KWIVER Documentation

Release 1

Kitware, Inc.

Oct 27, 2017

Contents

Contents:

CHAPTER 1

Introduction

The KWIVER toolkit is a collection of software tools designed to tackle challenging image and video analysis problems and other related challenges. Recently started by Kitware's Computer Vision and Scientific Visualization teams, KWIVER is an ongoing effort to transition technology developed over multiple years to the open source domain to further research, collaboration, and product development. KWIVER is a collection of C++ libraries with C and Python bindings and uses an permissive BSD License.

Visit the repository on how to get and build the KWIVER code base

CHAPTER 2

Architecture

One of the primary design goals of KWIVER is to make it easier to pull together algorithms from a wide variety of third-party, open source image and video processing projects and integrate them into highly modular, run-time configurable systems.

This goal is achieved through the three main components of KWIVER: Vital, Arrows, and Sprokit.

Vital Architecture

Vital is the core of KWIVER and is designed to provide data and algorithm abstractions with minimal library dependencies. Vital only depends on the C++ standard library and the header-only Eigen library. Vital defines the core data types and abstract interfaces for core vision algorithms using these types. Vital also provides various system utility functions like logging, plugin management, and configuration file handling. Vital does **not** provide implementations of the abstract algorithms. Implementations are found in Arrows and are loaded dynamically, by vital, at run-time via plugins.

The design of KWIVER allows end-user applications to link only against the Vital libraries and have minimal hard dependencies. One can then dynamically add algorithmic capabilities, with new dependencies, via plugins without needing to recompile Vital or the application code. Only Vital is built by default when building KWIVER without enabling any options in CMake. You will need to enable various Arrows in order for vital to instantiate those various implementations.

The Vital API is all that applications need to control the execute any KWIVER algorithm arrow. In the following sections we will breakdown the various the algorithms and data types provided in vital based on their functionality.

Images

Image Type

class `kwiver::vital::`**`image`**
The representation of an in-memory image.

Images share memory using the image_memory class. This is effectively a view on an image.

Subclassed by kwiver::vital::image_of< uint16_t >, kwiver::vital::image_of< T >

Public Functions

image (**const** image_pixel_traits &*pt* = image_pixel_traits ())
Default Constructor.

Parameters

- `pt`: Change the pixel traits of the image

image (size_t *width*, size_t *height*, size_t *depth* = 1, bool *interleave* = false, **const** image_pixel_traits &*pt* = image_pixel_traits ())
Constructor that allocates image memory.

Create a new blank (empty) image of specified size.

Parameters

- `width`: Number of pixels in width
- `height`: Number of pixel rows
- `depth`: Number of image channels
- `pt`: data type traits of the image pixels
- `interleave`: Set if the pixels are interleaved

image (**const** void **first_pixel*, size_t *width*, size_t *height*, size_t *depth*, ptrdiff_t *w_step*, ptrdiff_t *h_step*, ptrdiff_t *d_step*, **const** image_pixel_traits &*pt* = image_pixel_traits ())
Constructor that points at existing memory.

Create a new image from supplied memory.

Parameters

- `first_pixel`: Address of the first pixel in the image. This does not have to be the lowest memory address of the image memory.
- `width`: Number of pixels wide
- `height`: Number of pixels high
- `depth`: Number of image channels
- `w_step`: pointer increment to get to next pixel column
- `h_step`: pointer increment to get to next pixel row
- `d_step`: pointer increment to get to next image channel
- `pt`: data type traits of the image pixels

image (**const** image_memory_sptr &*mem*, **const** void **first_pixel*, size_t *width*, size_t *height*, size_t *depth*, ptrdiff_t *w_step*, ptrdiff_t *h_step*, ptrdiff_t *d_step*, **const** image_pixel_traits &*pt* = image_pixel_traits ())
Constructor that shares memory with another image.

Create a new image from existing image.

Parameters

- `mem`: Shared memory block to be used
- `first_pixel`: Address of the first pixel in the image. This does not have to be the lowest memory address of the image memory.
- `width`: Number of pixels wide
- `height`: Number of pixels high
- `depth`: Number of image channels
- `w_step`: pointer increment to get to next pixel column
- `h_step`: pointer increment to get to next pixel row
- `d_step`: pointer increment to get to next image channel
- `pt`: data type traits of the image pixels

image(**const** *image* &*other*)
Copy Constructor.

The new image will share the same memory as the old image

Parameters

- `other`: The other image.

const *image* &**operator=**(**const** *image* &*other*)
Assignment operator.

const image_memory_sptr &**memory**() **const**
Const access to the image memory.

image_memory_sptr **memory**()
Access to the image memory.

size_t **size**() **const**
The size of the image managed data in bytes.

The size of the image data in bytes.

This size includes all allocated image memory, which could be larger than width*height*depth*bytes_per_pixel.

Note This size only accounts for memory which is owned by the image. If this image was constructed as a view into third party memory then the size is reported as 0.

const void ***first_pixel**() **const**
Const access to the pointer to first image pixel.

This may differ from *data()* if the image is a window into a large image memory chunk.

void ***first_pixel**()
Access to the pointer to first image pixel.

This may differ from *data()* if the image is a window into a larger image memory chunk.

size_t **width**() **const**
The width of the image in pixels.

size_t **height**() **const**
The height of the image in pixels.

size_t **depth**() **const**
The depth (or number of channels) of the image.

const image_pixel_traits &**pixel_traits**() **const**
The trait of the pixel data type.

ptrdiff_t **w_step**() **const**
The the step in memory to next pixel in the width direction.

ptrdiff_t **h_step**() **const**
The the step in memory to next pixel in the height direction.

ptrdiff_t **d_step**() **const**
The the step in memory to next pixel in the depth direction.

bool **is_contiguous**() **const**
Return true if the pixels accessible in this image form a contiguous memory block.

template <typename T>
T &**at**(unsigned *i*, unsigned *j*)
Access pixels in the first channel of the image.

Parameters

- `i`: width position (x)
- `j`: height position (y)

template <typename T>
const T &**at**(unsigned *i*, unsigned *j*) **const**
Const access pixels in the first channel of the image.

template <typename T>
T &**at**(unsigned *i*, unsigned *j*, unsigned *k*)
Access pixels in the image (width, height, channel)

template <typename T>
const T &**at**(unsigned *i*, unsigned *j*, unsigned *k*) **const**
Const access pixels in the image (width, height, channel)

void **copy_from**(**const** *image* &*other*)
Deep copy the image data from another image into this one.

void **set_size**(size_t *width*, size_t *height*, size_t *depth*)
Set the size of the image.

If the size has not changed, do nothing. Otherwise, allocate new memory matching the new size.

Parameters

- `width`: a new image width
- `height`: a new image height
- `depth`: a new image depth

Time Stamp

class `kwiver::vital::`**timestamp**
Frame time.

This class represents a timestamp for a single video frame. The time is represented in seconds and frame numbers start at one.

A timestamp has the notion of valid time and valid frame. This is useful when dealing with interpolated timestamps. In this case, a timestamp may have a time, but no frame.

When comparing timestamps, they must be from the same domain. If not, then they are not comparable and **all** relative operators return false.

If both timestamps have a time, then they are ordered by that value. If both do not have time but both have frame numbers, they are ordered by frame number. If the timestamps do not have some way of being compared, all relational operators return false.

Public Functions

timestamp()
Default constructor.

Created an invalid timestamp.

timestamp(time_t *t*, frame_t *f*)
Constructor.

Creates a valid timestamp with specified time and frame number.

Parameters

- `t`: Time for timestamp
- `f`: Frame number for timestamp

bool **is_valid**() **const**
Is timestamp valid.

Both the time and frame must be set for a timestamp to be totally valid.

Return **true** if both time and frame are valid

bool **has_valid_time**() **const**
Timestamp has valid time.

Indicates that the time has been set for this timestamp.

Return **true** if time has been set

bool **has_valid_frame**() **const**
Timestamp has valid frame number.

Indicates that the frame number has been set for this timestamp.

Return **true** if frame number has been set

time_t **get_time_usec**() **const**
Get time from timestamp.

The time portion of the timestamp is returned in micro-seconds. The value will be undetermined if the timestamp does not have a valid time.

See *has_valid_time()*

Return Frame time in micro-seconds

double **get_time_seconds**() **const**
Get time in seconds.

The time portion of the timestamp is returned in seconds and fractions.

Return time in seconds.

frame_t **get_frame**() **const**
Get frame number from timestamp.

The frame number value from the timestamp is returned. The first frame in a sequence is usually one. The frame number will be undetermined if the timestamp does not have a valid frame number set.

See *has_valid_frame()*

Return Frame number.

timestamp &**set_time_usec**(time_t *t*)
Set time portion of timestamp.

Parameters

- `t`: Time for frame.

timestamp &**set_time_seconds**(double *t*)
Set time portion of timestamp.

Parameters

- `t`: Time for frame.

timestamp &**set_frame**(frame_t *f*)
Set frame portion of timestamp.

Parameters

- `f`: Frame number

timestamp &**set_invalid**()
Set timestamp totally invalid.

Both the frame and time are set to invalid

timestamp &**set_time_domain_index**(int *dom*)
Set time domain index for this timestamp.

Return Reference to this object.

Parameters

- `dom`: Time domain index

std::string **`pretty_print`**`()` **const**
Format object in a readable manner.

This method formats a time stamp in a readable and recognizable manner suitable form debugging and logging.

Return formatted timestamp

Image Container Type

class `kwiver::vital::image_container`
An abstract representation of an image container.

This class provides an interface for passing image data between algorithms. It is intended to be a wrapper for image classes in third-party libraries and facilitate conversion between various representations. It provides limited access to the underlying data and is not intended for direct use in image processing algorithms.

Subclassed by kwiver::arrows::ocv::image_container, kwiver::arrows::vcl::image_container, kwiver::arrows::vxl::image_container, kwiver::vital::simple_image_container

Public Functions

virtual **`~image_container`**`()`
Destructor.

virtual size_t **`size`**`()` **const** = 0
The size of the image data in bytes.

This size includes all allocated image memory, which could be larger than width*height*depth.

virtual size_t **`width`**`()` **const** = 0
The width of the image in pixels.

virtual size_t **`height`**`()` **const** = 0
The height of the image in pixels.

virtual size_t **`depth`**`()` **const** = 0
The depth (or number of channels) of the image.

virtual *image* **`get_image`**`()` **const** = 0
Get and in-memory image class to access the data.

virtual video_metadata_sptr **`get_metadata`**`()` **const**
Get metadata associated with this image.

virtual void **`set_metadata`**`(`video_metadata_sptr *md*`)`
Set metadata associated with this image.

Image I/O Algorithm

Instantiate with:

```
kwiver::vital::algo::image_io_sptr img_io = kwiver::vital::algo::image_io::create("
↪<impl_name>");
```

Arrow & Configuration	<impl_name> options	CMake Flag to Enable
OpenCV	ocv	KWIVER_ENABLE_OPENCV
VXL	vxl	KWIVER_ENABLE_VXL

class `kwiver::vital::algo::`**`image_io`**

An abstract base class for reading and writing images.

This class represents an abstract interface for reading and writing images.

Inherits from *kwiver::vital::algorithm_def< image_io >*

Subclassed by kwiver::vital::algorithm_impl< image_io, vital::algo::image_io >, kwiver::vital::algorithm_impl< image_io_dummy, kwiver::vital::algo::image_io >

Public Functions

image_container_sptr **`load`**(std::string **const** &*filename*) **const**

Load image from the file.

Return an image container refering to the loaded image

Exceptions

- `kwiver::vital::path_not_exists`: Thrown when the given path does not exist.
- `kwiver::vital::path_not_a_file`: Thrown when the given path does not point to a file (i.e. it points to a directory).

