by `myWave` 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);

SquareWave myWave(2.0);

void begin() {}

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

1.3.3 Step 3 : Experiment!

So far so good. Let's see if we can push this a bit further.

Change initial parameters of a unit

The `SquareWave` unit type provides two parameters when it is created that allows you to configure the oscillator's behavior.

```
SquareWave myWave(period, width);
```

- `period` can be any positive number representing the period of oscillation (in seconds)
- `width` 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") (default: 0.5)

> **Note**
>
> We call this step the **construction** or **instantiation** of the object `myWave`.

Try changing the first parameter (period) in the square oscillator unit to change the period of oscillation.

- `SquareWave myWave(1.0);` for a period of one second
- `SquareWave myWave(2.5);` for a period of 2.5 seconds
- `SquareWave myWave(10.0);` for a period of 10 seconds
- `SquareWave myWave(0.5);` for a period of half a second (500 milliseconds)

> **Warning**
>
> Don't forget to re-upload the sketch after each change.

Now try adding a second parameter (width) to control the oscillator's width. For a fixed period, try changing the duty cycle to different percentages between 0.0 and 1.0.

- `SquareWave myWave(2.0, 0.5);` for a duty-cycle of 50% (default)
- `SquareWave myWave(2.0, 0.25);` for a duty-cycle of 25%
- `SquareWave myWave(2.0, 0.75);` for a duty-cycle of 75%

- `SquareWave myWave(2.0, 0.9);` for a duty-cycle of 90%

Change parameters of a unit during runtime

What if we wanted to change the parameters of the oscillator during runtime rather than just at the beginning? The `SquareWave` unit type allows real-time modification of its parameters using function calls using the `.` *(dot)* operator.

For example, to change the period, simply call the following inside the step() function:

```
void step() {
  myWave.period(newPeriod);
  myWave >> myLed;
}
```

Of course, to accomplish our goal, we need a way to *change* the value `newPeriod` during runtime. We can accomplish this in many different ways, but let's try something simple: we will use another wave to *modulate* our wave's period.

For this, we will be using another kind of source called a `SineWave` and will use its outputs to change the period of `myWave`.

```
SineWave myModulator(20.0);
```

This wave will oscillate smoothly from 0 to 1 every 20 seconds.

```
void step() {
  myWave.period(myModulator);
  myWave >> myLed;
}
```

Upload the sketch and you should see the LED blinking as before, with the difference that the blinking speed will now change from blinking very fast (in fast, infinitely fast, with a period of zero seconds!) to very slow (period of 20 seconds).

Note

If you want to visualize the values of both waves on your computer, you can print them on the serial port one after the other, separated by a space. Add the following code to your `step()` function:

```
print(myWave); print(" "); println(myModulator);
```

Then, launch the Arduino Serial Plotter by selecting in in **Tools > Serial Plotter**.

Now try modulating the width of `myWave` instead of its period:

```
myWave.width(myModulator);
```

Use a button

Now let's try to do some very simple interactivity by using a simple switch or button. For this we will be using the internal pull-up resistor available on Arduino boards for a very simple circuit. One leg of the button should be connected to ground (GND) while the other should be connected to digital pin 2.

Note

If you do not have a button or switch, you can just use two electric wires: one connected to ground (GND) and the other one to digital pin 2. When you want to press the button, simply touch the wires together to close the circuit.

Declare the button unit with the other units at the top of your sketch:

```
DigitalIn myButton(2, INTERNAL_PULLUP);
```

You will notice that the type of this unit (*DigitalIn*) resembles that of our LED-controlling unit (*DigitalOut*). This is because both units have something in common: they have only two states: either on or off, high or low, true or false, one or zero, hence the adjective `Digital`. However, while the LED is considered an output or actuator (`Out`) our button is rather an input or sensor (`In`).

Note

If you are curious, you might also want to know that there is an *AnalogIn* and an *AnalogOut* types which support sensors and actuators that work with continuous values between 0 and 1 (0% to 100%).

Now, let's use this button as a way to control whether the LED blinks or not. For this, we will need to use the value of the button as part of a **condition** for an if…else statement.

```
void step() {
  if (myButton)
    myWave >> myLed;
  else
    0 >> myLed;
}
```

Note

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

```
void step() {
  (myButton ? myWave : 0) >> myLed;
}
```

Full code

```
#include <Plaquette.h>

DigitalOut myLed(13);

SquareWave myWave(2.0);

SineWave myModulator(20.0);

DigitalIn myButton(2, INTERNAL_PULLUP);

void begin() {}

void step() {
```

(continues on next page)

(continued from previous page)

```
  myWave.period(myModulator);

  if (myButton)
    myWave >> myLed;
  else
    0 >> myLed;
}
```

More examples

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

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

We also recommend watching our introductory video tutorial series.

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 and 1023).
  int value = analogRead(photoCellPin);

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

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

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

1.4.1 Step 1 : Direct Input-to-Output

To begin, we will re-implement the example above, by using a 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 by 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() {
  // Send photocell value directly to the LED.
  photoCell >> led;
}
```

1.4.2 Step 2 : Getting the Full Range of the 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 possibility. 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 this 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 of 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 use this approach, 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 arbitrary 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 use sudden changes in the photocell's value to trigger the on/off state of the LED? In other words, we would like to work with the **peaks** in the incoming signal (such as when someone points a light source towards the photocell).

One 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 bit of an arbitrary, 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, this approach will still be sensitive to changes in the ambient light, and behave differently under different light conditions (for example, it might work as expected in the morning, but work less well in the late afternoon when the sun starts to go down.)

One thing we could do would be to make sure that our regularization unit adapts to changing conditions. In order to do this, rather than having our MinMaxScaler remap values depending on every single incoming value, we can have it adapt over a **time window**. This will allow our regularizer to slowly forget what it has learned, 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. Someone switching the lights on and off again rapidly 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 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 for triggering 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. In this mode, 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 outlier, 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 Vanilla Coding Style

You can avoid Plaquette's `>>` operator or auto-conversion of units to values (eg., `if (input)`, `input >> output`) in favor of a more conventional programming style by simply using the `get()` and `put()` functions of Plaquette units.

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*, *Metronome* have a `boolean isOn()` method that works for boolean `true/false` values, while 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`.

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

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

