# Contribution Guide

First of all, any kind of contribution is highly appreciated, you don't have to be a pro in C++, neither am I. If you are totally new to native Node.js development with nan and would like to get started, you can have a look at my article series as a quick introduction:

<a href="https://medium.com/netscape/tutorial-building-native-c-modules-for-node-js-using-nan-part-1-755b07389c7c"><b>Tutorial to Native Node.js Modules with C++</b></a>

Most of the time adding bindings is done similarly to what already exists in the codebase. Thus, you can take the existing stuff as an example to help you to get started. In the following, you can find some basic guidelines for adding new OpenCV function bindings to the package.

## API Design

The API is designed such that

1: Parameters passed to a function call are type checked and appropriate messages are displayed to the user in case an error occured. Nobody wants passing garbage to a function by coincidence to fail silently, which may produce unexpected results.

2: A function, which takes more than a single parameter with default values, can conveniently be invoked by passing a JSON object with named parameters in substitution of the optional parameters.

3: If the first argument of a function corresponds to one of the OpenCV classes (usually cv::Mat), the function binding should be exported as a global cv method as well as a class method of the wrapped class, to allow chaining of function calls such as: `mat.resizeToMax(500).toGray().mean()`.

For example consider the following function signature from the official OpenCV 3 docs:

``` c++

void GaussianBlur(InputArray src, OutputArray dst, Size ksize, double sigmaX, double sigmaY=0, int borderType=BORDER\_DEFAULT)

```

cv::InputArray are usually corresponding to cv::Mat, but sometimes they can also be a std::vector of Points or Vectors. All OutputArrays are converted to return values in the binding. If a function has multiple return values, they are returned as a JSON object containing the return values as key value pairs.

Passing optional arguments as named parameters shall provide the convenience of being able to pass single optional parameters without having to pass every other optional parameter.

The function should be invokable in the following ways:

``` javascript

const mat = new cv.Mat(...)

// required arguments

const size = new cv.Size(...)

const sigmaX = 1.2

// optional arguments

const sigmaY = 1.2

const borderType = cv.BORDER\_CONSTANT

let dst

/\* invocation on the global cv object: \*/

// with required arguments

dst = cv.gaussianBlur(mat, size, sigmaX)

// with optional arguments

dst = cv.gaussianBlur(mat, size, sigmaX, sigmaY)

dst = cv.gaussianBlur(mat, size, sigmaX, sigmaY, borderType)

// with named optional arguments as JSON object

dst = cv.gaussianBlur(mat, size, sigmaX, { sigmaY: 1.2 })

dst = cv.gaussianBlur(mat, size, sigmaX, { borderType: cv.BORDER\_CONSTANT })

dst = cv.gaussianBlur(mat, size, sigmaX, { sigmaY: 1.2, borderType: cv.BORDER\_CONSTANT })

/\* invocation as a class method: \*/

// with required arguments

dst = mat.gaussianBlur(size, sigmaX)

// with optional arguments

dst = mat.gaussianBlur(size, sigmaX, sigmaY)

dst = mat.gaussianBlur(size, sigmaX, sigmaY, borderType)

// with named optional arguments as JSON object

dst = mat.gaussianBlur(size, sigmaX, { sigmaY: 1.2 })

dst = mat.gaussianBlur(size, sigmaX, { borderType: cv.BORDER\_CONSTANT })

dst = mat.gaussianBlur(size, sigmaX, { sigmaY: 1.2, borderType: cv.BORDER\_CONSTANT })

```

## Guide to Adding New Function Bindings

Adding a new nodejs binding to an OpenCV function is done in 3 simple steps:

1. Add the function binding

2. Write a unit test

3. Add the type declaration for TypeScript

### 1. Add the Function Binding

Let's consider the GaussianBlur example. Since the first argument is a cv::Mat, we are going to make this a class method binding. Furthermore, GaussianBlur is implemented in the imgproc package of OpenCV (as we have to include `#include <opencv2/imgproc.hpp>` to use this method). Therefore we want to implement the binding in imgprocBindings.h:

``` c++

namespace ImgprocBindings {

...

class GaussianBlur : public CvClassMethodBinding<Mat> {

public:

void createBinding(std::shared\_ptr<FF::Value<cv::Mat>> self) {

// required parameters

auto kSize = req<Size::Converter>();

auto sigmaX = req<FF::DoubleConverter>();

// optional parameters

auto sigmaY = opt<FF::DoubleConverter>("sigmaY", 0);

auto borderType = opt<FF::IntConverter>("borderType", cv::BORDER\_CONSTANT);

// return values

auto blurMat = ret<Mat::Converter>("blurMat");

// the actual function call

executeBinding = [=]() {

cv::GaussianBlur(self, blurMat->ref(), kSize->ref(), sigmaX->ref(), sigmaY->ref(), borderType->ref());

};

};

};

}

```

To expose the synchronous and asynchronous bindings for GaussianBlur we first declare the global methods in Imgproc.h:

``` c++

class Imgproc {

...

static NAN\_METHOD(GaussianBlur);

static NAN\_METHOD(GaussianBlurAsync);

}

```

And then expose the bindings in Imgproc.cc:

``` c++

// in the init hook, we are telling the package to expose the

// global function bindings to the module object (target)

NAN\_MODULE\_INIT(Imgproc::Init) {

...

Nan::SetMethod(target, "gaussianBlur", GaussianBlur);

Nan::SetMethod(target, "gaussianBlurAsync", GaussianBlurAsync);

}

// synchronous binding

NAN\_METHOD(Imgproc::GaussianBlur) {

FF::syncBinding<ImgprocBindings::GaussianBlur>("Imgproc", "GaussianBlur", info);

}

// asynchronous binding

NAN\_METHOD(Imgproc::GaussianBlurAsync) {

FF::asyncBinding<ImgprocBindings::GaussianBlur>("Imgproc", "GaussianBlur", info);

}