Parameters

- `filename`: the path to the file the load

void **`save`**(std::string **const** &*filename*, kwiver::vital::image_container_sptr *data*) **const**

Save image to a file.

Image file format is based on file extension.

Exceptions

- `kwiver::vital::path_not_exists`: Thrown when the expected containing directory of the given path does not exist.
- `kwiver::vital::path_not_a_directory`: Thrown when the expected containing directory of the given path is not actually a directory.

Parameters

- `filename`: the path to the file to save
- `data`: the image container refering to the image to write

Public Static Functions

static std::string **`static_type_name`**()

Return the name of this algorithm.

Image Filter Algorithm

Instantiate with:

```
kwiver::vital::algo::image_filter_sptr img_filter = kwiver::vital::algo::image_
↪filter::create("<impl_name>");
```

Arrow & Configuration	<impl_name> options	CMake Flag to Enable
N/A	N/A	N/A

** Currently there are no arrows implementing the image_filter algorithm **

class `kwiver::vital::algo::`**`image_filter`**

Abstract base class for feature set filter algorithms.

Inherits from *kwiver::vital::algorithm_def< image_filter >*

Subclassed by kwiver::vital::algorithm_impl< matlab_image_filter, vital::algo::image_filter >

Public Functions

virtual kwiver::vital::image_container_sptr **`filter`**(kwiver::vital::image_container_sptr *image_data*) = 0

Filter a input image and return resulting image.

This method implements the filtering operation. The resulting image should be the same size as the input image.

Return a filtered version of the input image

Parameters

- `image_data`: Image to filter.

Public Static Functions

static std::string **`static_type_name`**()

Return the name of this algorithm.

Split Image Algorithm

Instantiate with:

```
kwiver::vital::algo::split_image_sptr img_filter = kwiver::vital::algo::split_
↪image::create("<impl_name>");
```

Arrow & Configuration	<impl_name> options	CMake Flag to Enable
OpenCV	ocv	KWIVER_ENABLE_OPENCV
VXL	vxl	KWIVER_ENABLE_VXL

class `kwiver::vital::algo::`**`split_image`**

An abstract base class for converting base image type.

Inherits from *kwiver::vital::algorithm_def< split_image >*

Subclassed by kwiver::vital::algorithm_impl< split_image, vital::algo::split_image >

Public Functions

void **set_configuration**(kwiver::vital::config_block_sptr *config*)
Set this algorithm's properties via a config block.

bool **check_configuration**(kwiver::vital::config_block_sptr *config*) **const**
Check that the algorithm's currently configuration is valid.

Check that the algorithm's current configuration is valid.

virtual std::vector<kwiver::vital::image_container_sptr> **split**(kwiver::vital::image_container_sptr *img*) **const** = 0
Split image.

Public Static Functions

static std::string **static_type_name**()
Return the name of this algorithm.

Code Example

```
#include "vital/types/image.h"
#include "vital/types/image_container.h"

#include "vital/algo/image_io.h"
#include "vital/algo/image_filter.h"
#include "vital/algo/split_image.h"

#include "vital/plugin_loader/plugin_manager.h"

// We will be calling some OpenCV code, so we need to include
// some OpenCV related files
#include <opencv2/highgui/highgui.hpp>
#include "arrows/ocv/image_container.h"

void how_to_part_01_images()
{
  // Note that the use of _sptr in objet typing.
  // All vital objects (types, algorithms, etc.) provide a shared_pointer typedef
  // This shared pointer typedef is used through out kwiver to elimate the need of memory ownership managers

  // All algorithms are implemented/encapsulated in an arrow, and operate on vital classes
  // There are various algorithms (arrows) that kwiver provides that you can use to analyze imagry
  // In this example, while we will look at a few algorithms, this example highlights the vital data types used by algorithms
  // These vital data types can then be used as inputs or outputs for algorithms.
  // The vital data types are a sort of common 'glue' between dispart algorithms allowing them to work together.

  // Image I/O algorithms are derived from the kwiver::vital::image_io algorithm interface

```

```
  // While we could instantiate a particular algorithm object directly with this code
  // kwiver::arrows::ocv::image_io ocv_io;
  // kwiver::arrows::vxl::image_io vxl_io;
  // This would require our application to include specific headers be include in our code
  // and require our application to directly link to OpenCV and cause a dependency

  // A key feature of the KWIVER architecture is the ability to dynamically load available algorithms at runtime.
  // This ability allow you to write your application with a set of basic data types and algorithm interfaces and
  // then dynamically replace or reconfigure algorithms at run time without needing to recompile
  // New algorithms can be dropped on disk at and KWIVER can run them
  // The first thing to do is to tell kwiver to load up all it's plugins (which includes all the algorithms)
  kwiver::vital::plugin_manager::instance().load_all_plugins();

  // Refer to this page : http://kwiver.readthedocs.io/en/latest/vital/images.html
  // Documenting the types and algorithms associated with images:
  //               Various implementations of the algorithm,
  //               The string to use to specify creation of a specific implementation,
  //               The KWIVER CMake option that builds the specific implementation

  ///////////////
  // Image I/O //
  ///////////////

  // The main image libraries used in KWIVER are the OpenCV and VXL libraries
  kwiver::vital::algo::image_io_sptr ocv_io = kwiver::vital::algo::image_io::create("ocv");
  kwiver::vital::algo::image_io_sptr vxl_io = kwiver::vital::algo::image_io::create("vxl");

  // The image_io interface is simple, and has a load and save method
  // These methods will operate on the vital object image_container
  // The image_container is intended to be a wrapper for image to facilitate conversion between
  // various representations. It provides limited access to the underlying
  // data and is not intended for direct use in image processing algorithms.
  kwiver::vital::image_container_sptr ocv_img = ocv_io->load("./cat.jpg");
  kwiver::vital::image_container_sptr vxl_img = vxl_io->load("./cat.jpg");

  // Let's use OpenCV to display the images
  // NOTE, this requires that our application CMakeLists properly find_package(OpenCV)
  // And that we tell our application CMake targets about OpenCV (See the CMakeLists.txt for this file)
  cv::Mat mat;
  // First, convert the image to an OpenCV image object
  mat = kwiver::arrows::ocv::image_container::vital_to_ocv(ocv_img->get_image());
  cv::namedWindow("Image loaded by OpenCV", cv::WINDOW_AUTOSIZE);// Create a window for display.
  cv::imshow("Image loaded by OpenCV", mat);                     // Show our image inside it.
  cv::waitKey(5);
  Sleep(2000);                                                   // Wait for 2s
  cvDestroyWindow("Image loaded by OpenCV");
```

```
75
76    // We can do the same, even if the image was originally loaded with VXL
77    mat = kwiver::arrows::ocv::image_container::vital_to_ocv(vxl_img->get_image());
78    cv::namedWindow("Image loaded by VXL", cv::WINDOW_AUTOSIZE);// Create a window for␣
↪display.
79    cv::imshow("Image loaded by VXL", mat);                    // Show our image␣
↪inside it.
80    cv::waitKey(5);
81    Sleep(2000);                                               // Wait for 2s
82    cvDestroyWindow("Image loaded by VXL");
83
84    ////////////////
85    // Image Filter //
86    ////////////////
87
88    // Currently, there is no arrow implementing image filtering
89    //kwiver::vital::algo::image_filter_sptr _filter = kwiver::vital::algo::image_
↪filter::create("<impl_name>");
90
91    ////////////////
92    // Split Image //
93    ////////////////
94
95    // These algorithms split an image in half (left and right)
96    kwiver::vital::algo::split_image_sptr ocv_split = kwiver::vital::algo::split_
↪image::create("ocv");
97    kwiver::vital::algo::split_image_sptr vxl_split = kwiver::vital::algo::split_
↪image::create("vxl");
98
99    std::vector<kwiver::vital::image_container_sptr> ocv_imgs = ocv_split->split(vxl_
↪img);
100   for (kwiver::vital::image_container_sptr i : ocv_imgs)
101   {
102     mat = kwiver::arrows::ocv::image_container::vital_to_ocv(i->get_image());
103     cv::namedWindow("OpenCV Split Image", cv::WINDOW_AUTOSIZE);// Create a window for␣
↪display.
104     cv::imshow("OpenCV Split Image", mat);                    // Show our image␣
↪inside it.
105     cv::waitKey(5);
106     Sleep(2000);                                              // Wait for 2s
107     cvDestroyWindow("OpenCV Split Image");
108   }
109
110   std::vector<kwiver::vital::image_container_sptr> vxl_imgs = ocv_split->split(ocv_
↪img);
111   for (kwiver::vital::image_container_sptr i : vxl_imgs)
112   {
113     mat = kwiver::arrows::ocv::image_container::vital_to_ocv(i->get_image());
114     cv::namedWindow("VXL Split Image", cv::WINDOW_AUTOSIZE);// Create a window for␣
↪display.
115     cv::imshow("VXL Split Image", mat);                    // Show our image inside␣
↪it.
116     cv::waitKey(5);
117     Sleep(2000);                                           // Wait for 2s
118     cvDestroyWindow("VXL Split Image");
119   }
120
121 }
```

Detections

The following types are provided to encapsulate data generated by an algorithm, associated with a particular image.

bounding_box	*detected_object*	*detected_object_set*

Vital Doxygen

Types

Other

There are various other vital types that are also used to help direct algorithms or hold specific data associated with an image.

camera	*camera_intrinsics*	
rgb_color	*covariance*	*descriptor*
descriptor_request	*descriptor_set*	

Image

class `kwiver::vital::image`

The representation of an in-memory image.

Images share memory using the image_memory class. This is effectively a view on an image.

Subclassed by kwiver::vital::image_of< uint16_t >, kwiver::vital::image_of< T >

Public Functions

image (**const** image_pixel_traits &*pt* = image_pixel_traits ())

Default Constructor.

Parameters

- `pt`: Change the pixel traits of the image

image (size_t *width*, size_t *height*, size_t *depth* = 1, bool *interleave* = false, **const** image_pixel_traits &*pt* = image_pixel_traits ())

Constructor that allocates image memory.

Create a new blank (empty) image of specified size.

Parameters

- `width`: Number of pixels in width
- `height`: Number of pixel rows
- `depth`: Number of image channels
- `pt`: data type traits of the image pixels
- `interleave`: Set if the pixels are interleaved

image(**const** void **first_pixel*, size_t *width*, size_t *height*, size_t *depth*, ptrdiff_t *w_step*, ptrdiff_t *h_step*, ptrdiff_t *d_step*, **const** image_pixel_traits &*pt* = image_pixel_traits ())

Constructor that points at existing memory.

Create a new image from supplied memory.

Parameters

- `first_pixel`: Address of the first pixel in the image. This does not have to be the lowest memory address of the image memory.
- `width`: Number of pixels wide
- `height`: Number of pixels high
- `depth`: Number of image channels
- `w_step`: pointer increment to get to next pixel column
- `h_step`: pointer increment to get to next pixel row
- `d_step`: pointer increment to get to next image channel
- `pt`: data type traits of the image pixels

image(**const** image_memory_sptr &*mem*, **const** void **first_pixel*, size_t *width*, size_t *height*, size_t *depth*, ptrdiff_t *w_step*, ptrdiff_t *h_step*, ptrdiff_t *d_step*, **const** image_pixel_traits &*pt* = image_pixel_traits ())

Constructor that shares memory with another image.

Create a new image from existing image.

