HAL open science

IPipeline: a development framework for image processing pipelines

Cédric Pradalier

▶ To cite this version:

Cédric Pradalier. IPipeline: a development framework for image processing pipelines. [Technical Report] UMI 2958 GeorgiaTech-CNRS. 2017. hal-01470371

HAL Id: hal-01470371
https://hal.science/hal-01470371v1

Submitted on 17 Feb 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

IPipeline: a development framework for image processing pipelines

Cédric Pradalier
UMI 2958 GT-CNRS - GeorgiaTech Lorraine
Metz, France

October 14, 2016

1 Introduction

Developing an image processing or computer vision application typically involves the sequencing of a number of elementary operations on images. Some operations can be simple image processing (blurring, thresholding, morphological operations, ...), other operations will be arithmetic (additions, logical operation, transcendant functions, ...) and others will be more advanced function (edge detection, rectification, ...). In most cases, a tool like OpenCV (http://www.opencv.org) provide optimized and efficient implementations, but in some cases, specialized functions also need to be implemented.

During the development phases, it can sometimes be useful to visualize all the intermediary output and input, or even to create some debugging visualizations that require creating temporary images. In a linear development framework, this requires allocating or reusing specific memory blocks, cluttering the code and impeding readability. In an other century, poor memory management was also an easy source of memory leaks. An additional requirement of the development phase is to design modules that can very clearly check if their input correspond to their expectations in terms of size, color depth, pixel format.

In the optimization phase, the processing tree contains multiple parallelizable path, a deployment of the image processing operations into multiple threads is often desirable. Independently of multi-threading, profiling the timing required for every elementary operation is often very useful in targetting the optimization requirements of a particular application.

For documentation and reusability, one often want to build a visualization of the processing tree with all its inputs and outputs, data types and condition of execution. With a similar objective, a centralized representation of all the processing parameters that can be edited without recompiling is an obvious requirement.

The *IPipeline* framework is a tool that has been developed with all the above goals in mind since 2004. The first version was aimed at the joint processing of 5 individual fish-eye cameras for 3D reconstruction. Later version have been extended with more and more operations, and integrated with various middlewares, DDX, Yarp, and finally ROS. The current version builds upon OpenCV in such a tight way that all objects exchanged between application modules are array of `cv::Mat`. An example of the kind of processing tree we can build with the IPipeline framwork in given in figure 1. Note that the graph description is auto-generated once all the modules have been loaded.

The code is available on https://github.com/cedricpradalier/ipipeline.

2 IPipeline: Design Principle

The core principle behind *IPipeline* is the notion of an ImageProcessor (IP) object. An IP typically has a name and a number of input and, when all the input are available, produces an output, with

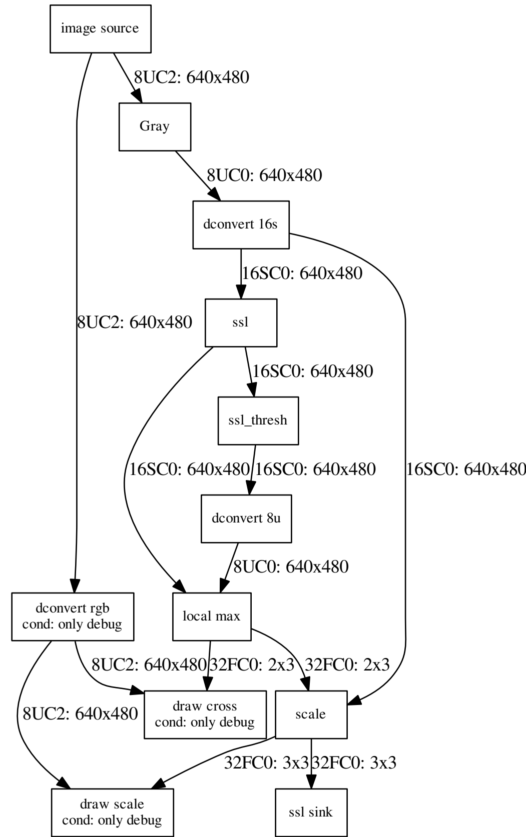

Figure 1: Example of processing tree for the extraction of Self-Similar Landmarks

the specificity that both inputs and outputs are `std::vector<cv::Mat>`. In terms of memory ownership, a key design principle, is that each IP **owns** its output. Programmatically, once IP has defined its input, it must define two functions: `checkInput` that validates that all the inputs have the expected types, consistent sizes, etc, and `processInput` that does the actual computation.