```
#include <PlaquetteLib.h>

using namespace pq;

DigitalOut myLed(13);

SquareWave myWave(2.0, 0.5);

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

void loop() {
  Plaquette.step();
```

(continues on next page)

(continued from previous page)

```
  myWave >> myLed;
}
```

1.6 Credits

Core Developers:

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

Contributors:

- Logo: Ian Donnelly • Website
- Code: Matthew Loewen • Website • GitHub
- Code: Samuel Favreau • Website
- Testing: Luana Belinsky • Website
- Documentation Editing: Erin Gee • Website • 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

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 `DIRECT` mode (default) the value is expressed as a percentage of the reference voltage (Vref, typically 5V)
- in `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);

SineWave 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 Unit, public PinUnit, public Smoothable

A generic class representing a simple analog input.

Public Functions

AnalogIn(uint8_t pin, uint8_t mode = DIRECT)

Constructor.

Parameters

- **pin** – the pin number
- **mode** – the mode (DIRECT or 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 **mode**(uint8_t mode)

Changes 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 converts a value between 0 and 1 (ie. 0% and 100%) into an analog voltage on one of the analog output pins.

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 source of current is external (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, uint8_t mode = DIRECT)

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.

inline virtual void **mode**(uint8_t mode)

Changes 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 DIRECT mode (default) the unit will be "on" when the voltage on the pin is high (Vref, typically 5V)
- in INVERTED mode the unit will be "on" when the voltage on the pin is low (GND)
- in INTERNAL_PULLUP mode the internal pullup resistor is used, simplifying usage of switches and buttons

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: stable (default) (DEBOUNCE_STABLE), 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. Pushbutton should be wired by connecting one side to pin 2 and the other to ground.

```
#include <Plaquette.h>

DigitalIn button(2, INTERNAL_PULLUP);

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, uint8_t mode = DIRECT)

Constructor.

Parameters

- **pin** – the pin number
- **mode** – the mode (DIRECT, INVERTED, or INTERNAL_PULLUP)

virtual void **mode**(uint8_t mode)

Changes the mode of the component.

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 void **onRise**(EventCallback callback)

Registers event callback on rise event.

inline virtual void **onFall**(EventCallback callback)

Registers event callback on fall event.

inline virtual void **onChange**(EventCallback callback)

Registers event callback on change event.

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, typically 5V)
- in `SINK` mode the source of current is external (Vcc) 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, uint8_t mode = DIRECT)

Constructor.

Parameters

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

virtual void **mode**(uint8_t mode)

Changes the mode of the component.

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 bool **toggle**()

Switches between on and off.

inline virtual int8_t **changeState**()

Difference between current and previous value of the unit.

inline virtual void **onRise**(EventCallback callback)

Registers event callback on rise event.

inline virtual void **onFall**(EventCallback callback)

Registers event callback on fall event.

inline virtual void **onChange**(EventCallback callback)

Registers event callback on change event.

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

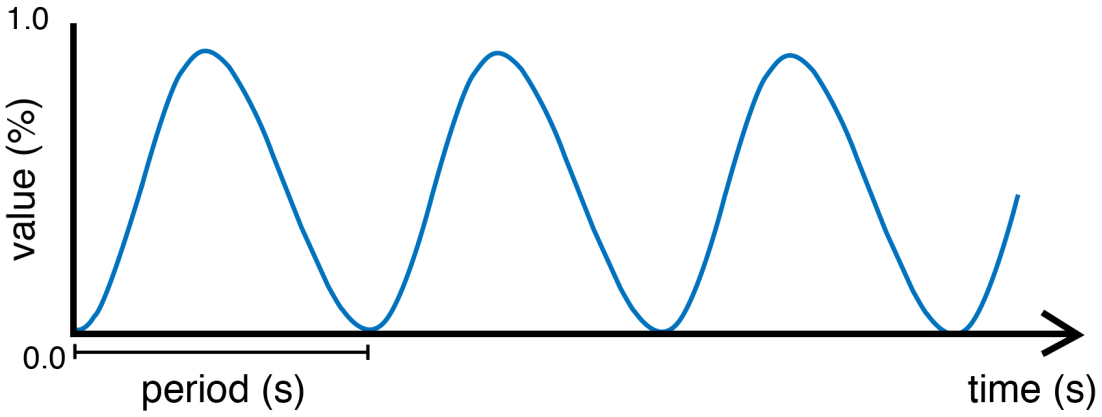
- *AnalogOut*
- *DigitalOut*
- *StreamIn*
- Arduino serial
- Arduino streams

1.9 Generators

Source units that generate different kinds of signals.

1.9.1 SineWave

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);

SineWave osc;

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

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

class **SineWave** : public AbstractWave

Sine oscillator. Phase is expressed as % of period.

Public Functions

virtual void **period**(float period)

Sets the period (in seconds).

Parameters
period – the period of oscillation (in seconds)

virtual void **frequency**(float frequency)

Sets the frequency (in Hz).

Parameters
frequency – the frequency of oscillation (in Hz)

inline virtual float **frequency**() const

Returns the frequency (in Hz).

virtual void **bpm**(float bpm)

Sets the frequency in beats-per-minute.

Parameters
bpm – the frequency of oscillation (in BPM)

inline virtual float **bpm**() const

Returns the frequency (in BPM).

virtual void **width**(float width)

Sets the width of the signal as a % of period.

Parameters
width – the width as a value in [0, 1]

inline virtual float **width**() const

Returns the width of the signal.

virtual void **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).

inline virtual float **amplitude**() const

Returns the amplitude of the wave.

virtual void **phase**(float phase)

Sets the phase (ie.

the offset, in % of period).

Parameters
phase – the phase (in % of period)

inline virtual float **phase**() const

Returns the phase (in % of period).

virtual float **shiftBy**(float phaseShift)

Returns oscillator's value with given phase shift (in %).

Supports negative phase shifts. Eg. shiftBy(0.2) returns future value of oscillator after 20% of its period would have passed.