Parameters

- `mem`: Shared memory block to be used
- `first_pixel`: Address of the first pixel in the image. This does not have to be the lowest memory address of the image memory.
- `width`: Number of pixels wide
- `height`: Number of pixels high
- `depth`: Number of image channels
- `w_step`: pointer increment to get to next pixel column
- `h_step`: pointer increment to get to next pixel row
- `d_step`: pointer increment to get to next image channel
- `pt`: data type traits of the image pixels

image(**const** *image* &*other*)

Copy Constructor.

The new image will share the same memory as the old image

Parameters

- `other`: The other image.

const *image* &**operator=**(**const** *image* &*other*)

Assignment operator.

const image_memory_sptr &**memory**() **const**
Const access to the image memory.

image_memory_sptr **memory**()
Access to the image memory.

size_t **size**() **const**
The size of the image managed data in bytes.

The size of the image data in bytes.

This size includes all allocated image memory, which could be larger than width*height*depth*bytes_per_pixel.

Note This size only accounts for memory which is owned by the image. If this image was constructed as a view into third party memory then the size is reported as 0.

const void ***first_pixel**() **const**
Const access to the pointer to first image pixel.

This may differ from *data()* if the image is a window into a large image memory chunk.

void ***first_pixel**()
Access to the pointer to first image pixel.

This may differ from *data()* if the image is a window into a larger image memory chunk.

size_t **width**() **const**
The width of the image in pixels.

size_t **height**() **const**
The height of the image in pixels.

size_t **depth**() **const**
The depth (or number of channels) of the image.

const image_pixel_traits &**pixel_traits**() **const**
The trait of the pixel data type.

ptrdiff_t **w_step**() **const**
The the step in memory to next pixel in the width direction.

ptrdiff_t **h_step**() **const**
The the step in memory to next pixel in the height direction.

ptrdiff_t **d_step**() **const**
The the step in memory to next pixel in the depth direction.

bool **is_contiguous**() **const**
Return true if the pixels accessible in this image form a contiguous memory block.

template <typename T>
T &**at**(unsigned *i*, unsigned *j*)
Access pixels in the first channel of the image.

Parameters

- `i`: width position (x)
- `j`: height position (y)

template <typename T>
const T &**at**(unsigned *i*, unsigned *j*) **const**
Const access pixels in the first channel of the image.

template <typename T>
T &**at**(unsigned *i*, unsigned *j*, unsigned *k*)
Access pixels in the image (width, height, channel)

template <typename T>
const T &**at**(unsigned *i*, unsigned *j*, unsigned *k*) **const**
Const access pixels in the image (width, height, channel)

void **copy_from**(**const** *image* &*other*)
Deep copy the image data from another image into this one.

void **set_size**(size_t *width*, size_t *height*, size_t *depth*)
Set the size of the image.

If the size has not changed, do nothing. Otherwise, allocate new memory matching the new size.

Parameters

- `width`: a new image width
- `height`: a new image height
- `depth`: a new image depth

class `kwiver::vital::image_container`
An abstract representation of an image container.

This class provides an interface for passing image data between algorithms. It is intended to be a wrapper for image classes in third-party libraries and facilitate conversion between various representations. It provides limited access to the underlying data and is not intended for direct use in image processing algorithms.

Subclassed by kwiver::arrows::ocv::image_container, kwiver::arrows::vcl::image_container, kwiver::arrows::vxl::image_container, kwiver::vital::simple_image_container

Public Functions

virtual ~image_container()
Destructor.

virtual size_t **size**() **const** = 0
The size of the image data in bytes.

This size includes all allocated image memory, which could be larger than width*height*depth.

virtual size_t **width**() **const** = 0
The width of the image in pixels.

virtual size_t **height**() **const** = 0
The height of the image in pixels.

virtual size_t **depth**() **const** = 0
The depth (or number of channels) of the image.

virtual *image* **get_image**() **const** = 0
Get and in-memory image class to access the data.

virtual video_metadata_sptr **get_metadata**() **const**
Get metadata associated with this image.

virtual void **set_metadata**(video_metadata_sptr *md*)
Set metadata associated with this image.

Detections

template <typename *T*>
class kwiver::vital::**bounding_box**
Coordinate aligned bounding box.

This class represents a coordinate aligned box. The coordinate system places the origin in the upper left.

A bounding box must be constructed with the correct geometry. Once created, the geometry can not be altered.

Public Functions

bounding_box(vector_type **const** &*upper_left*, vector_type **const** &*lower_right*)
Create box from two corner points.

Parameters

- upper_left: Upper left corner of box.
- lower_right: Lower right corner of box.

bounding_box(vector_type **const** &*upper_left*, T **const** &*width*, T **const** &*height*)
Create box from point and dimensions.

Parameters

- upper_left: Upper left corner point
- width: Width of box.
- height: Height of box.

bounding_box(T *xmin*, T *ymin*, T *xmax*, T *ymax*)
Create a box from four coordinates.

Parameters

- xmin: Minimum x coordinate
- ymin: minimum y coordinate
- xmax: Maximum x coordinate
- ymax: Maximum y coordinate

vector_type **center**() **const**
Get center coordinate of box.

Return Center coordinate of box.

vector_type **upper_left**() **const**
Get upper left coordinate of box.

Return Upper left coordinate of box.

vector_type **lower_right**() **const**
Get lower right coordinate of box.

Return Lower right coordinate of box.

T **width**() **const**
Get width of box.

Return Width of box.

T **height**() **const**
Get height of box.

Return Height of box.

double **area**() **const**
Get area of box.

Return Area of box.

class kwiver::vital::**detected_object**
Detected object class.

This class represents a detected object in image space.

There is one object of this type for each detected object. These objects are defined by a bounding box in the image space. Each object has an optional classification object attached.

Public Functions

detected_object(**const** bounding_box_d &*bbox*, double *confidence* = 1.0, detected_object_type_sptr *classifications* = detected_object_type_sptr())
Create detected object with bounding box and other attributes.

Parameters

- bbox: Bounding box surrounding detected object, in image coordinates.
- confidence: Detectors confidence in this detection.
- classifications: Optional object classification.

detected_object_sptr **clone**() **const**
Create a deep copy of this object.

Return Managed copy of this object.

bounding_box_d **bounding_box**() **const**
Get bounding box from this detection.

The bounding box for this detection is returned. This box is in image coordinates. A default constructed (invalid) bounding box is returned if no box has been supplied for this detection.

Return A copy of the bounding box.

void **set_bounding_box**(**const** bounding_box_d &*bbox*)
Set new bounding box for this detection.

The supplied bounding box replaces the box for this detection.

Parameters

- bbox: Bounding box for this detection.

double **confidence**() **const**
Get confidence for this detection.

This method returns the current confidence value for this detection. Confidence values are in the range of 0.0 - 1.0.

Return Confidence value for this detection.

void **set_confidence**(double *d*)
Set new confidence value for detection.

This method sets a new confidence value for this detection. Confidence values are in the range of [0.0 - 1.0].

Parameters

- d: New confidence value for this detection.

uint64_t **index**() **const**
Get detection index.

This method returns the index for this detection.

The detection index is a general purpose field that the application can use to individually identify a detection. In some cases, this field can be used to correlate the detection of an object over multiple frames.

Return Detection index fof this detections.

void **set_index**(uint64_t *idx*)
Set detection index.

This method sets tne index value for this detection.

The detection index is a general purpose field that the application can use to individually identify a detection. In some cases, this field can be used to correlate the detection of an object over multiple frames.

Parameters

- idx: Detection index.

const std::string &**detector_name**() **const**
Get detector name.

This method returns the name of the detector that created this element. An empty string is returned if the detector name is not set.

Return Name of the detector.

void **set_detector_name**(**const** std::string &*name*)
Set detector name.

This method sets the name of the detector for this detection.

Parameters

- `name`: Detector name.

detected_object_type_sptr **type**()
Get pointer to optional classifications object.

This method returns the pointer to the classification object if there is one. If there is no classification object the pointer is NULL.

Return Pointer to classification object or NULL.

void **set_type**(detected_object_type_sptr *c*)
Set new classifications for this detection.

This method supplies a new set of class_names and scores for this detection.

Parameters

- `c`: New classification for this detection

image_container_sptr **mask**()
Get detection mask image.

This method returns the mask image associated with this detection.

Return Pointer to the mask image.

void **set_mask**(image_container_sptr *m*)
Set mask image for this detection.

This method supplies a new mask image for this detection.

Parameters

- `m`: Mask image

detected_object::descriptor_sptr **descriptor**() **const**
Get descriptor vector.

This method returns an optional descriptor vector that was used to create this detection. This is only set for certain object detectors.

Return Pointer to the descriptor vector.

void **set_descriptor**(descriptor_sptr *d*)
Set descriptor for this detection.

This method sets a descriptor vector that was used to create this detection. This is only set for certain object detectors.

Parameters

- d: Descriptor vector

class kwiver::vital::**detected_object_set**
Set of detected objects.

This class represents a ordered set of detected objects. The detections are ordered on their basic confidence value.

Reentrancy considerations: Typical usage for a set is for a single detector thread to create a set. It is possible to have an application where two threads are accessing the same set concurrently.

Inherits from kwiver::vital::noncopyable

Unnamed Group

detected_object_set::iterator **begin**()
Detected object set iterators;.

This method returns an iterator for the set of detected objects. The iterator points to a shared pointer to a detected object.

Return An iterator over the objects in this set;

Public Functions

detected_object_set()
Create an empty detection set.

This CTOR creates an empty detection set. Detections can be added with the *add()* method.

detected_object_set(std::vector<detected_object_sptr> **const** &*objs*)
Create new set of detected objects.

This CTOR creates a detection set using the supplied vector of detection objects. This can be used to create a new detection set from the output of a *select()* method.

Parameters

- objs: Vector of detected objects.

detected_object_set_sptr **clone**() **const**
Create deep copy.

This method creates a deep copy of this object.

Return Managed copy of this object.

void **add**(detected_object_sptr *object*)
Add detection to set.

This method adds a new detection to this set.

Parameters

- `object`: Detection to be added to set.

void **add**(detected_object_set_sptr *detections*)
Add detection set to set.

This method adds a new detection set to this set.

Parameters

- `detections`: Detection set to be added to set.

size_t **size**() **const**
Get number of detections in this set.

This method returns the number of detections in the set.

Return Number of detections.

bool **empty**() **const**
Returns whether or not this set is empty.

This method returns true if the set is empty, false otherwise.

Return Whether or not the set is empty.

detected_object_set_sptr **select**(double *threshold* = detected_object_type::INVALID_SCORE) **const**
Select detections based on confidence value.

This method returns a vector of detections ordered by confidence value, high to low. If the optional threshold is specified, then all detections from the set that are less than the threshold are not in the selected set. Note that the selected set may be empty.

The returned vector refers to the actual detections in the set, so if you make changes to the selected set, you are also changing the object in the set. If you want a clean set of detections, call *clone()* first.

Return List of detections.

Parameters

- `threshold`: Select all detections with confidence not less than this value. If this parameter is omitted, then all detections are selected.

detected_object_set_sptr **select**(**const** std::string &*class_name*, double *threshold* = detected_object_type::INVALID_SCORE) **const**
Select detections based on class_name.

This method returns a vector of detections that have the specified class_name. These detections are ordered by descending score for the name. Note that the selected set may be empty.

The returned vector refers to the actual detections in the set, so if you make changes to the selected set, you are also changing the object in the set. If you want a clean set of detections, call *clone()* first.

Return List of detections.

Parameters

- `class_name`: class name
- `threshold`: Select all detections with confidence not less than this value. If this parameter is omitted, then all detections with the label are selected.

void **`scale`**(double *scale_factor*)
Scale all detection locations by some scale factor.

This method changes the bounding boxes within all stored detections by scaling them by some scale factor.

Parameters

- `scale`: Scale factor

void **`shift`**(double *col_shift*, double *row_shift*)
Shift all detection locations by some translation offset.