```

We repeat this procedure for the class method bindings and declare them in MatImgproc.h

``` c++

class MatImgproc {

...

static NAN\_METHOD(GaussianBlur);

static NAN\_METHOD(GaussianBlurAsync);

}

```

And then expose the bindings in MatImgproc.cc:

``` c++

// in the init hook, we are telling the package to expose those bindings on the

// Mat prototype so that we can actually call it from JavaScript

void MatImgproc::Init(v8::Local<v8::FunctionTemplate> ctor) {

...

Nan::SetPrototypeMethod(ctor, "gaussianBlur", GaussianBlur);

Nan::SetPrototypeMethod(ctor, "gaussianBlurAsync", GaussianBlurAsync);

}

// synchronous binding

NAN\_METHOD(MatImgproc::GaussianBlur) {

Mat::syncBinding<ImgprocBindings::GaussianBlur>("GaussianBlur", info);

}

// asynchronous binding

NAN\_METHOD(MatImgproc::GaussianBlurAsync) {

Mat::asyncBinding<ImgprocBindings::GaussianBlur>("GaussianBlur", info);

}

```

### 2. Writing Unit Tests

We test the bindings directly from JS with a classic mocha + chai setup. The purpose of unit testing is not to ensure correct behaviour of OpenCV function calls as OpenCV functionality is tested and none of our business. However, we want to ensure that our bindings can be called without crashing, that all parameters are passed and objects unwrapped correctly and that the function call returns what we expect it to.

You can use generateAPITests to easily generate default tests for a function binding that is implemented sync and async. This will generate the tests which ensure that the synchronous as well as the callbacked and promisified async bindings are called correctly. However, you are welcome to write additional tests. For the gaussianBlur example we use generateClassMethodTests instead of generateAPITests, which will generate tests for the global method binding `cv.gaussianBlur(mat, ...)` as well as the class method binding `mat.gaussianBlur(...)`.

For the gaussianBlur example generating unit tests can by adding the following to imgprocTests.js located in test/tests/core/Mat:

``` javascript

describe('gaussianBlur', () => {

const matData = [

[0, 0, 128],

[0, 128, 255],

[128, 255, 255]

]

const mat = new cv.Mat(matData, cv.CV\_8U)

const expectOutput = (blurred) => {

assertMetaData(blurred)(mat.rows, mat.cols, mat.type);

expect(dangerousDeepEquals(blurred.getDataAsArray(), matData)).to.be.false;

};

const kSize = new cv.Size(3, 3);

const sigmaX = 1.2;

generateClassMethodTests({

getClassInstance: () => mat,

methodName: 'gaussianBlur',

classNameSpace: 'Mat',

methodNameSpace: 'Imgproc',

getRequiredArgs: () => ([

kSize,

sigmaX

]),

getOptionalArgsMap: () => ([

['sigmaY', 1.2],

['borderType', cv.BORDER\_CONSTANT]

]),

expectOutput

});

});

```

### 3. Adding the Type Declaration

All type declarations are located in lib/typings. We simply add the type information of the signature of the function binding we just implemented to Mat.d.ts:

``` typescript

export class Mat {

...

gaussianBlur(kSize: Size, sigmaX: number, sigmaY?: number, borderType?: number): Mat;

gaussianBlurAsync(kSize: Size, sigmaX: number, sigmaY?: number, borderType?: number): Promise<Mat>;

}

```

And we add the typings for the global function binding to cv.d.ts:

``` typescript

export function gaussianBlur(mat: Mat, kSize: Size, sigmaX: number, sigmaY?: number, borderType?: number): Mat;

export function gaussianBlurAsync(mat: Mat, kSize: Size, sigmaX: number, sigmaY?: number, borderType?: number): Promise<Mat>;

```

And that's it! You can now open a Pull Request, which will be built on the CI.

## Implementing a Wrapped Class

If you want to add a new class wrapper for one of the OpenCV classes you should make this class extend

the FF::ObjectWrap<TClass, T> template, which allows us to use all the helper functions and converters

for the class:

``` c++

// the second template parameter corresponds to the OpenCV class (cv::Mat)

class Mat : public FF::ObjectWrap<Mat, cv::Mat> {

public:

// declare the constructor, required for FF::ObjectWrap

static Nan::Persistent<v8::FunctionTemplate> constructor;

// declare the class name, required for FF::ObjectWrap

static const char\* getClassName() {

return "Mat";

}

// every class binding has to implement and Init method

static NAN\_MODULE\_INIT(Init);

};

```

I provided a step by step guide for implementing class bindings in this [tutorial](https://medium.com/netscape/tutorial-building-native-c-modules-for-node-js-using-nan-part-1-755b07389c7c).

## CI

For continous integration we use AppVeyor and Travis CI, which will run a rebuild of the package on Windows and Linux and run the unit tests for each of the maintained OpenCV versions. This ensures compatibility across the OpenCV versions as in some minor cases the OpenCV interface may have changed or new features have been added.

The build task will be executed on every push to your working branch as well as every pull request before merging to the master branch. If you have docker set up on your local machine you can run the build tasks on your local machine via the provided npm scripts under ci/test.

For example to execute a build for OpenCV 3.4.6 under node 11:

``` bash

# with contrib (OpenCV extra modules)

npm run test 3.4.6-contrib 11

# without contrib

npm run test 3.4.6 11

```