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

Returns
the value of oscillator with given phase shift

inline virtual float **get**()

Returns value in [0, 1].

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

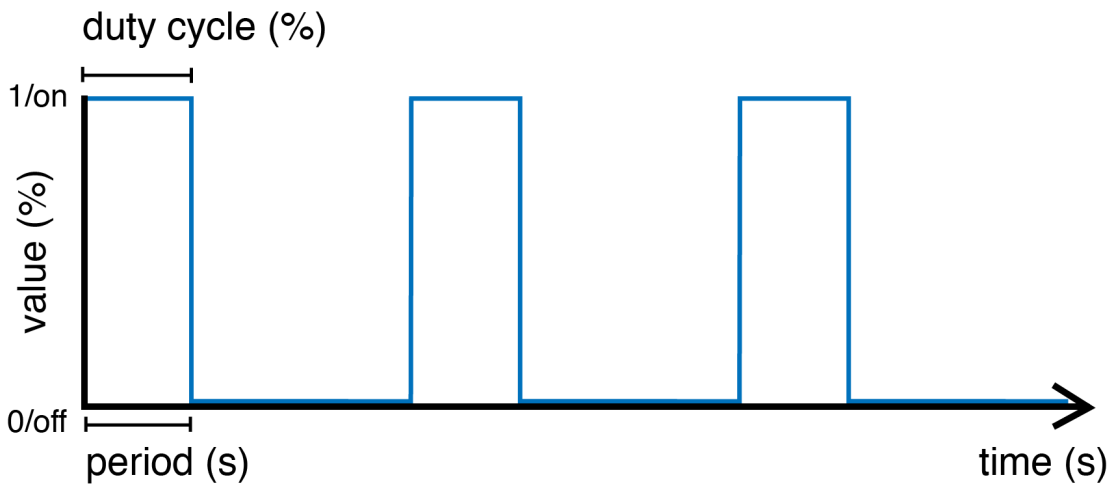
Maps value to new range.

See Also

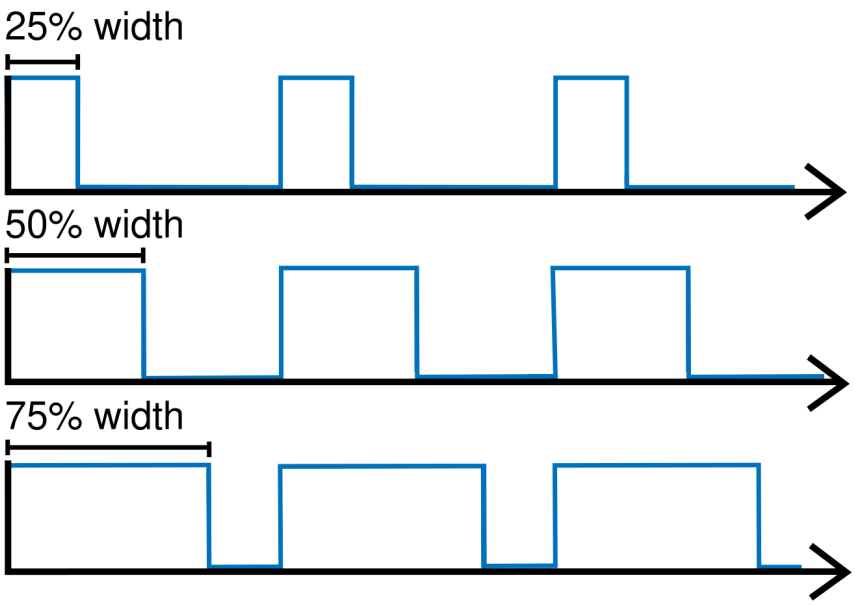
- *SquareWave*
- *TriangleWave*

1.9.2 SquareWave

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 `width`.



The width 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);

SquareWave blinkOsc(4.0);

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

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

class **SquareWave** : public AbstractWave

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

Public Functions

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

Constructor.

Parameters

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

virtual void **period**(float period)

Sets the period (in seconds).

Parameters
period – the period of oscillation (in seconds)

virtual void **frequency**(float frequency)

Sets the frequency (in Hz).

Parameters
frequency – the frequency of oscillation (in Hz)

inline virtual float **frequency**() const

Returns the frequency (in Hz).

virtual void **bpm**(float bpm)

Sets the frequency in beats-per-minute.

Parameters
bpm – the frequency of oscillation (in BPM)

inline virtual float **bpm**() const

Returns the frequency (in BPM).

virtual void **width**(float width)

Sets the width of the signal as a % of period.

Parameters
width – the width as a value in [0, 1]

inline virtual float **width**() const

Returns the width of the signal.

virtual void **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).

inline virtual float **amplitude**() const

Returns the amplitude of the wave.

virtual void **phase**(float phase)

Sets the phase (ie.

the offset, in % of period).

Parameters
phase – the phase (in % of period)

inline virtual float **phase**() const

Returns the phase (in % of period).

virtual float **shiftBy**(float phaseShift)

Returns oscillator's value with given phase shift (in %).

Supports negative phase shifts. Eg. shiftBy(0.2) returns future value of oscillator after 20% of its period would have passed.

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

Returns
the value of oscillator with given phase shift

inline virtual float **get**()

Returns value in [0, 1].