This method shifts the bounding boxes within all stored detections by a supplied column and row shift.

Note: Detections in this set can be shared by multiple sets, so shifting the detections in this set will also shift the detection in other sets that share this detection. If this is going to be a problem, *clone()* this set before shifting.

Parameters

- `col_shift`: Column (a.k.a. x, i, width) translation factor
- `row_shift`: Row (a.k.a. y, j, height) translation factor

kwiver::vital::attribute_set_sptr **`attributes`**() **const**
Get attributes set.

This method returns a pointer to the attribute set that is attached to this object. It is possible that the pointer is NULL, so check before using it.

Return Pointer to attribute set or NULL

void **`set_attributes`**(attribute_set_sptr *attrs*)
Attach attributes set to this object.

This method attaches the specified attribute set to this object.

Parameters

- `attrs`: Pointer to attribute set to attach.

Other

class `kwiver::vital::`**`camera`**
An abstract representation of camera.

The base class of cameras is abstract and provides a double precision interface. The templated derived class can store values in either single or double precision.

Subclassed by kwiver::vital::simple_camera

Public Functions

virtual ~camera()
Destructor.

virtual camera_sptr **clone**() **const** = 0
Create a clone of this camera object.

virtual vector_3d **center**() **const** = 0
Accessor for the camera center of projection (position)

virtual vector_3d **translation**() **const** = 0
Accessor for the translation vector.

virtual covariance_3d **center_covar**() **const** = 0
Accessor for the covariance of camera center.

virtual rotation_d **rotation**() **const** = 0
Accessor for the rotation.

virtual camera_intrinsics_sptr **intrinsics**() **const** = 0
Accessor for the intrinsics.

virtual camera_sptr **clone_look_at**(**const** vector_3d &*stare_point*, **const** vector_3d &*up_direction* = vector_3d::UnitZ()) **const** = 0
Create a clone of this camera that is rotated to look at the given point.

Return New clone, but set to look at the given point.

Parameters

- `stare_point`: the location at which the camera is oriented to point
- `up_direction`: the vector which is "up" in the world (defaults to Z-axis)

matrix_3x4d **as_matrix**() **const**
Convert to a 3x4 homogeneous projection matrix.

Note This matrix representation does not account for lens distortion models that may be used in the *camera_intrinsics*

vector_2d **project**(**const** vector_3d &*pt*) **const**
Project a 3D point into a 2D image point.

double **depth**(**const** vector_3d &*pt*) **const**
Compute the distance of the 3D point to the image plane.

Points with negative depth are behind the camera

class `kwiver::vital::camera_intrinsics`
An abstract representation of camera intrinsics.

Subclassed by kwiver::vital::simple_camera_intrinsics

Public Functions

virtual ~camera_intrinsics()
Destructor.

virtual camera_intrinsics_sptr **clone**() **const** = 0
Create a clone of this object.

virtual double **focal_length**() **const** = 0
Access the focal length.

virtual vector_2d **principal_point**() **const** = 0
Access the principal point.

virtual double **aspect_ratio**() **const** = 0
Access the aspect ratio.

virtual double **skew**() **const** = 0
Access the skew.

virtual std::vector<double> **dist_coeffs**() **const**
Access the distortion coefficients.

matrix_3x3d **as_matrix**() **const**
Access the intrinsics as an upper triangular matrix.

Convert to a 3x3 calibration matrix.

Note This matrix includes the focal length, principal point, aspect ratio, and skew, but does not model distortion

vector_2d **map**(**const** vector_2d &*norm_pt*) **const**
Map normalized image coordinates into actual image coordinates.

This function applies both distortion and application of the calibration matrix to map into actual image coordinates

vector_2d **map**(**const** vector_3d &*norm_hpt*) **const**
Map a 3D point in camera coordinates into actual image coordinates.

vector_2d **unmap**(**const** vector_2d &*norm_pt*) **const**
Unmap actual image coordinates back into normalized image coordinates.

This function applies both application of the inverse calibration matrix and undistortion of the normalized coordinates.

virtual vector_2d **distort**(**const** vector_2d &*norm_pt*) **const**
Map normalized image coordinates into distorted coordinates.

The default implementation is the identity transformation (no distortion)

virtual vector_2d **undistort**(**const** vector_2d &*dist_pt*) **const**
Unmap distorted normalized coordinates into normalized coordinates.

The default implementation is the identity transformation (no distortion)

struct kwiver::vital::**rgb_color**
Struct to represent an RGB tuple.

Public Functions

rgb_color()
Default constructor - set the color to white.

rgb_color(uint8_t **const** &*cr*, uint8_t **const** &*cg*, uint8_t **const** &*cb*)
Constructor.

rgb_color(*rgb_color* **const** &*c*)
Copy Constructor.

template <**class** Archive>
void **serialize**(Archive &*archive*)
Serialization of the class data.

template <unsigned *N*, typename *T*>
class kwiver::vital::**covariance_**
A representation of covariance of a measurement.

Public Functions

covariance_()
Default Constructor - Initialize to identity.

covariance_(**const** *covariance_*<N, T> &*other*)
Copy constructor.

template <**typename** U>
covariance_(**const** *covariance_*<N, U> &*other*)
Copy Constructor from another type.

covariance_(**const** T &*value*)
Constructor - initialize to identity matrix times a scalar.

covariance_(**const** Eigen::Matrix<T, N, N> &*mat*)
Constructor - from a matrix.

Averages off diagonal elements to enforce symmetry

Parameters

- mat: matrix to construct from.

*covariance_*<N, T> &**operator=**(**const** *covariance_*<N, T> &*other*)
Assignment operator.

Eigen::Matrix<T, N, N> **matrix**() **const**
Extract a full matrix.

T &**operator()**(unsigned int *i*, unsigned int *j*)
Return the i-th row, j-th column.

const T &**operator()**(unsigned int *i*, unsigned int *j*) **const**
Return the i-th row, j-th column (const)

const T ***data**() **const**
Access the underlying data.

bool **operator==**(*covariance_*<N, T> **const** &*other*) **const**
Equality operator.

bool **operator!=**(*covariance_*<N, T> **const** &*other*) **const**
Inequality operator.

template <**class** Archive>
void **serialize**(Archive &*archive*)
Serialization of the class data.

Public Static Attributes

const unsigned int **data_size** = N * (N + 1) / 2
Number of unique values in a NxN symmetric matrix.

class kwiver::vital::**descriptor**
A representation of a feature descriptor used in matching.

Subclassed by kwiver::vital::descriptor_array_of< T >

Public Functions

virtual **~descriptor**()
Destructor.

virtual std::type_info **const** &**data_type**() **const** = 0
Access the type info of the underlying data (double or float)

virtual std::size_t **size**() **const** = 0
The number of elements of the underlying type.

virtual std::size_t **num_bytes**() **const** = 0
The number of bytes used to represent the data.

virtual std::vector<byte> **as_bytes**() **const** = 0
Return the descriptor as a vector of bytes.

This should always work, even if the underlying type is not bytes

virtual std::vector<double> **as_double**() **const** = 0
Return the descriptor as a vector of doubles.

Return an empty vector if this makes no sense for the underlying type.

bool **operator==**(*descriptor* **const** &*other*) **const**
Equality operator.

bool **operator!=**(*descriptor* **const** &*other*) **const**
Inequality operator.

class kwiver::vital::**descriptor_request**
A representation of a descriptor request.

This is used by some arbitrary GUI to request and return computed descriptors on some region of the input imagery.

class kwiver::vital::**descriptor_set**
An abstract ordered collection of feature descriptors.

The base class *descriptor_set* is abstract and provides an interface for returning a vector of descriptors. There is a simple derived class that stores the data as a vector of descriptors and returns it. Other derived classes can store the data in other formats and convert on demand.

Subclassed by kwiver::arrows::ocv::descriptor_set, kwiver::arrows::vcl::descriptor_set, kwiver::vital::simple_descriptor_set

Public Functions

virtual ~descriptor_set()
Destructor.

virtual size_t **size**() **const** = 0
Return the number of descriptors in the set.

virtual std::vector<descriptor_sptr> **descriptors**() **const** = 0
Return a vector of descriptor shared pointers.

Algorithms

Base Types

class kwiver::vital::**algorithm**
An abstract base class for all algorithms.

This class is an abstract base class for all algorithm implementations.

Subclassed by *kwiver::vital::algorithm_def< analyze_tracks >*, *kwiver::vital::algorithm_def< bundle_adjust >*, *kwiver::vital::algorithm_def< close_loops >*, *kwiver::vital::algorithm_def< compute_ref_homography >*, *kwiver::vital::algorithm_def< compute_stereo_depth_map >*, *kwiver::vital::algorithm_def< compute_track_descriptors >*, *kwiver::vital::algorithm_def< convert_image >*, *kwiver::vital::algorithm_def< detect_features >*, *kwiver::vital::algorithm_def< detected_object_filter >*, *kwiver::vital::algorithm_def< detected_object_set_input >*, *kwiver::vital::algorithm_def< detected_object_set_output >*, *kwiver::vital::algorithm_def< draw_detected_object_set >*, *kwiver::vital::algorithm_def< draw_tracks >*, *kwiver::vital::algorithm_def< dynamic_configuration >*, *kwiver::vital::algorithm_def< estimate_canonical_transform >*, *kwiver::vital::algorithm_def< estimate_essential_matrix >*, *kwiver::vital::algorithm_def< estimate_fundamental_matrix >*, *kwiver::vital::algorithm_def< estimate_homography >*, *kwiver::vital::algorithm_def< estimate_similarity_transform >*, *kwiver::vital::algorithm_def< extract_descriptors >*, *kwiver::vital::algorithm_def< feature_descriptor_io >*, *kwiver::vital::algorithm_def< filter_features >*, *kwiver::vital::algorithm_def< filter_tracks >*, *kwiver::vital::algorithm_def< formulate_query >*, *kwiver::vital::algorithm_def< image_filter >*, *kwiver::vital::algorithm_def< image_io >*, *kwiver::vital::algorithm_def< image_object_detector >*, *kwiver::vital::algorithm_def< initialize_cameras_landmarks >*, *kwiver::vital::algorithm_def< match_features >*, *kwiver::vital::algorithm_def< optimize_cameras >*, *kwiver::vital::algorithm_def< refine_detections >*, *kwiver::vital::algorithm_def< split_image >*, *kwiver::vital::algorithm_def< track_descriptor_set_input >*, *kwiver::vital::algorithm_def< track_descriptor_set_output >*, *kwiver::vital::algorithm_def< track_features >*, *kwiver::vital::algorithm_def< train_detector >*, *kwiver::vital::algorithm_def< triangulate_landmarks >*, *kwiver::vital::algorithm_def< uuid_factory >*, *kwiver::vital::algorithm_def< video_input >*, *kwiver::vital::algorithm_def< Self >*

Public Functions

virtual std::string **type_name**() **const** = 0
Return the name of the base algorithm.

std::string **impl_name**() **const**
Return the name of this implementation.

config_block_sptr **get_configuration**() **const**
Get this algorithm's configuration block .

Get this alg's configuration block .

This method returns the required configuration for the algorithm. The implementation of this method should be light-weight and only create and fill in the config block.

This base virtual function implementation returns an empty configuration.

Return `config_block` containing the configuration for this algorithm and any nested components.

virtual void **set_configuration**(config_block_sptr *config*) = 0
Set this algorithm's properties via a config block.

This method is called to pass a configuration to the algorithm. The implementation of this method should be light-weight and only save the necessary config values. Defer any substantial processing in another method.

Exceptions

- `no_such_configuration_value_exception`: Thrown if an expected configuration value is not present.
- `algorithm_configuration_exception`: Thrown when the algorithm is given an invalid `config_block` or is otherwise unable to configure itself.