Example of typical IPs are the following:

- Unary operators: image blurring, image thresholding, image negation, fft2, ...
- Binary operators: image addition, image plane combination, ...
- Image sources: file reader, video reader, ROS image transport subscribers, ...
- Image sinks: file saver, video writer, ROS image transport publishers, ...

The sequence of IPs is managed by a specific scheduler. Starting from the known list of source IPs, the scheduler will execute each IP and then push its output to the depending IPs. When an IP has all been provided with all its inputs, it is executed as well (which practically means that is `checkInput` and `processInput` methods are called). When using a multi-threaded scheduler, the threads are organised in a pool of execution agents, while the available IPs are pooled together. When a thread is available, it takes the first available IP, checks that all its input are ready and if so executes it. Otherwise, the IP is put back in the pool, at the back of the list. This round-robin execution of the IPs is sometimes inefficient as it does not maintain cache consistency and does not schedule IPs based on their expected run-time.

Finally, the scheduler and the list of IPs are managed by a driver class that will **own** all these objects and implement our visualization, profiling, ROS integration and documentation objectives.

The core modules are implemented in the subdirectory `ipipeline_core`

2.1 Integration with ROS

Given that the *IPipeline* framework has been designed from the beginning with robotics in mind, it is tightly integrated with ROS www.ros.org, both at the level of its build system and at the level of its middleware.

At the build level, *IPipeline* is split into a number of ROS packages that are managed by Catkin, the package management layer developed over CMake by the ROS community. The consequence of this integration with the ROS framework is minimal beyond a simpler expression of the dependencies between packages in the CMakeLists.txt and package.xml.

At the level of the ROS communication middleware, *IPipeline* provides specialized IPs to publish their input data to ROS and to act as data sources. The ROS message type used can either be `std_msgs::FloatArray` or `sensor_msgs::Image`. In the latter case, the `image_transport` framework is used in combination with `cv_bridge` to convert to/from `cv::Mat` to image messages.

In terms of scheduling, the subscriber modules are specific because they are image sources that are triggered by ROS when an image has been received. In this case, the program using the *IPipeline* framework must wait for this trigger using the driver `waitTrigger` member function.

The ROS modules are implemented in the package `ipipeline_io`.

2.2 Visualization Tools

The purpose of the visualization tools is to be able to display in real-time the output of any of the IPs involved in our processing tree. To this end, a specific type of IP is created to represent probes. The specificity of this IP is that it does not have a fix place in the processing tree and can be moved around interactively. It is sometimes useful to observe simultaneously the output of multiple IPs. In consequence, the *IPipeline* framework support a run-time decided dynamic number of probes.

In practice, when integrated with ROS, the probes are specific image sinks using the `image_transport` framework to publish their input as ROS images. In comparison with normal ROS publishers, they are also doing the required conversion to make their input renderable as either gray-level images or RGB images.

Two tools are provided to interact with the probes in the `ipipeline_view` package: `probe_view` built upon the code of the `image_view` node from the eponymous package, and `sdlprobe` which uses the SDL library to potentially use a little bit less resources.

To launch one of them, use the following command (similar to `image_view`:

```
rosrun ipipeline_view sdlprobe probe:=/Probe0000
```

where the probe topic depends on the configuration file (Probe0000 in this case) and can be deduced from a simple `rostopic list`.

Use mouse clicks (or arrow keys) to move the probe inside the processing tree, mouse wheel to select image index in case of a multi image output.

The probe modules and visualization tools are implemented in the package `ipipeline_view`.

2.3 Conditional Execution

In some context, it may be useful to have a conditional execution of a given IP. It may be based on some parameters in the configuration file, dynamic context variables or even variables control from the `dynamic_reconfigure` package in ROS. An example of such a need would be IPs only useful for debugging as shown in the processing tree in figure 1. Image pre-processing (contrast improvement, bluring, ...) may also be something that need to be activated or not depending on the context. To this end, the *IPipeline* framework includes a concept of Condition variables that can be attached to an IP. When this condition is true, the IP is executed normally. Otherwise,

the input declared as the *default* input is passed through as-is to the output. Note that this is one exception to an IP owning its output since no copy is done before passing the input.