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

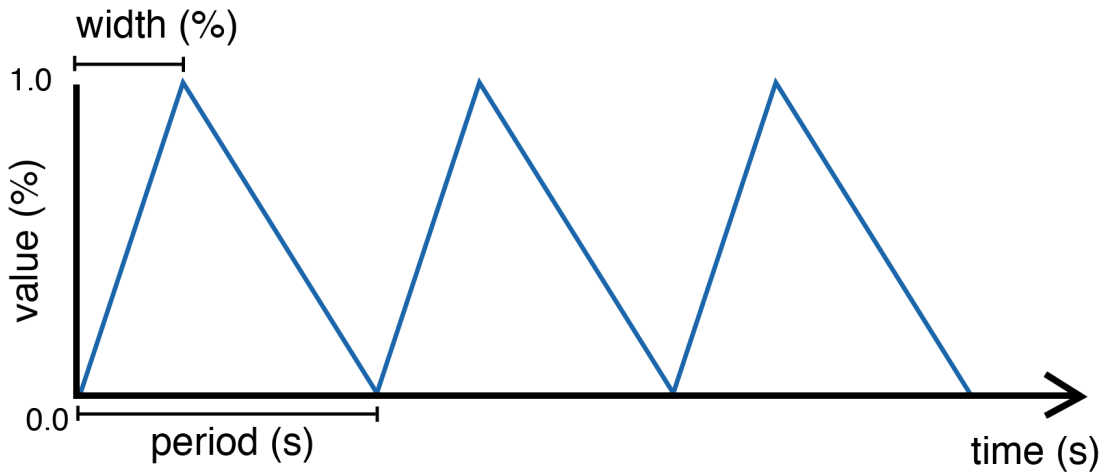
Maps value to new range.

See Also

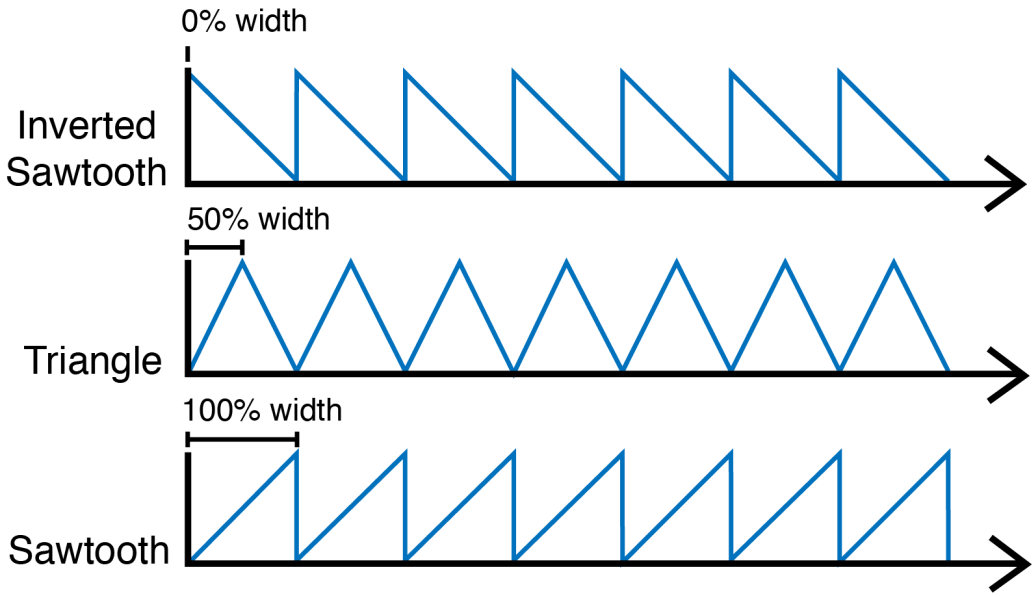
- *SineWave*
- *TriangleWave*

1.9.3 TriangleWave

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);

TriangleWave 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 **TriangleWave** : public AbstractWave

Triangle/sawtooth oscillator.

Public Functions

virtual void **period**(float period)

Sets the period (in seconds).

Parameters
period – the period of oscillation (in seconds)

virtual void **frequency**(float frequency)

Sets the frequency (in Hz).

Parameters
frequency – the frequency of oscillation (in Hz)

inline virtual float **frequency**() const

Returns the frequency (in Hz).

virtual void **bpm**(float bpm)

Sets the frequency in beats-per-minute.

Parameters
bpm – the frequency of oscillation (in BPM)

inline virtual float **bpm**() const

Returns the frequency (in BPM).

virtual void **width**(float width)

Sets the width of the signal as a % of period.

Parameters
width – the width as a value in [0, 1]

inline virtual float **width**() const

Returns the width of the signal.

virtual void **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).

inline virtual float **amplitude**() const

Returns the amplitude of the wave.

virtual void **phase**(float phase)

Sets the phase (ie.

the offset, in % of period).

Parameters
phase – the phase (in % of period)

inline virtual float **phase**() const

Returns the phase (in % of period).

virtual float **shiftBy**(float phaseShift)

Returns oscillator's value with given phase shift (in %).

Supports negative phase shifts. Eg. shiftBy(0.2) returns future value of oscillator after 20% of its period would have passed.

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

Returns
the value of oscillator with given phase shift

inline virtual float **get**()

Returns value in [0, 1].

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

Maps value to new range.

See Also

- *Ramp*
- *SineWave*
- *SquareWave*

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 activate built-in LED. Button is used to reset the alarm at random periods of time.

```
#include <Plaquette.h>

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

DigitalOut led(13);

DigitalIn button(2, INTERNAL_PULLUP);

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

void step() {
  // Activate LED when alarm rings.
  myAlarm >> led; // the alarm will stay "on" until it is stopped or restarted
```

(continues on next page)

(continued from previous page)

```
  // Reset alarm when button is pushed.
  if (myAlarm && button.rose())
  {
    // Restarts the alarm with a random duration between 1 and 5 seconds.
    myAlarm.duration(randomFloat(1.0, 5.0));
    myAlarm.start();
  }
}
```

Reference

class **Alarm** : public DigitalSource, public AbstractTimer

Chronometer class which becomes "on" after a given duration.

Public Functions

inline virtual bool **isOn**()

Returns true iff the input is "on".

inline virtual void **onRise**(EventCallback callback)

Registers event callback on rise event.

inline virtual void **onFall**(EventCallback callback)

Registers event callback on fall event.

inline virtual void **onChange**(EventCallback callback)

Registers event callback on change event.

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 and resets to zero.

virtual void **pause**()

Interrupts the chronometer.

virtual void **resume**()

Resumes process.

inline virtual float **elapsed**() const

The time currently elapsed by the chronometer (in seconds).

virtual void **set**(float time)

Forces current time (in seconds).

virtual void **add**(float time)

Adds/subtracts time to the chronometer.

inline bool **isRunning**() const

Returns true iff the chronometer is currently running.