Parameters

- `config`: The `config_block` instance containing the configuration parameters for this algorithm

virtual bool **check_configuration**(config_block_sptr *config*) **const** = 0
Check that the algorithm's configuration config_block is valid.

This checks solely within the provided `config_block` and not against the current state of the instance. This isn't static for inheritance reasons.

Return true if the configuration check passed and false if it didn't.

Parameters

- `config`: The config block to check configuration of.

Public Static Functions

void **get_nested_algo_configuration**(std::string **const** &*type_name*, std::string **const** &*name*, config_block_sptr *config*, algorithm_sptr *nested_algo*)
Helper function for properly getting a nested algorithm's configuration.

Adds a configurable algorithm implementation switch for this algorithm. If the variable pointed to by `nested_algo` is a defined sptr to an implementation, its configuration parameters are merged with the given config_block .

Parameters

- `type_name`: The type name of the nested algorithm.
- `name`: An identifying name for the nested algorithm
- `config`: The `config_block` instance in which to put the nested algorithm's configuration.
- `nested_algo`: The nested algorithm's sptr variable.

void **set_nested_algo_configuration**(std::string **const** &*type_name*, std::string **const** &*name*, config_block_sptr *config*, algorithm_sptr &*nested_algo*)

Helper function for properly setting a nested algorithm's configuration.

Helper method for properly setting a nested algorithm's configuration.

If the value for the config parameter "type" is supported by the concrete algorithm class, then a new algorithm object is created, configured and returned via the `nested_algo` pointer.

The nested algorithm will not be set if the implementation switch (as defined in the `get_nested_algo_configuration`) is not present or set to an invalid value relative to the registered names for this `type_name`

Parameters

- `type_name`: The type name of the nested algorithm.
- `name`: Config block name for the nested algorithm.
- `config`: The `config_block` instance from which we will draw configuration needed for the nested algorithm instance.
- `nested_algo`: The nested algorithm's sptr variable.

bool **check_nested_algo_configuration**(std::string **const** &*type_name*, std::string **const** &*name*, config_block_sptr *config*)

Helper function for checking that basic nested algorithm configuration is valid.

Helper method for checking that basic nested algorithm configuration is valid.

Check that the expected implementation switch exists and that its value is registered implementation name.

If the name is valid, we also recursively call *check_configuration()* on the set implementation. This is done with a fresh create so we don't have to rely on the implementation being defined in the instance this is called from.

Parameters

- `type_name`: The type name of the nested algorithm.
- `name`: An identifying name for the nested algorithm.
- `config`: The `config_block` to check.

template <typename *Self*>

class `kwiver::vital::algorithm_def`

An intermediate templated base class for algorithm definition.

Uses the curiously recurring template pattern (CRTP) to declare the clone function and automatically provide functions to register algorithm, and create new instance by name. Each algorithm definition should be declared as shown below

```
class my_algo_def
: public algorithm_def<my_algo_def>
{
  ...
};
```

See algorithm_impl

Inherits from *kwiver::vital::algorithm*

Public Types

typedef std::shared_ptr<Self> **base_sptr**

Shared pointer type of the templated *vital::algorithm_def* class.

Public Functions

virtual std::string **type_name**() **const**

Return the name of this algorithm.

Public Static Functions

static *base_sptr* **create**(std::string **const** &*impl_name*)

Factory method to make an instance of this algorithm by impl_name.

static std::vector<std::string> **registered_names**()

Return a vector of the impl_name of each registered implementation.

static void **get_nested_algo_configuration**(std::string **const** &*name*, config_block_sptr *config*, *base_sptr nested_algo*)

Helper function for properly getting a nested algorithm's configuration.

Adds a configurable algorithm implementation switch for this *algorithm_def*. If the variable pointed to by `nested_algo` is a defined sptr to an implementation, its configuration parameters are merged with the given config_block .

Parameters

- `name`: An identifying name for the nested algorithm
- `config`: The `config_block` instance in which to put the nested algorithm's configuration.
- `nested_algo`: The nested algorithm's sptr variable.

static void **set_nested_algo_configuration**(std::string **const** &*name*, config_block_sptr *config*, *base_sptr* &*nested_algo*)

Instantiate nested algorithm.

A new concrete algorithm object is created if the value for the config parameter "type" is supported. The new object is returned through the nested_algo parameter.

The nested algorithm will not be set if the implementation switch (as defined in the `get_nested_algo_configuration`) is not present or set to an invalid value relative to the registered names for this *`algorithm_def`*.

Parameters

- `name`: Config block name for the nested algorithm.
- `config`: The `config_block` instance from which we will draw configuration needed for the nested algorithm instance.
- `nested_algo`: Pointer to the algorithm object is returned here.

static bool **check_nested_algo_configuration**(std::string **const** &*name*, config_block_sptr *config*)

Helper function for checking that basic nested algorithm configuration is valid.

Check that the expected implementation switch exists and that its value is registered implementation name.

If the name is valid, we also recursively call *check_configuration()* on the set implementation. This is done with a fresh create so we don't have to rely on the implementation being defined in the instance this is called from.

Parameters

- `name`: An identifying name for the nested algorithm.
- `config`: The `config_block` to check.

Functionality

class `kwiver::vital::algo::`**analyze_tracks**

Abstract base class for writing out human readable track statistics.

Inherits from *kwiver::vital::algorithm_def< analyze_tracks >*

Subclassed by kwiver::vital::algorithm_impl< analyze_tracks, vital::algo::analyze_tracks >

Public Functions

virtual void **print_info**(kwiver::vital::track_set_sptr *track_set*, stream_t &*stream* = std::cout) **const** = 0

Output various information about the tracks stored in the input set.

Parameters

- `track_set`: the tracks to analyze
- `stream`: an output stream to write data onto

Public Static Functions

static std::string **static_type_name**()
Return the name of this algorithm.

class kwiver::vital::algo::**bundle_adjust**
An abstract base class for bundle adjustment using feature tracks.

Inherits from *kwiver::vital::algorithm_def< bundle_adjust >*

Subclassed by kwiver::vital::algorithm_impl< bundle_adjust, vital::algo::bundle_adjust >, kwiver::vital::algorithm_impl< hierarchical_bundle_adjust, vital::algo::bundle_adjust >

Public Types

typedef std::function<bool (kwiver::vital::camera_map_sptr, kwiver::vital::landmark_map_sptr)> **callback_t**
Typedef for the callback function signature.

Public Functions

virtual void **optimize**(kwiver::vital::camera_map_sptr *&cameras*, kwiver::vital::landmark_map_sptr *&landmarks*, kwiver::vital::feature_track_set_sptr *tracks*, kwiver::vital::video_metadata_map_sptr *metadata* = nullptr) **const** = 0
Optimize the camera and landmark parameters given a set of feature tracks.

Implementations of this function should not modify the underlying objects contained in the input structures. Output references should either be new instances or the same as input.

Parameters

- `cameras`: the cameras to optimize
- `landmarks`: the landmarks to optimize
- `tracks`: the feature tracks to use as constraints
- `metadata`: the frame metadata to use as constraints

void **set_callback**(*callback_t cb*)
Set a callback function to report intermediate progress.

Public Static Functions

static std::string **static_type_name**()
Return the name of this algorithm.

class kwiver::vital::algo::**close_loops**
Abstract base class for loop closure algorithms.

Different algorithms can perform loop closure in a variety of ways, either in attempt to make either short or long term closures. Similarly to *track_features*, this class is designed to be called in an online fashion.

Inherits from *kwiver::vital::algorithm_def< close_loops >*

Subclassed by kwiver::vital::algorithm_impl< close_loops_bad_frames_only, vital::algo::close_loops >, kwiver::vital::algorithm_impl< close_loops_exhaustive, vital::algo::close_loops >, kwiver::vital::algorithm_impl< close_loops_keyframe, vital::algo::close_loops >, kwiver::vital::algorithm_impl< close_loops_multi_method, vital::algo::close_loops >, kwiver::vital::algorithm_impl< vxl::close_loops_homography_guided, vital::algo::close_loops >

Public Functions

virtual kwiver::vital::feature_track_set_sptr **stitch**(kwiver::vital::frame_id_t *frame_number*, kwiver::vital::feature_track_set_sptr *input*, kwiver::vital::image_container_sptr *image*, kwiver::vital::image_container_sptr *mask* = kwiver::vital::image_container_sptr ()) **const** = 0

Attempt to perform closure operation and stitch tracks together.

Return an updated set of feature tracks after the stitching operation

Parameters

- `frame_number`: the frame number of the current frame
- `input`: the input feature track set to stitch
- `image`: image data for the current frame
- `mask`: Optional mask image where positive values indicate regions to consider in the input image.

Public Static Functions

static std::string **static_type_name**()

Return the name of this algorithm.

class `kwiver::vital::algo::`**compute_ref_homography**

Abstract base class for mapping each image to some reference image.

This class differs from estimate_homographies in that estimate_homographies simply performs a homography regression from matching feature points. This class is designed to generate different types of homographies from input feature tracks, which can transform each image back to the same coordinate space derived from some initial refrerence image.

Inherits from *kwiver::vital::algorithm_def< compute_ref_homography >*

Subclassed by kwiver::vital::algorithm_impl< compute_ref_homography_core, vital::algo::compute_ref_homography >

Public Functions

virtual kwiver::vital::f2f_homography_sptr **estimate**(kwiver::vital::frame_id_t *frame_number*, kwiver::vital::feature_track_set_sptr *tracks*) **const** = 0

Estimate the transformation which maps some frame to a reference frame.

Similarly to *track_features*, this class was designed to be called in an online fashion for each sequential frame. The output homography will contain a transformation mapping points from the current frame (with frame_id frame_number) to the earliest possible reference frame via post multiplying points on the current frame with the computed homography.

The returned homography is internally allocated and passed back through a smart pointer transferring ownership of the memory to the caller.

Return estimated homography

Parameters

- `frame_number`: frame identifier for the current frame
- `tracks`: the set of all tracked features from the image

Public Static Functions

static std::string **`static_type_name`**()
Return the name of this algorithm.

class `kwiver::vital::algo::`**`compute_stereo_depth_map`**
An abstract base class for detecting feature points.

Inherits from *kwiver::vital::algorithm_def< compute_stereo_depth_map >*

Public Functions

virtual kwiver::vital::image_container_sptr **`compute`**(kwiver::vital::image_container_sptr *left_image*, kwiver::vital::image_container_sptr *right_image*) **const** = 0
Compute a stereo depth map given two images.

Return a depth map image

Exceptions

- `image_size_mismatch_exception`: When the given input image sizes do not match.

Parameters

- `left_image`: contains the first image to process
- `right_image`: contains the second image to process

Public Static Functions

static std::string **`static_type_name`**()
Return the name of this algorithm.

class `kwiver::vital::algo::`**`compute_track_descriptors`**
An abstract base class for computing track descriptors.

Inherits from *kwiver::vital::algorithm_def< compute_track_descriptors >*

Subclassed by kwiver::vital::algorithm_impl< burnout_track_descriptors, vital::algo::compute_track_descriptors >

Public Functions

virtual kwiver::vital::track_descriptor_set_sptr **compute**(kwiver::vital::image_container_sptr *image_data*, kwiver::vital::track_set_sptr *tracks*) = 0

Compute track descriptors given an image and tracks.

Return a set of track descriptors

Parameters

- `image_data`: contains the image data to process
- `tracks`: the tracks to extract descriptors around

Public Static Functions

static std::string **static_type_name**()

Return the name of this algorithm.

class `kwiver::vital::algo::convert_image`

An abstract base class for converting base image type.