2.4 Profiling Tools

Profiling a computer vision application consists in identifying how every image operation contributes to the total processing tree computing time. Unless working with a framework that provides the tools, once requires to manually add a structure to record the time taken by every operation. Because it is such a common requirement in computer vision, the IPipeline framework provides a common profiling context that is active by default. In practice, everytime the scheduler starts executing its processing tree, it records the start time and end time, and does it as well for every single IP. When the program terminates, a profiling file. In the case of the processing tree presented in figure 1, the profiling data looks like the following:

```
  Iterations = 12315
  Total time = 261.705204 s
    Overhead = 0.379954% 0.994360 s
   Used time = 260.710843 s 0.021170 s/run 47.2 Hz
         ssl = 88.360998% 231.245331 s 0.018778 s/run
   local max = 6.570274% 17.194748 s 0.001396 s/run
        Gray = 0.978708% 2.561331 s 0.000208 s/run
 dconvert 8u = 0.708158% 1.853286 s 0.000150 s/run
dconvert 16s = 0.550779% 1.441417 s 0.000117 s/run
  draw scale = 0.526726% 1.378470 s 0.000112 s/run
dconvert rgb = 0.515246% 1.348426 s 0.000109 s/run
  draw cross = 0.514835% 1.347351 s 0.000109 s/run
       scale = 0.443326% 1.160208 s 0.000094 s/run
  ssl_thresh = 0.389457% 1.019229 s 0.000083 s/run
    ssl sink = 0.060955% 0.159522 s 0.000013 s/run
image source = 0.000582% 0.001524 s 0.000000 s/run
```

This shows that the program run for 261.70s and the IPs used 260.70s of this time, hence only 0.37% of the time was lost due to the *IPipeline* framework overhead. Most of the time was spent in the `ssl` and `local max` IPs. All the other IPs used a negligible amount of time. If an optimization effort is required, it should thus be focused on the `ssl` IP.

In the case of a multi-threaded execution, the percentage interpretation might be a little bit less obvious to interprete, but the ranking of the time used by each module will still be a very valuable information.

2.5 Configuration File

The configuration file used by *IPipeline* is inspired from the INI file in old versions of MS Windows, with a couple of improvements. The core structure is a list of key-values separated by an equal sign, separated into sections. The key difference with the INI file is that the values are strongly typed (int, float, bool, string) and that values can be referred to between variables or between sections using a syntax such as `$varname` or `$section.varname`. The $ notation can also be used to refer to environment variables (`$HOME`, ...). Similarly to the Bash variables, configuration variable concatenation can be forced using braces: `var = "${HOME}/Pictures"`.

An example of such configuration file is shown below (corresponding to the processing tree in figure 1):

```
[main]
middleware = "ros"
period = 0
debug = 1
```

```
[IPLDriver]
numProbes = 3
numThreads = 1
progressWheel = 1
profiling = 1
stepByStep = 0

[IPLTest]
use_source_image = true
infile = "test/SSL_circ3_small.png"
intopic = "/image_raw"
outtopic = "/processed"

[ssl_thresh]
threshold = 5000
max_value = 255
```

It can be seen that there is a main section for high level parameters, a section specific to the generic `IPLDriver` structure and to its specialization `IPLTest`. Finally the `ssl_thresh` IP has its own section with numerical parameters.

2.6 Documentation

To conclude this section on design principle, we felt it was important to build a system that would automatically prepare a graphical representation of itself, for the purpose of validation of the completeness of the processing tree, as well as for the ability to revisit and understand quickly an old processing tree. To this end, every IP has a way to describe itself by printing its name and output type in a format that can be interepreted by the graphviz or vcg packages (Linux). All the examples proposed in the repository will process the first frame while asking to print the processing tree (`spd.nextFrame(printtree=true)`).

In graphviz format, the resulting files (extension .dot) will look like the following example:

```
digraph "Image Processing" {
    node [shape=box]
    000 [label="image source"]
    001 [label="image sink"]
    000 -> 001 [label="8UC2: 420x300"]
}
```

which results in the minimal graph shown in figure 2.

In figure 1, the condition variables are also shown in the processing tree.

3 Developing A New Application

The simplest way to develop a new application is to start from the `ipipeline_template` package. This package implement 3 objects:

- IPLMyProcessor, in src and include: a customized IP that does nothing but is a good starting point. See section 4 for the development of a new IP.
- IPLTestDriver, in nodes: inherits from the generic IPLDriver to control the execution of a processing tree. Its main role is the definition of the processing tree itself.