See Also

- *Chronometer*
- *Metronome*
- *Ramp*
- *SquareWave*

1.10.2 Chronometer

An analog unit that counts time in seconds. It can be started, stopped, paused, and resumed.

Example

Uses a chronometer to change the frequency a blinking LED. Restarts after 10 seconds.

```
#include <Plaquette.h>

Chronometer chrono;

DigitalOut led(13);

SquareOsc osc(1.0); // a square oscillator

void begin() {
  chrono.start(); // start chrono
}

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

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

  // If the chronometer reaches 10 seconds: restart it.
  if (chrono >= 10.0)
  {
    // Restarts the chronometer.
    chrono.start();
  }
}
```

Reference

class **Chronometer** : public Unit, public AbstractChronometer

Public Functions

Chronometer()
Constructor.

inline virtual float **get**()
Returns elapsed time since start (in seconds).

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

virtual void **stop**()
Interrupts the chronometer and resets to zero.

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

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

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

virtual void **set**(float time)
Forces current time (in seconds).

virtual void **add**(float time)
Adds/subtracts time to the chronometer.

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

See Also

- *Alarm*
- *Metronome*
- *Ramp*

1.10.3 Metronome

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

Example

```
#include <Plaquette.h>

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

DigitalOut led(13);
```

(continues on next page)

(continued from previous page)

```
void begin() {
}

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

Reference

class **Metronome** : public DigitalUnit

Chronometer digital unit which emits 1/true/"on" for one frame, at a regular pace.

Public Functions

Metronome(float period = 1.0f)

Constructor.

Parameters
period – the period of oscillation (in seconds)

inline virtual bool **isOn**()

Returns true iff the metronome fires.

virtual void **period**(float period)

Sets the period (in seconds).

Parameters
period – the period of oscillation (in seconds)

inline virtual float **period**() const

Returns the period (in seconds).

virtual void **frequency**(float frequency)

Sets the frequency (in Hz).

Parameters
frequency – the frequency of oscillation (in Hz)

inline virtual float **frequency**() const

Returns the frequency (in Hz).

virtual void **bpm**(float bpm)

Sets the frequency in beats-per-minute.

Parameters
bpm – the frequency of oscillation (in BPM)

inline virtual float **bpm**() const

Returns the frequency (in BPM).

virtual void **phase**(float phase)

Sets the phase (ie.

the offset, in % of period).

Parameters
phase – the phase (in % of period)

inline virtual float **phase**() const

Returns the phase (in % of period).

virtual void **onBang**(EventCallback callback)

Registers event callback on metronome tick event.

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

- *Alarm*
- *Chronometer*
- *Ramp*
- *SquareWave*

1.10.4 Ramp

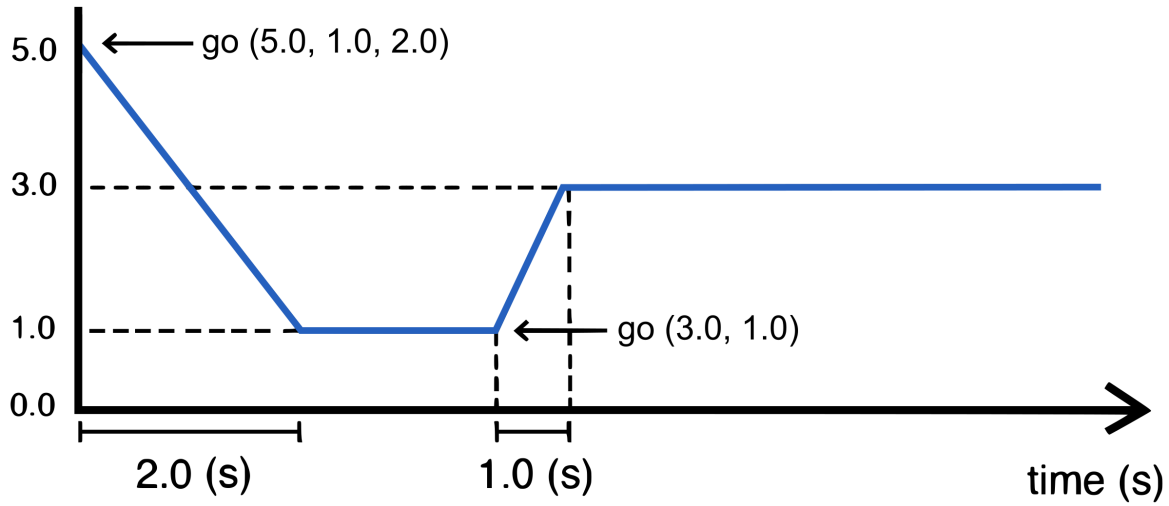
A source unit that generates a smooth transition between two values. The unit can be triggered to start transitioning to a target value for a certain duration.

There are two ways to start the ramp.

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

Alternatively, calling `go(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 `go(5.0, 1.0, 2.0)` is called, followed later by `go(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 `go()` 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(3.0); // initial duration: 3 seconds

StreamOut serialOut(Serial);

void begin() {
  // Apply an easing function (optional).
  myRamp.easing(easeOutSine);
  // Launch ramp: ramp from -10 to 10.
  myRamp.go(-10, 10);
}

void step() {
  if (myRamp.isFinished())
  {
    // Launch ramp from current value to half, increasing duration by one second.
    myRamp.go(myRamp / 2, myRamp.duration() + 1);
  }
```

(continues on next page)

(continued from previous page)

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

Reference

class **Ramp** : public Unit, public AbstractTimer

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

Public Functions

Ramp()

Default constructor.

Ramps from 0 to 1 in one second.

Ramp(float duration)

Constructor with duration.

Parameters
duration – duration of the ramp

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 duration)

Starts/restarts the chronometer with specific duration.

virtual float **progress**() const

The progress of the timer process (in %).

inline virtual bool **isFinished**() const

Returns true iff the chronometer has finished its process.

virtual void **stop**()

Interrupts the chronometer and resets to zero.

virtual void **pause**()

Interrupts the chronometer.

virtual void **resume**()

Resumes process.

inline virtual float **elapsed**() const

The time currently elapsed by the chronometer (in seconds).

virtual void **set**(float time)

Forces current time (in seconds).

virtual void **add**(float time)

Adds/subtracts time to the chronometer.

inline bool **isRunning**() const

Returns true iff the chronometer is currently running.