Inherits from *kwiver::vital::algorithm_def< convert_image >*

Subclassed by kwiver::vital::algorithm_impl< convert_image, vital::algo::convert_image >, kwiver::vital::algorithm_impl< convert_image_bypass, vital::algo::convert_image >

Public Functions

void **set_configuration**(kwiver::vital::config_block_sptr *config*)

Set this algorithm's properties via a config block.

bool **check_configuration**(kwiver::vital::config_block_sptr *config*) **const**

Check that the algorithm's currently configuration is valid.

Check that the algorithm's current configuration is valid.

virtual kwiver::vital::image_container_sptr **convert**(kwiver::vital::image_container_sptr *img*) **const** = 0

Convert image base type.

Public Static Functions

static std::string **static_type_name**()

Return the name of this algorithm.

class `kwiver::vital::algo::detect_features`

An abstract base class for detecting feature points.

Inherits from *kwiver::vital::algorithm_def< detect_features >*

Subclassed by kwiver::vital::algorithm_impl< detect_features, vital::algo::detect_features >, kwiver::arrows::ocv::detect_features

Public Functions

virtual kwiver::vital::feature_set_sptr **detect**(kwiver::vital::image_container_sptr *image_data*, kwiver::vital::image_container_sptr *mask* = kwiver::vital::image_container_sptr ()) **const** = 0

Extract a set of image features from the provided image.

A given mask image should be one-channel (mask->depth() == 1). If the given mask image has more than one channel, only the first will be considered.

Return a set of image features

Exceptions

- `image_size_mismatch_exception`: When the given non-zero mask image does not match the size of the dimensions of the given image data.

Parameters

- `image_data`: contains the image data to process
- `mask`: Mask image where regions of positive values (boolean true) indicate regions to consider. Only the first channel will be considered.

Public Static Functions

static std::string **static_type_name**()

Return the name of this algorithm.

class `kwiver::vital::algo::detected_object_filter`

An abstract base class for filtering sets of detected objects.

A detected object filter accepts a set of detections and produces another set of detections. The output set may be different from the input set. It all depends on the actual implementation. In any case, the input detection set shall be unmodified.

Inherits from *kwiver::vital::algorithm_def< detected_object_filter >*

Subclassed by kwiver::vital::algorithm_impl< class_probablity_filter, vital::algo::detected_object_filter >

Public Functions

virtual detected_object_set_sptr **filter**(**const** detected_object_set_sptr *input_set*) **const** = 0

Filter set of detected objects.

This method applies a filter to the input set to create an output set. The input set of detections is unmodified.

Return Filtered set of detections.

Parameters

- `input_set`: Set of detections to be filtered.

Public Static Functions

static std::string **static_type_name**()
Return the name of this algorithm.

class kwiver::vital::algo::**detected_object_set_input**
Read detected object sets.

This class is the abstract base class for the detected object set writer.

Detection sets from multiple images are stored in a single file with enough information to recreate a unique image identifier, usually the file name, and an associated set of detections.

Inherits from *kwiver::vital::algorithm_def< detected_object_set_input >*

Subclassed by kwiver::vital::algorithm_impl< detected_object_set_input_csv, vital::algo::detected_object_set_input >, kwiver::vital::algorithm_impl< detected_object_set_input_kw18, vital::algo::detected_object_set_input >

Public Functions

void **open**(std::string **const** &*filename*)
Open a file of detection sets.

This method opens a detection set file for reading.

Parameters

- `filename`: Name of file to open

Exceptions

- `kwiver::vital::path_not_exists`: Thrown when the given path does not exist.
- `kwiver::vital::path_not_a_file`: Thrown when the given path does not point to a file (i.e. it points to a directory).
- `kwiver::vital::file_not_found_exception`:

void **use_stream**(std::istream **strm*)
Read detections from an existing stream.

This method specifies the input stream to use for reading detections. Using a stream is handy when the detections are available in a stream format.

Parameters

- `strm`: input stream to use

void **close**()
Close detection set file.

The currently open detection set file is closed. If there is no currently open file, then this method does nothing.

virtual bool **read_set**(kwiver::vital::detected_object_set_sptr &*set*, std::string &*image_name*) = 0
Read next detected object set.

This method reads the next set of detected objects from the file. **False** is returned when the end of file is reached.

Return **true** if detections are returned, **false** if end of file.

Parameters

- `set`: Pointer to the new set of detections. Set may be empty if there are no detections on an image.
- `image_name`: Name of the image that goes with the detections. This string may be empty depending on the source format.

bool **`at_eof`**`()` **const**
Determine if input file is at end of file.

This method reports the end of file status for a file open for reading.

Return **true** if file is at end.

Public Static Functions

static std::string **`static_type_name`**`()`
Return the name of this algorithm.

class `kwiver::vital::algo::`**`detected_object_set_output`**
Read and write detected object sets.

This class is the abstract base class for the detected object set reader and writer.

Detection sets from multiple images are stored in a single file with enough information to recreate a unique image identifier, usually the file name, and an associated wet of detections.

Inherits from *kwiver::vital::algorithm_def< detected_object_set_output >*

Subclassed by kwiver::vital::algorithm_impl< detected_object_set_output_csv, vital::algo::detected_object_set_output >, kwiver::vital::algorithm_impl< detected_object_set_output_kw18, vital::algo::detected_object_set_output >, kwiver::vital::algorithm_impl< matlab_detection_output, vital::algo::detected_object_set_output >

Public Functions

void **`open`**(std::string **const** &*filename*)
Open a file of detection sets.

This method opens a detection set file for writing.

Parameters

- `filename`: Name of file to open

Exceptions

- `kwiver::vital::path_not_exists`: Thrown when the given path does not exist.
- `kwiver::vital::path_not_a_file`: Thrown when the given path does not point to a file (i.e. it points to a directory).

void **use_stream**(std::ostream *_strm_)
Write detections to an existing stream.

This method specifies the output stream to use for writing detections. Using a stream is handy when the detections output is available in a stream format.

Parameters

- `strm`: output stream to use

void **close**()
Close detection set file.

The currently open detection set file is closed. If there is no currently open file, then this method does nothing.

virtual void **write_set**(**const** kwiver::vital::detected_object_set_sptr *set*, std::string **const** &*image_path*) = 0
Write detected object set.

This method writes the specified detected object set and image name to the currently open file.

Parameters

- `set`: Detected object set
- `image_path`: File path to image associated with the detections.

Public Static Functions

static std::string **static_type_name**()
Return the name of this algorithm.

class `kwiver::vital::algo::`**draw_detected_object_set**
An abstract base class for algorithms which draw tracks on top of images in various ways, for analyzing results.

Inherits from *kwiver::vital::algorithm_def< draw_detected_object_set >*

Subclassed by kwiver::vital::algorithm_impl< draw_detected_object_set, vital::algo::draw_detected_object_set >

Public Functions

virtual kwiver::vital::image_container_sptr **draw**(kwiver::vital::detected_object_set_sptr *detected_set*, kwiver::vital::image_container_sptr *image*) = 0
Draw detected object boxes on Image.

This method draws the detections on a copy of the image. The input image is unmodified. The actual boxes that are drawn are controlled by the configuration for the implementation.

Return Image with boxes and other annotations added.

Parameters

- `detected_set`: Set of detected objects
- `image`: Boxes are drawn in this image

Public Static Functions

static std::string **static_type_name**()
Return the name of this algorithm.

class kwiver::vital::algo::**draw_tracks**
An abstract base class for algorithms which draw tracks on top of images in various ways, for analyzing results.

Inherits from *kwiver::vital::algorithm_def< draw_tracks >*

Subclassed by kwiver::vital::algorithm_impl< draw_tracks, vital::algo::draw_tracks >

Public Functions

virtual kwiver::vital::image_container_sptr **draw**(kwiver::vital::track_set_sptr *display_set*, kwiver::vital::image_container_sptr_list *image_data*, kwiver::vital::track_set_sptr *comparison_set* = kwiver::vital::track_set_sptr ()) = 0
Draw features tracks on top of the input images.

This process can either be called in an offline fashion, where all tracks and images are provided to the function on the first call, or in an online fashion where only new images are provided on sequential calls. This function can additionally consume a second track set, which can optionally be used to display additional information to provide a comparison between the two track sets.

Return a pointer to the last image generated

Parameters

- `display_set`: the main track set to draw
- `image_data`: a list of images the tracks were computed over
- `comparison_set`: optional comparison track set

Public Static Functions

static std::string **static_type_name**()
Return the name of this algorithm.

class kwiver::vital::algo::**dynamic_configuration**
Abstract algorithm for getting dynamic configuration values from an external source. This class represents an interface to an external source of configuration values. A typical application would be an external U.I. control that is desired to control the performance of an algorithm by varying some of its configuration values.

Inherits from *kwiver::vital::algorithm_def< dynamic_configuration >*

Subclassed by kwiver::vital::algorithm_impl< dynamic_config_none, vital::algo::dynamic_configuration >

Public Functions

virtual void **set_configuration**(config_block_sptr *config*) = 0
Set this algorithm's properties via a config block.

This method is called to pass a configuration to the algorithm. The implementation of this method should be light-weight and only save the necessary config values. Defer any substantial processing in another method.

Exceptions

- `no_such_configuration_value_exception`: Thrown if an expected configuration value is not present.
- `algorithm_configuration_exception`: Thrown when the algorithm is given an invalid `config_block` or is otherwise unable to configure itself.

Parameters

- `config`: The `config_block` instance containing the configuration parameters for this algorithm

virtual bool **`check_configuration`**(config_block_sptr *config*) **const** = 0

Check that the algorithm's configuration config_block is valid.

This checks solely within the provided `config_block` and not against the current state of the instance. This isn't static for inheritance reasons.

Return true if the configuration check passed and false if it didn't.

Parameters

- `config`: The config block to check configuration of.

virtual config_block_sptr **`get_dynamic_configuration`**() = 0

Return dynamic configuration values.

This method returns dynamic configuration values. a valid config block is returned even if there are not values being returned.

class `kwiver::vital::algo::`**`estimate_canonical_transform`**

Algorithm for estimating a canonical transform for cameras and landmarks.

A canonical transform is a repeatable transformation that can be recovered from data. In this case we assume at most a similarity transformation. If data sets P1 and P2 are equivalent up to a similarity transformation, then applying a canonical transform to P1 and separately a canonical transform to P2 should bring the data into the same coordinates.

Inherits from *kwiver::vital::algorithm_def< estimate_canonical_transform >*

Subclassed by kwiver::vital::algorithm_impl< estimate_canonical_transform, vital::algo::estimate_canonical_transform >

Public Functions

virtual kwiver::vital::similarity_d **`estimate_transform`**(kwiver::vital::camera_map_sptr **const** *cameras*, kwiver::vital::landmark_map_sptr **const** *landmarks*) **const** = 0

Estimate a canonical similarity transform for cameras and points.

Return An estimated similarity transform mapping the data to the canonical space.

Note This algorithm does not apply the transformation, it only estimates it.

Parameters

- `cameras`: The camera map containing all the cameras
- `landmarks`: The landmark map containing all the 3D landmarks

Exceptions

- `algorithm_exception`: When the data is insufficient or degenerate.

Public Static Functions

static std::string **`static_type_name`**()
Name of this algo definition.

class `kwiver::vital::algo::`**`estimate_essential_matrix`**
An abstract base class for estimating an essential matrix from matching 2D points.