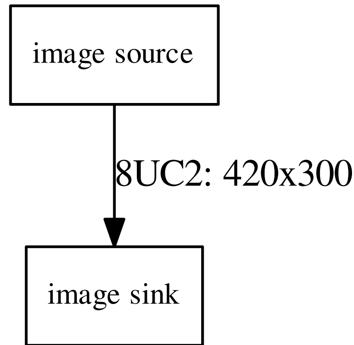

Figure 2: Minimal example of a processing tree described in section 2.6

- IPLTest.cpp: the main code that creates a ros node and an IPLTestDriver instance and run it.

3.1 Writing the driver class

The role of the IPLDriver is to collect some global configuration parameters that will define the processing tree, for instance whether the tree should depend on a ROS source or just an image source, and then define the processing tree. The code for the template is as follows, starting from the headers:

```
#ifndef _IPL_TEST_DRIVER_H_
#define _IPL_TEST_DRIVER_H_

#include "ipipeline_core/IPLDriver.h"

/**
 * \class IPLTestDriver
 * Class implementing a basic dataflow for our template
 * \sa IPLImageProcessor \sa IPLIPScheduler
 * **/
class IPLTestDriver : public IPLDriver
{
   public :
      /**
       * Constructor
       * **/
      IPLTestDriver();
      ~IPLTestDriver();

   protected :

      /**
       * Build the complete processing tree
       * **/
      virtual bool buildProcessingTree();

      virtual bool initialise();

        bool use_source_image;
      std::string infile;
      std::string intopic;
      std::string outtopic;
        ros::NodeHandle nh;
        bool debug;
};
```

```
36
37
38 #endif // _IPL_TEST_DRIVER_H_
```

And the corresponding implementation:

```
1 #include "ipipeline_io/IPLRosImageTransportPublisher.h"
2 #include "ipipeline_io/IPLRosImageTransportSubscriber.h"
3 #include "ipipeline_core/IPLCvImageSource.h"
4 #include "ipipeline_modules/IPLImageToGray.h"
5 #include "ipipeline_template/IPLMyProcessor.h"
6 #include "IPLTestDriver.h"
7
8 // Constructor has to specify the prefix that will be used for the profiling
9 // and graph files.
10 IPLTestDriver::IPLTestDriver() : IPLDriver("IPLTest")
11 {
12     use_source_image = true;
13     debug = false;
14 }
15
16 IPLTestDriver::~IPLTestDriver() { }
17
18
19 bool IPLTestDriver::initialise()
20 {
21     ROS_INFO("In␣'%s::initialise",this->getName().c_str());
22     if (config.selectSection("main")) {
23         config.getBool("debug",&debug);
24     }
25     if (config.selectSection(this->getName())) {
26         config.getBool("use_source_image",&use_source_image);
27         config.getString("infile",infile);
28         config.getString("intopic",intopic);
29         config.getString("outtopic",outtopic);
30         ROS_INFO("Test␣driver:␣reading␣from␣'%s'␣writing␣to␣'%s'",use_source_image?infile.
               c_str():intopic.c_str(),outtopic.c_str());
31     } else {
32         ROS_WARN("Could␣not␣select␣section␣'%s'",this->getName().c_str());
33     }
34     return true;
35 }
36
37 bool IPLTestDriver::buildProcessingTree()
38 {
39     // For each image processor in the tree, we do two or three things.
40     // - We create an instance of the image processor object (new)
41     // - If needed we add the instance as a receiver for the object creating its input
42     // - We store the pointer in the IPLDriver. After this point, the pointer
43     // is owned by the IPLDriver and will be deleted automatically.
44
45     // First create an image source, either from ROS or from a file
46     IPLImageProcessor * image = NULL;
47     if (use_source_image) {
48         image = new IPLCvImageSource("image␣source", infile);
49     } else {
50         image = new IPLRosImageTransportSubscriber("image␣source", nh, intopic, "raw");
51     }
52     store(image);
53
54     // Use a standard module to convert it to 8b gray (path through if it is
55     // already gray)
56    IPLImageToGray * gray = new IPLImageToGray("Gray");
57    image->addReceiver(gray);
58    store(gray);
59
60     // Instantiate our customize image processor
61    IPLMyProcessor * my_proc = new IPLMyProcessor("my␣processor");
62    gray->addReceiver(my_proc);
63    store(my_proc);
64
65     // Instantiate an image sink that will export the result to ROS
66    IPLRosImageTransportPublisher * publisher = new IPLRosImageTransportPublisher("image␣
          sink",nh,outtopic);
67    my_proc->addReceiver(publisher);
68    store(publisher);
69
70    return true;
71 }
```

3.2 Writing the ROS node to operate the driver

The ROS node has several roles. It must first instantiate the ROS infrastructure, then instantiate the IPLDriver and link it with probes that will help monitor the intermediary processing steps. Once this is done, it just keep asking the driver to process new frames until it is interrupted by the user or the driver declares itself complete (which does not happen at this level for simple processing trees).