See Also

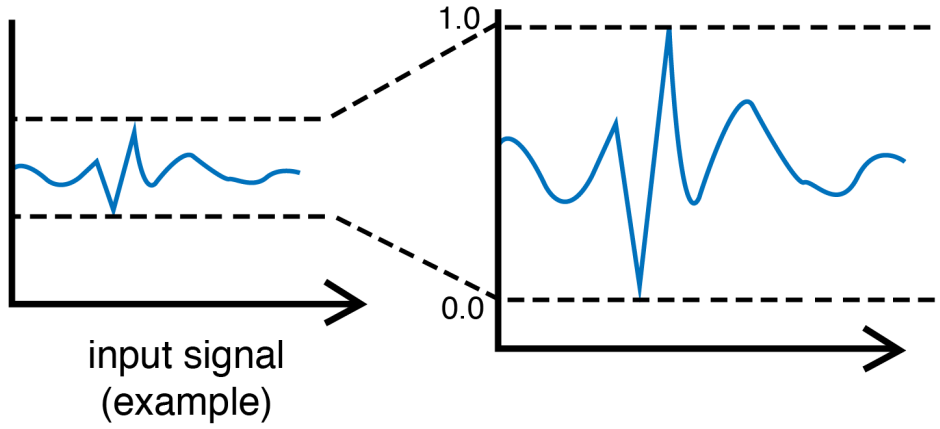
- *Alarm*
- *Chronometer*
- *Easings*
- *Metronome*
- *TriangleWave*

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 accommodate signals that might be changing through time, the user can specify a "decay time window" to control the rate of decay of the minimum 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 minimum 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);

```

(continues on next page)

(continued from previous page)

```
void begin() {}

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 isRunning() 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 **resumeCalibrating**()

Switches to calibration mode (default).

Calls to put(value) will return filtered value AND update the normalization statistics.

virtual void **pauseCalibrating**()

Switches to non-calibration mode: calls to put(value) will return filtered value without updating the normalization statistics.

virtual bool **isCalibrating**() const

Returns true iff the moving filter is in calibration mode.

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 isRunning() 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 **resumeCalibrating**()

Switches to calibration mode (default).

Calls to put(value) will return filtered value AND update the normalization statistics.

virtual void **pauseCalibrating**()

Switches to non-calibration mode: calls to put(value) will return filtered value without updating the normalization statistics.

virtual bool **isCalibrating**() const

Returns true iff the moving filter is in calibration mode.

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*

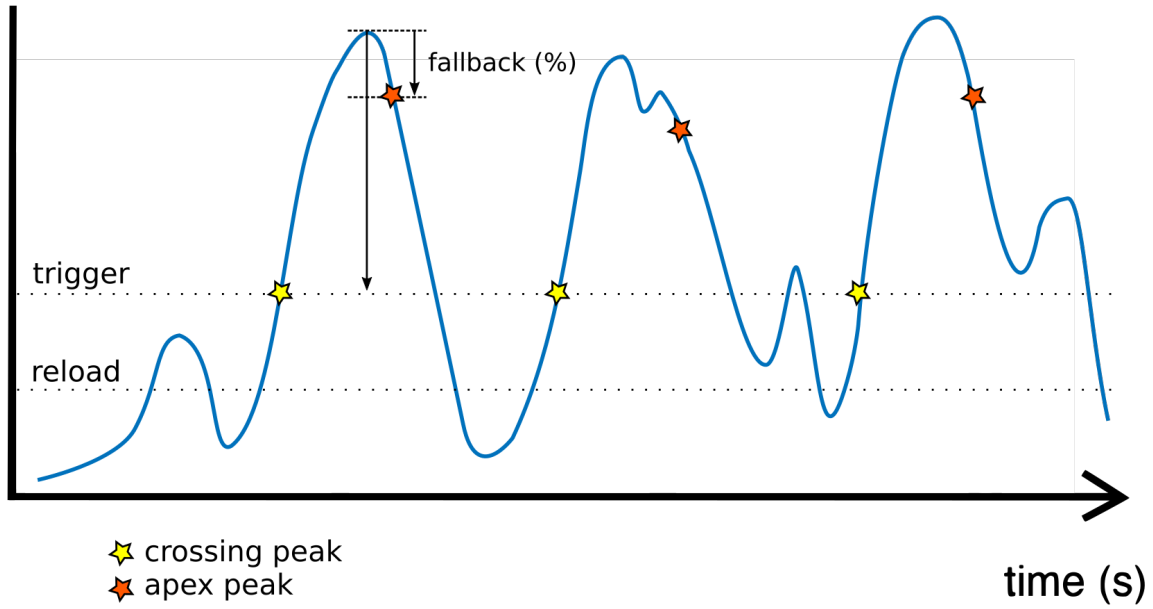
1.11.3 PeakDetector

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);
```

(continues on next page)

(continued from previous page)

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

// 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 DigitalUnit

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.

virtual void **onBang**(EventCallback callback)

Registers event callback on peak detection.

inline virtual float **get**()

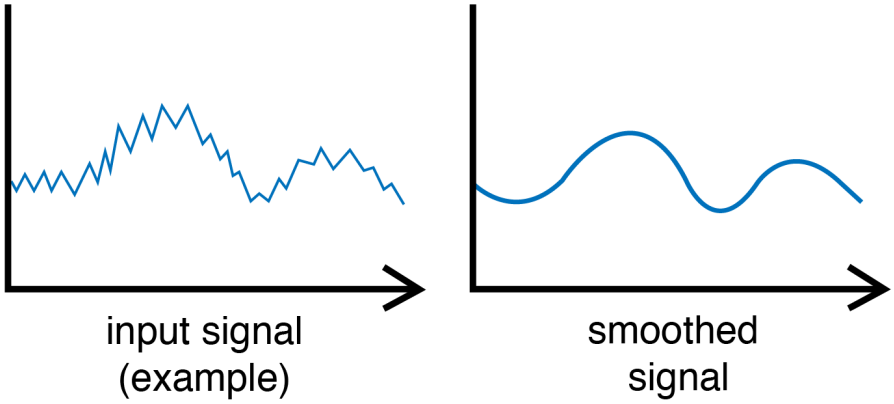
Returns value as float (either 0.0 or 1.0).

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() {}
```

(continues on next page)

(continued from previous page)

```
void step() {
  // Smooth value and send it to serial output.
  sensor >> smoother >> serialOut;
}
```

Note

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

Reference

class **Smoother** : public Unit, public MovingAverage

Simple moving average transform filter.