Inherits from *kwiver::vital::algorithm_def< estimate_essential_matrix >*

Subclassed by kwiver::vital::algorithm_impl< estimate_essential_matrix, vital::algo::estimate_essential_matrix >

Public Functions

essential_matrix_sptr **`estimate`**(**const** kwiver::vital::feature_set_sptr *feat1*, **const** kwiver::vital::feature_set_sptr *feat2*, **const** kwiver::vital::match_set_sptr *matches*, **const** kwiver::vital::camera_intrinsics_sptr *cal1*, **const** kwiver::vital::camera_intrinsics_sptr *cal2*, std::vector<bool> &*inliers*, double *inlier_scale* = 1.0) **const**
Estimate an essential matrix from corresponding features.

Parameters

- `feat1`: the set of all features from the first image
- `feat2`: the set of all features from the second image
- `matches`: the set of correspondences between *feat1* and *feat2*
- `cal1`: the intrinsic parameters of the first camera
- `cal2`: the intrinsic parameters of the second camera
- `inliers`: for each point pair, the value is true if this pair is an inlier to the estimate
- `inlier_scale`: error distance tolerated for matches to be inliers

essential_matrix_sptr **`estimate`**(**const** kwiver::vital::feature_set_sptr *feat1*, **const** kwiver::vital::feature_set_sptr *feat2*, **const** kwiver::vital::match_set_sptr *matches*, **const** kwiver::vital::camera_intrinsics_sptr *cal*, std::vector<bool> &*inliers*, double *inlier_scale* = 1.0) **const**
Estimate an essential matrix from corresponding features.

Parameters

- `feat1`: the set of all features from the first image

- `feat2`: the set of all features from the second image
- `matches`: the set of correspondences between *feat1* and *feat2*
- `cal`: the intrinsic parameters, same for both cameras
- `inliers`: for each point pair, the value is true if this pair is an inlier to the estimate
- `inlier_scale`: error distance tolerated for matches to be inliers

virtual kwiver::vital::essential_matrix_sptr **`estimate`**(**const** std::vector<kwiver::vital::vector_2d> &*pts1*, **const** std::vector<kwiver::vital::vector_2d> &*pts2*, **const** kwiver::vital::camera_intrinsics_sptr *cal*, std::vector<bool> &*inliers*, double *inlier_scale* = 1.0) **const**

Estimate an essential matrix from corresponding points.

Parameters

- `pts1`: the vector or corresponding points from the first image
- `pts2`: the vector of corresponding points from the second image
- `cal`: the intrinsic parameters, same for both cameras
- `inliers`: for each point pair, the value is true if this pair is an inlier to the estimate
- `inlier_scale`: error distance tolerated for matches to be inliers

virtual kwiver::vital::essential_matrix_sptr **`estimate`**(**const** std::vector<kwiver::vital::vector_2d> &*pts1*, **const** std::vector<kwiver::vital::vector_2d> &*pts2*, **const** kwiver::vital::camera_intrinsics_sptr *cal1*, **const** kwiver::vital::camera_intrinsics_sptr *cal2*, std::vector<bool> &*inliers*, double *inlier_scale* = 1.0) **const** = 0

Estimate an essential matrix from corresponding points.

Parameters

- `pts1`: the vector or corresponding points from the first image
- `pts2`: the vector of corresponding points from the second image
- `cal1`: the intrinsic parameters of the first camera
- `cal2`: the intrinsic parameters of the second camera
- `inliers`: for each point pa:wir, the value is true if this pair is an inlier to the estimate
- `inlier_scale`: error distance tolerated for matches to be inliers

Public Static Functions

static std::string **`static_type_name`**()

Return the name of this algorithm.

class `kwiver::vital::algo::`**`estimate_fundamental_matrix`**

An abstract base class for estimating a fundamental matrix from matching 2D points.

Inherits from *kwiver::vital::algorithm_def< estimate_fundamental_matrix >*

Subclassed by kwiver::vital::algorithm_impl< estimate_fundamental_matrix, vital::algo::estimate_fundamental_matrix >

Public Functions

fundamental_matrix_sptr **estimate**(**const** kwiver::vital::feature_set_sptr *feat1*, **const** kwiver::vital::feature_set_sptr *feat2*, **const** kwiver::vital::match_set_sptr *matches*, std::vector<bool> &*inliers*, double *inlier_scale* = 1.0) **const**

Estimate an fundamental matrix from corresponding features.

Parameters

- `feat1`: the set of all features from the first image
- `feat2`: the set of all features from the second image
- `matches`: the set of correspondences between *feat1* and *feat2*
- `inliers`: for each point pair, the value is true if this pair is an inlier to the estimate
- `inlier_scale`: error distance tolerated for matches to be inliers

virtual kwiver::vital::fundamental_matrix_sptr **estimate**(**const** std::vector<kwiver::vital::vector_2d> &*pts1*, **const** std::vector<kwiver::vital::vector_2d> &*pts2*, std::vector<bool> &*inliers*, double *inlier_scale* = 1.0) **const** = 0

Estimate an fundamental matrix from corresponding points.

Parameters

- `pts1`: the vector or corresponding points from the first image
- `pts2`: the vector of corresponding points from the second image
- `inliers`: for each point pair, the value is true if this pair is an inlier to the estimate
- `inlier_scale`: error distance tolerated for matches to be inliers

Public Static Functions

static std::string **static_type_name**()

Return the name of this algorithm.

class `kwiver::vital::algo::estimate_homography`

An abstract base class for estimating a homography from matching 2D points.

Inherits from *kwiver::vital::algorithm_def< estimate_homography >*

Subclassed by kwiver::vital::algorithm_impl< estimate_homography, vital::algo::estimate_homography >

Public Functions

homography_sptr **estimate**(kwiver::vital::feature_set_sptr *feat1*, kwiver::vital::feature_set_sptr *feat2*, kwiver::vital::match_set_sptr *matches*, std::vector<bool> &*inliers*, double *inlier_scale* = 1.0) **const**

Estimate a homography matrix from corresponding features.

If estimation fails, a NULL-containing sptr is returned

Parameters

- `feat1`: the set of all features from the source image
- `feat2`: the set of all features from the destination image
- `matches`: the set of correspondences between *feat1* and *feat2*
- `inliers`: for each match in *matcher*, the value is true if this pair is an inlier to the homography estimate
- `inlier_scale`: error distance tolerated for matches to be inliers

virtual kwiver::vital::homography_sptr **estimate**(**const** std::vector<kwiver::vital::vector_2d> &*pts1*, **const** std::vector<kwiver::vital::vector_2d> &*pts2*, std::vector<bool> &*inliers*, double *inlier_scale* = 1.0) **const** = 0

Estimate a homography matrix from corresponding points.

If estimation fails, a NULL-containing sptr is returned

Parameters

- `pts1`: the vector or corresponding points from the source image
- `pts2`: the vector of corresponding points from the destination image
- `inliers`: for each point pair, the value is true if this pair is an inlier to the homography estimate
- `inlier_scale`: error distance tolerated for matches to be inliers

Public Static Functions

static std::string **static_type_name**()

Return the name of this algorithm.

class `kwiver::vital::algo::estimate_similarity_transform`

Algorithm for estimating the similarity transform between two point sets.

Inherits from *kwiver::vital::algorithm_def< estimate_similarity_transform >*

Subclassed by kwiver::vital::algorithm_impl< estimate_similarity_transform, vital::algo::estimate_similarity_transform >

Public Functions

virtual kwiver::vital::similarity_d **estimate_transform**(std::vector<kwiver::vital::vector_3d> **const** &*from*, std::vector<kwiver::vital::vector_3d> **const** &*to*) **const** = 0

Estimate the similarity transform between two corresponding point sets.

Return An estimated similarity transform mapping 3D points in the `from` space to points in the `to` space.

Parameters

- `from`: List of length N of 3D points in the from space.
- `to`: List of length N of 3D points in the to space.

Exceptions

- `algorithm_exception`: When the from and to point sets are misaligned, insufficient or degenerate.

similarity_d **estimate_transform**(std::vector<kwiver::vital::camera_sptr> **const** &*from*, std::vector<kwiver::vital::camera_sptr> **const** &*to*) **const**

Estimate the similarity transform between two corresponding sets of cameras.

Return An estimated similarity transform mapping camera centers in the `from` space to camera centers in the `to` space.

Parameters

- `from`: List of length N of cameras in the from space.
- `to`: List of length N of cameras in the to space.

Exceptions

- `algorithm_exception`: When the from and to point sets are misaligned, insufficient or degenerate.

similarity_d **estimate_transform**(std::vector<kwiver::vital::landmark_sptr> **const** &*from*, std::vector<kwiver::vital::landmark_sptr> **const** &*to*) **const**

Estimate the similarity transform between two corresponding sets of landmarks.

Return An estinated similarity transform mapping landmark locations in the `from` space to located in the `to` space.

Parameters

- `from`: List of length N of landmarks in the from space.
- `to`: List of length N of landmarks in the to space.

Exceptions

- `algorithm_exception`: When the from and to point sets are misaligned, insufficient or degenerate.

similarity_d **estimate_transform**(kwiver::vital::camera_map_sptr **const** *from*, kwiver::vital::camera_map_sptr **const** *to*) **const**

Estimate the similarity transform between two corresponding camera maps.

Cameras with corresponding frame IDs in the two maps are paired for transform estimation. Cameras with no corresponding frame ID in the other map are ignored. An algorithm_exception is thrown if there are no shared frame IDs between the two provided maps (nothing to pair).

Return An estimated similarity transform mapping camera centers in the `from` space to camera centers in the `to` space.

Exceptions

- `algorithm_exception`: When the from and to point sets are misaligned, insufficient or degenerate.

Parameters

- `from`: Map of original cameras, sharing N frames with the transformed cameras, where N > 0.
- `to`: Map of transformed cameras, sharing N frames with the original cameras, where N > 0.

similarity_d **`estimate_transform`**(kwiver::vital::landmark_map_sptr **const** *from*, kwiver::vital::landmark_map_sptr **const** *to*) **const**

Estimate the similarity transform between two corresponding landmark maps.

Landmarks with corresponding frame IDs in the two maps are paired for transform estimation. Landmarks with no corresponding frame ID in the other map are ignored. An algoirithm_exception is thrown if there are no shared frame IDs between the two provided maps (nothing to pair).

Return An estimated similarity transform mapping landmark centers in the `from` space to camera centers in the `to` space.

Exceptions

- `algorithm_exception`: When the from and to point sets are misaligned, insufficient or degenerate.

Parameters

- `from`: Map of original landmarks, sharing N frames with the transformed landmarks, where N > 0.
- `to`: Map of transformed landmarks, sharing N frames with the original landmarks, where N > 0.

Public Static Functions

static std::string **`static_type_name`**()

Name of this algo definition.

class `kwiver::vital::algo::`**`extract_descriptors`**

An abstract base class for extracting feature descriptors.

Inherits from *kwiver::vital::algorithm_def< extract_descriptors >*

Subclassed by kwiver::vital::algorithm_impl< extract_descriptors, vital::algo::extract_descriptors >, kwiver::arrows::ocv::extract_descriptors

Public Functions

virtual kwiver::vital::descriptor_set_sptr **extract**(kwiver::vital::image_container_sptr *image_data*, kwiver::vital::feature_set_sptr *features*, kwiver::vital::image_container_sptr *image_mask* = kwiver::vital::image_container_sptr ()) **const** = 0

Extract from the image a descriptor corresoponding to each feature.

Return a set of feature descriptors

Parameters

- `image_data`: contains the image data to process
- `features`: the feature locations at which descriptors are extracted
- `image_mask`: Mask image of the same dimensions as `image_data` where positive values indicate regions of `image_data` to consider.

Public Static Functions

static std::string **static_type_name**()

Return the name of this algorithm.

class `kwiver::vital::algo::`**feature_descriptor_io**

An abstract base class for reading and writing feature and desriptor sets.