The basic code is commented below:

```


/* Includes */

#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
#include <sys/time.h>
#include <time.h>

#include <ros/ros.h>

#include "ipipeline_core/IPLGenericProbeFactory.h"
#include "ipipeline_io/IPLRosProbeFactory.h"
#include "IPLTestDriver.h"

/**
 * This program instantiate a IPLDriver object.
 * Use SDL/sdlprobe to explore this processing
 * **/

unsigned int end = 0;

void sighdl(int n)
{
   end += 1;
   ROS_INFO("Countdown␣:␣end␣=␣%d",4-end);
   if (end > 3) {
      kill(getpid(),SIGKILL);
   }
}

int main(int argc,char * argv[])
{
    // Standard ROS initialization
    ros::init(argc,argv,"IPLTest");
    ros::NodeHandle nh;

    // We want one argument: the config file
   if (argc < 2) {
      ROS_ERROR("Expecting␣one␣cmdline␣argument␣(config␣file)");
      return 1;
   }

    // We handle Ctrl-C ourselves.
   signal(SIGINT,sighdl);

    // Create a probe factory that use our node handle.
    IPLRosProbeFactory gpf(nh);

    // And the test driver, that will build the processing tree
   IPLTestDriver spd;

    // Read the config file and distribute it to all image processors
    if (!spd.loadParamsFromFile(argv[1])) {
      ROS_ERROR("Fail␣to␣load␣config␣file␣%s",argv[1]);
    }

    // Create the scheduler and install the probes.
   if (!spd.prepareScheduler(&gpf)) {
      ROS_ERROR("initialisation␣failed");
      return 1;
   }


   ROS_INFO("In␣the␣loop");
   bool first = true;
    // Start a ROS Asyncspinner, because we need to control our loop ourselves.
    // Could be somewhat done with a spinOnce()
```

```
    ros::AsyncSpinner spinner(1);
    spinner.start();

   while (ros::ok() && (end<1)) {
        // Wait for triggers, if any (resulting from subscribers' callbacks).
        if (!spd.waitTrigger(1.0)) {
            continue;
        }

      // Process the received image frame, and the first time, save the
        // processing tree as dot and vcg file
      switch (spd.nextFrame(first)) {
         case IPLDriver::ERROR :
            ROS_ERROR("Processing error");
            end++;
            break;
         case IPLDriver::COMPLETED :
                // If the driver told us that it is done.
            end+=2;
            break;
         case IPLDriver::READY_FOR_RESTART :
                // Only relevant if there are sources that stays constant
                // until something is finished.
            spd.updateCachedSources();
            break;
         case IPLDriver::READY_FOR_NEXT_FRAME :
                // Normal outcome for a real-time image processing application.
            break;
      }
      first = false;

   }
    // And we're done. We'll save the profiling data in the IPLDriver's
    // destructor.
   printf("Terminating\n");

   return 0;
}
```

Finally, to operate our new processing tree, we need to create a configuration file that contains the expected variables. The template provides the following example:

```
[main]
middleware = "ros"
period = 0
debug = 1


[IPLDriver]
numProbes = 3
numThreads = 1
progressWheel = 1
profiling = 1
stepByStep = 0

[IPLTest]
use_source_image = true
infile = "test/my_image.png"
intopic = "/image_raw"
outtopic = "/processed"

[my processor]
double_variable = 3.1415
```