Public Functions

Smoother(float timeWindow = 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`, and values in-between to values in-between, proportionally.

```
float y = mapFloat(x, 10.0, 50.0, 100.0, 0.0);
```

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

```
float y = mapFloat(x, 10.0, 50.0, 100.0, -100.0);
```

is also valid and works well.

By default, does *not* constrain output to stay within the [`fromHigh`, `toHigh`] range, because out-of-range values are sometimes intended and useful. In order to constrain the return value within range, you can use one of the alternative modes: * the `CONSTRAIN` mode to simply keep the value within range by restricting extreme values as in *constrain() <https://www.arduino.cc/reference/en/language/functions/math/constrain/>* the `WRAP` mode to wrap the values around as in *wrap()*

```
mapFloat(x, 10.0, 50.0, 100.0, -100.0, CONSTRAIN);
mapFloat(x, 10.0, 50.0, 100.0, -100.0, WRAP);
```

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

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

Example

```
#include <Plaquette.h>

SquareWave oscillator(1.0);

DigitalOut led(13);

void begin() {
}

void step() {
  // Change frequency between 2Hz and 15Hz over a 30 seconds period, then the frequency␣
↪will stay at 15Hz.
  float freq = mapFloat(seconds(), 0.0, 30.0, 2.0, 15.0, CONSTRAIN); // try removing␣
↪CONSTRAIN and see what happens
  oscillator.frequency(freq);

  // Send to LED.
```

(continues on next page)

(continued from previous page)

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

Reference

float `pq::mapFloat`(double value, double fromLow, double fromHigh, double toLow, double toHigh, uint8_t mode = UNCONSTRAIN)

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
- **mode** – set to CONSTRAIN to constrain the return value between toLow and toHigh or WRAP for the value to wrap around

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:

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

By default, does *not* constrain output to stay within the [`fromHigh`, `toHigh`] range, because out-of-range values are sometimes intended and useful. In order to constrain the return value within range, use the `CONSTRAIN` argument as the last parameter:

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

See *mapFloat()* for more details.

Example

```
#include <Plaquette.h>

SineWave modulator(10.0);
```

(continues on next page)

(continued from previous page)

```
SquareWave oscillator(1.0);

DigitalOut led(13);

void begin() {
}

void step() {
  // Change duty-cycle of oscillator in range [0.2, 0.8].
  float width = mapFrom01(modulator, 0.2, 0.8); // alternative: modulator.mapTo(0.2, 0.8)
  oscillator.width(width);

  // Send to LED.
  oscillator >> led;
}
```

Reference

float pq::**mapFrom01**(double value, double toLow, double toHigh, uint8_t mode = UNCONSTRAIN)

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
- **mode** – set to CONSTRAIN to constrain the return value between toLow and toHigh or WRAP for the value to wrap around

Returns

the mapped value in [toLow, toHigh]

See Also

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

1.12.3 mapTo01()

Re-maps a number between 0.0 and 1.0. That is, a value of `fromLow` would get mapped to 0.0, a value of `fromHigh` to 1.0, values in-between to values in-between, etc.

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

is equivalent to:

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

By default, does *not* constrain output to stay within the [`fromHigh`, `toHigh`] range, because out-of-range values are sometimes intended and useful. In order to constrain the return value within range, use the `CONSTRAIN` argument as the last parameter:

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

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 the range [0, 1] and send to LED.
  mapTo01(x, -1, 1) >> led;
}
```

Reference

float pq::**mapTo01**(double value, double fromLow, double fromHigh, uint8_t mode = UNCONSTRAIN)

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
- **mode** – set to CONSTRAIN to constrain the return value between toLow and toHigh or WRAP for the value to wrap around

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);
```

(continues on next page)

(continued from previous page)

```
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()

1.12.6 wrap()

Restricts a value to an interval [low, high) by wrapping it around.

Code	Result
`wrap(1.0, 1.0, 5.0)`	`1.0`
`wrap(3.0, 1.0, 5.0)`	`3.0`
`wrap(4.9999, 1.0, 5.0)`	`4.9999`
`wrap(5.0, 1.0, 5.0)`	`1.0`
`wrap(10.0, 1.0, 5.0)`	`1.0`
`wrap(13.0, 1.0, 5.0)`	`3.0`

Two alternative versions are provided:

Version	Equivalent to
`wrap(x, high)`	`wrap(x, 0.0, high)`
`wrap01(x)`	`wrap(x, 0.0, 1.0)`

Example

Ramp LED up and then back to zero once every 10 second:

```
#include <Plaquette.h>

AnalogOut led(9);

void begin() {
}

void step() {
   wrap(seconds(), 10.0) >> led;
}
```

Reference

float `pq::wrap`(double x, double low, double high)

Restricts value to the interval [low, high) by wrapping it around.

Parameters

- **x** – the value to wrap
- **low** – the lower boundary
- **high** – the higher boundary

Returns
the value wrapped around [low, high) or [high, low) if high < low

float `pq::wrap`(double x, double high)

Restricts value to the interval [0, high) by wrapping it around.

Parameters

- **x** – the value to wrap
- **high** – the higher bound

Returns
the value wrapped around [0, high) or [high, 0) if high is negative

float `pq::wrap01`(double x)

Restricts value to the interval [0, 1) by wrapping it around.

Parameters
x – the value to wrap

Returns
the value wrapped around [0, 1).

See Also

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

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>

SquareWave oscillator;
AnalogIn input(A0);

void begin() {
  oscillator.period(1.0);
  oscillator.width(0.75);
  input.smooth();
```