4 Developing A New Image Processor

When developing a new image processor, the simplest is to start from the basic modules provided by `ipipeline_core`. In particular, most image processing task require to create image filters, i.e. IPs that take one input and create one output, or binary operators, i.e. IPs that take two inputs and create one output. When starting from these, there is not much to do since the output allocation has already been done. In the case of our template, we are creating an new IP called `IPLMyProcessor`. Its header only refers to the required virtual functions to do its processing, check its input consistency and read its configuration:

```
1 #ifndef IPL_MY_PROCESSOR_H
2 #define IPL_MY_PROCESSOR_H
3
4 #include "ipipeline_core/IPLImageFilter.h"
5
6 /**
7  * \class IPLMyProcessor : \see IPLImageFilter
8  * Template class for a customized image processor, inheriting from ImageFilter
9  * i.e. it receives an image and output one.
10  * \see IPLImageProcessor for virtual function
11  * **/
12 class IPLMyProcessor : public IPLImageFilter
13 {
14    protected:
15         // Add local variables here
16         double local_variable;
17    public :
18       IPLMyProcessor(const char * name);
19
20       virtual bool processInput();
21
22       virtual bool checkInput() const;
23
24         virtual void readConfig(Config *config);
25 };
26 #endif // IPL_MY_PROCESSOR_H
```

These functions must then be implemented, but they can rely on a lot of helper functions from the ImageProcessor and ImageFilter classes:

```
1 #include "ipipeline_template/IPLMyProcessor.h"
2
3 IPLMyProcessor::IPLMyProcessor(const char * name) :
4    IPLImageFilter(name)
5 {
6     // Nothing to initialize here, everything is inherited from the image
7     // filter.
8 }
9
10 void IPLMyProcessor::readConfig(Config *config)
11 {
12     // Only one dummy variable to read here.
13    config->getDouble("local_variable",&local_variable);
14 }
15
16
17 bool IPLMyProcessor::checkInput() const
18 {
19     // First get the image processor that produced our input (deftInput is the
20     // default input defined by ImageFilter).
21     IPLImageProcessor * input = getInput(deftInput);
22     // Request the output of our input processor, which will be our input data.
23     // ImageProcessorOutput is an array of cv::Mat
24     ImageProcessorOutput in = input->getOutput();
25     // Now check that all our inputs have the correct type (8bit gray here).
26     for (unsigned int i=0;i<in.size();i++) {
27         if (in[i].type() != CV_8UC1) {
28             return error(INVALID_TYPE);
29         }
30     }
31     // OK, everything is consistent
32    return true;
33 }
34
35 bool IPLMyProcessor::processInput()
36 {
```

```
37      // First get the image processor that produced our input (deftInput is the
38      // default input defined by ImageFilter).
39     IPLImageProcessor * input = getInput(deftInput);
40      // Request the output of our input processor, which will be our input data.
41      // ImageProcessorOutput is an array of cv::Mat
42      ImageProcessorOutput in = input->getOutput();
43      // Make sure our output has the same number of cv::Mat as our input
44      outputVector.resize(in.size());
45      // Now iterate on all the cv::Mat in our input
46      for (unsigned int i=0;i<in.size();i++) {
47          // Enforce output image sizes
48          outputVector[i].create(in[i].size(),CV_16SC1);
49          // Initialize to the correct value
50          outputVector[i] = cv::Scalar(0);
51          // And that's all for now... Add your own code below
52      }
53
54      // And we're done. Return false if the processing tree needs to be aborted.
55     return true;
56 }
```

To explore the possibilities to create more complex or generic image processor, check the modules available in the `ipipeline_modules` package.

5 Nested Processing Trees

As a final note, it is sometimes necessary to have one processing step of an algorithm that requires to apply a complex processing tree to many image subwindow. This is a situation where it is necessary to nest processing trees. The *IPipeline* framework support that without particular difficulties. The key points to pay attention to are the following:

- The high-level IP must instantiate and own the IPLDriver with a different prefix than the top-level driver. This makes sure that profiling and other generated files are generated with different names, and that the probe set has a different ROS prefix.
- The high-level IP is responsible for calling the IPLDriver nextFrame function. Because it is a different processing tree, it is also a different scheduler and it must be triggered independently.
- The image source `IPLImageSourceWithView` is a useful asset for this type of problem since it is an IP that publishes a region of interest of a constant cv::Mat. This region of interest can be changed by the driver before every execution tu run the processing tree on a different area of the source image.

6 Conclusion

This document provides a starting point to explore the design of image processing applications with the *IPipeline* framework. A lot of use case have been considered when developing this framework and most of the very specific ones have not been described in this document. To move deeper into the framework at this stage, there is no really better recommendation than checking the source code of the modules in the `ipipeline_modules` and `ipipeline_core` packages.