(continues on next page)

(continued from previous page)

```
}

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 [] (arrays)

An array is a collection of variables or objects that are accessed with an index number. Arrays can be complicated, but using simple arrays is relatively straightforward.

For a general description of arrays, please refer to this page.

Arrays of Plaquette units such as *DigitalIn*. *SineWave*, and *MinMaxScaler* can be easily created using the following syntax:

```
UnitType array[] = { UnitType(...), UnitType(...), ... };
```

For example, the following code will create an array of three (3) digital outputs on pins 10, 11, and 12:

```
DigitalOut leds[] = { DigitalOut(10), DigitalOut(11), DigitalOut(12) };
```

When initializing a unit with a single parameter, one can simply use the value of the parameter at creation time. Hence the previous code could be rewritten as:

```
DigitalOut leds[] = { 10, 11, 12 };
```

When more than a single parameter is used, however, it needs to be called explicitely with the unit name:

```
SquareWave oscillators[] = { 1.0, 2.0, SquareWave(3.0, 0.8) };
```

Warning

Units in array need to be all of the same type. In other words, it is not currently possible to mix different types of objects such as DigitalIn and SquareWave in the same array.

Example

```
#include <Plaquette.h>

AnalogOut leds[] = { 9, 10, 11 };

// Creates three triangle oscillators with a 2 seconds period, with different width.
TriangleWave oscillators[] = { TriangleWave(2.0, 0.0), 2.0, TriangleWave(2.0, 1.0) };

void begin() {}

void step() {
  // Send each oscillator to its corresponding LED.
  for (int i=0; i<3; i++) {
    oscillators[i] >> leds[i];
  }
}
```

1.13.4 . (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();
```

(continues on next page)

(continued from previous page)

```
}

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

Syntax

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

1.13.5 >> (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

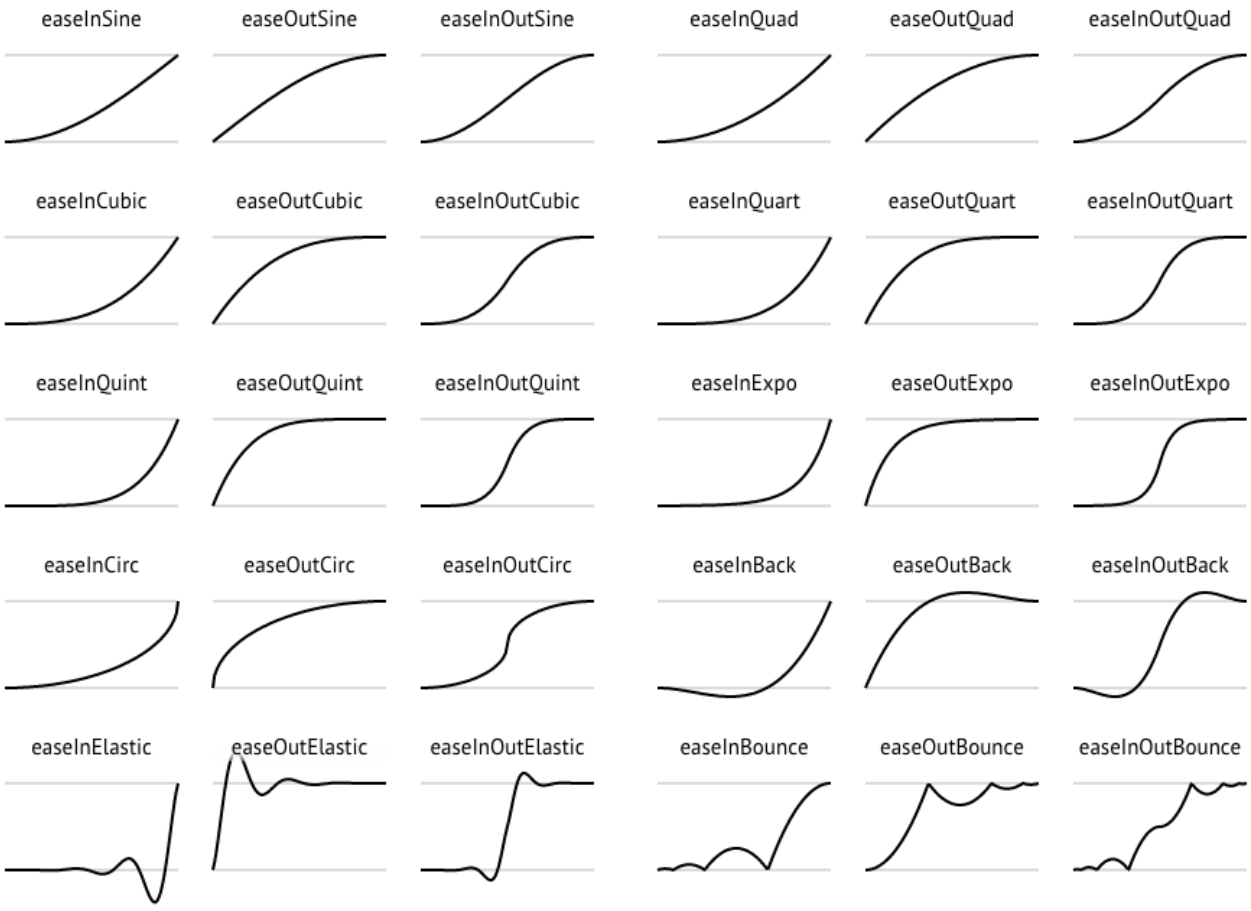
Easing functions apply non-linear effects to changing values, in order to create expressive real-time outputs. Plaquette provides users with a wide range of such functions, typically used with a Ramp unit.

All easing functions have the same signature:

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

Value `x` should be between 0.0 and 1.0, the returned value is also between 0.0 and 1.0.

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 continuous 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

Every time 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);
```

(continues on next page)

(continued from previous page)

```
// Oscillator to make the servo sweep.
SineWave 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

M

N

P

R

S

T