This class represents an abstract interface for reading and writing feature and descriptor sets

Inherits from *kwiver::vital::algorithm_def< feature_descriptor_io >*

Subclassed by kwiver::vital::algorithm_impl< feature_descriptor_io, vital::algo::feature_descriptor_io >

Public Functions

void **load**(std::string **const** &*filename*, feature_set_sptr &*feat*, descriptor_set_sptr &*desc*) **const**

Load features and descriptors from a file.

Exceptions

- `kwiver::vital::path_not_exists`: Thrown when the given path does not exist.
- `kwiver::vital::path_not_a_file`: Thrown when the given path does not point to a file (i.e. it points to a directory).

Parameters

- `filename`: the path to the file the load
- `feat`: the set of features to load from the file
- `desc`: the set of descriptors to load from the file

void **save**(std::string **const** &*filename*, feature_set_sptr *feat*, descriptor_set_sptr *desc*) **const**

Save features and descriptors to a file.

Saves features and/or descriptors to a file. Either `feat` or `desc` may be Null, but not both. If both `feat` and `desc` are provided then the must be of the same size.

Exceptions

- `kwiver::vital::path_not_exists`: Thrown when the expected containing directory of the given path does not exist.
- `kwiver::vital::path_not_a_directory`: Thrown when the expected containing directory of the given path is not actually a directory.

Parameters

- `filename`: the path to the file to save
- `feat`: the set of features to write to the file
- `desc`: the set of descriptors to write to the file

Public Static Functions

static std::string **`static_type_name`**()
Return the name of this algorithm.

class `kwiver::vital::algo::filter_features`
Abstract base class for feature set filter algorithms.

Inherits from *kwiver::vital::algorithm_def< filter_features >*

Subclassed by kwiver::vital::algorithm_impl< filter_features_magnitude, vital::algo::filter_features >, kwiver::vital::algorithm_impl< filter_features_scale, vital::algo::filter_features >

Public Functions

feature_set_sptr **`filter`**(kwiver::vital::feature_set_sptr *input*) **const**
Filter a feature set and return a subset of the features.

The default implementation call the pure virtual function filter(feature_set_sptr feat, std::vector<unsigned int> &indices) const

Return a filtered version of the feature set (simple_feature_set)

Parameters

- `input`: The feature set to filter

std::pair<feature_set_sptr, descriptor_set_sptr> **`filter`**(kwiver::vital::feature_set_sptr *feat*, kwiver::vital::descriptor_set_sptr *descr*) **const**
Filter a feature_set and its coresponding *descriptor_set*.

The default implementation calls filter(feature_set_sptr feat, std::vector<unsigned int> &indices) const using with `feat` and then uses the resulting `indices` to construct a simple_descriptor_set with the corresponding descriptors.

Return a pair of the filtered features and descriptors

Parameters

- `feat`: The feature set to filter
- `descr`: The parallel descriptor set to filter

Public Static Functions

static std::string **static_type_name**()
Return the name of this algorithm.

class kwiver::vital::algo::**filter_tracks**
Abstract base class for track set filter algorithms.

Inherits from *kwiver::vital::algorithm_def< filter_tracks >*

Subclassed by kwiver::vital::algorithm_impl< filter_tracks, vital::algo::filter_tracks >

Public Functions

virtual kwiver::vital::track_set_sptr **filter**(kwiver::vital::track_set_sptr *input*) **const** = 0
Filter a track set and return a subset of the tracks.

Return a filtered version of the track set (simple_track_set)

Parameters

- input: The track set to filter

Public Static Functions

static std::string **static_type_name**()
Return the name of this algorithm.

class kwiver::vital::algo::**formulate_query**
An abstract base class for formulating descriptors for queries.

Inherits from *kwiver::vital::algorithm_def< formulate_query >*

Subclassed by kwiver::vital::algorithm_impl< formulate_query_core, vital::algo::formulate_query >

Public Functions

void **set_configuration**(kwiver::vital::config_block_sptr *config*)
Set this algorithm's properties via a config block.

bool **check_configuration**(kwiver::vital::config_block_sptr *config*) **const**
Check that the algorithm's currently configuration is valid.

Check that the algorithm's current configuration is valid.

virtual kwiver::vital::track_descriptor_set_sptr **formulate**(kwiver::vital::descriptor_request_sptr *request*) = 0
Formulate query.

Public Static Functions

static std::string **static_type_name**()
Return the name of this algorithm.

class kwiver::vital::algo::**image_filter**

Abstract base class for feature set filter algorithms.

Inherits from *kwiver::vital::algorithm_def< image_filter >*

Subclassed by kwiver::vital::algorithm_impl< matlab_image_filter, vital::algo::image_filter >

Public Functions

virtual kwiver::vital::image_container_sptr **filter**(kwiver::vital::image_container_sptr *image_data*) = 0

Filter a input image and return resulting image.

This method implements the filtering operation. The resulting image should be the same size as the input image.

Return a filtered version of the input image

Parameters

- image_data: Image to filter.

Public Static Functions

static std::string **static_type_name**()

Return the name of this algorithm.

class kwiver::vital::algo::**image_io**

An abstract base class for reading and writing images.

This class represents an abstract interface for reading and writing images.

Inherits from *kwiver::vital::algorithm_def< image_io >*

Subclassed by kwiver::vital::algorithm_impl< image_io, vital::algo::image_io >, kwiver::vital::algorithm_impl< image_io_dummy, kwiver::vital::algo::image_io >

Public Functions

image_container_sptr **load**(std::string **const** &*filename*) **const**

Load image from the file.

Return an image container refering to the loaded image

Exceptions

- kwiver::vital::path_not_exists: Thrown when the given path does not exist.
- kwiver::vital::path_not_a_file: Thrown when the given path does not point to a file (i.e. it points to a directory).

Parameters

- filename: the path to the file the load

void **save** (std::string **const** &*filename*, kwiver::vital::image_container_sptr *data*) **const**
Save image to a file.

Image file format is based on file extension.

Exceptions

- `kwiver::vital::path_not_exists`: Thrown when the expected containing directory of the given path does not exist.
- `kwiver::vital::path_not_a_directory`: Thrown when the expected containing directory of the given path is not actually a directory.

Parameters

- `filename`: the path to the file to save
- `data`: the image container refering to the image to write

Public Static Functions

static std::string **static_type_name** ()
Return the name of this algorithm.

class `kwiver::vital::algo::image_object_detector`
Image object detector base class/.

Inherits from *kwiver::vital::algorithm_def< image_object_detector >*

Subclassed by kwiver::vital::algorithm_impl< darknet_detector, vital::algo::image_object_detector >, kwiver::vital::algorithm_impl< hough_circle_detector, vital::algo::image_object_detector >, kwiver::vital::algorithm_impl< matlab_image_object_detector, vital::algo::image_object_detector >

Public Functions

virtual detected_object_set_sptr **detect** (image_container_sptr *image_data*) **const** = 0
Find all objects on the provided image.

This method analyzes the supplied image and along with any saved context, returns a vector of detected image objects.

Return vector of image objects found

Parameters

- `image_data`: the image pixels

Public Static Functions

static std::string **static_type_name** ()
Return the name of this algorithm.

class `kwiver::vital::algo::initialize_cameras_landmarks`
An abstract base class for initialization of cameras and landmarks.

Inherits from *kwiver::vital::algorithm_def< initialize_cameras_landmarks >*

Subclassed by kwiver::vital::algorithm_impl< initialize_cameras_landmarks, vital::algo::initialize_cameras_landmarks >

Public Types

typedef std::function<bool (kwiver::vital::camera_map_sptr, kwiver::vital::landmark_map_sptr) > **callback_t**
Typedef for the callback function signature.

Public Functions

virtual void **initialize** (kwiver::vital::camera_map_sptr *&cameras*, kwiver::vital::landmark_map_sptr *&landmarks*, kwiver::vital::feature_track_set_sptr *tracks*, kwiver::vital::video_metadata_map_sptr *metadata* = nullptr) **const** = 0
Initialize the camera and landmark parameters given a set of feature tracks.

The algorithm creates an initial estimate of any missing cameras and landmarks using the available cameras, landmarks, and feature tracks. It may optionally revise the estimates of exisiting cameras and landmarks.

Parameters

- `cameras`: the cameras to initialize
- `landmarks`: the landmarks to initialize
- `tracks`: the feature tracks to use as constraints
- `metadata`: the frame metadata to use as constraints

void **set_callback** (*callback_t cb*)
Set a callback function to report intermediate progress.

Public Static Functions

static std::string **static_type_name** ()
Return the name of this algorithm.

class `kwiver::vital::algo::match_features`
An abstract base class for matching feature points.

Inherits from *kwiver::vital::algorithm_def< match_features >*

Subclassed by kwiver::vital::algorithm_impl< match_features, vital::algo::match_features >, kwiver::vital::algorithm_impl< match_features_constrained, vital::algo::match_features >, kwiver::vital::algorithm_impl< match_features_fundamental_matrix, vital::algo::match_features >, kwiver::vital::algorithm_impl< match_features_homography, vital::algo::match_features >, kwiver::arrows::ocv::match_features

Public Functions

virtual kwiver::vital::match_set_sptr **match**(kwiver::vital::feature_set_sptr *feat1*, kwiver::vital::descriptor_set_sptr *desc1*, kwiver::vital::feature_set_sptr *feat2*, kwiver::vital::descriptor_set_sptr *desc2*) **const** = 0

Match one set of features and corresponding descriptors to another.

Return a set of matching indices from *feat1* to *feat2*

Parameters

- `feat1`: the first set of features to match
- `desc1`: the descriptors corresponding to *feat1*
- `feat2`: the second set fof features to match
- `desc2`: the descriptors corresponding to *feat2*

Public Static Functions

static std::string **static_type_name**()

Return the name of this algorithm.

class `kwiver::vital::algo::optimize_cameras`

Abstract algorithm definition base for optimizing cameras.

Inherits from *kwiver::vital::algorithm_def< optimize_cameras >*

Subclassed by kwiver::vital::algorithm_impl< optimize_cameras, vital::algo::optimize_cameras >

Public Functions

void **optimize**(kwiver::vital::camera_map_sptr &*cameras*, kwiver::vital::feature_track_set_sptr *tracks*, kwiver::vital::landmark_map_sptr *landmarks*, kwiver::vital::video_metadata_map_sptr *metadata* = nullptr) **const**

Optimize camera parameters given sets of landmarks and feature tracks.

We only optimize cameras that have associating tracks and landmarks in the given maps. The default implementation collects the corresponding features and landmarks for each camera and calls the single camera optimize function.

Exceptions

- `invalid_value`: When one or more of the given pointer is Null.

Parameters

- `cameras`: Cameras to optimize.
- `tracks`: The feature tracks to use as constraints.
- `landmarks`: The landmarks the cameras are viewing.
- `metadata`: The optional metadata to constrain the optimization.

virtual void **optimize**(kwiver::vital::camera_sptr *&camera*, **const** std::vector<kwiver::vital::feature_sptr> *&features*, **const** std::vector<kwiver::vital::landmark_sptr> *&landmarks*, kwiver::vital::video_metadata_vector *metadata* = kwiver::vital::video_metadata_vector()) **const** = 0

Optimize a single camera given corresponding features and landmarks.

This function assumes that 2D features viewed by this camera have already been put into correspondence with 3D landmarks by aligning them into two parallel vectors

Parameters

- `camera`: The camera to optimize.
- `features`: The vector of features observed by `camera` to use as constraints.
- `landmarks`: The vector of landmarks corresponding to `features`.
- `metadata`: The optional metadata to constrain the optimization.

Public Static Functions

static std::string **static_type_name**()

Return the name of this algorithm definition.

class `kwiver::vital::algo::`**refine_detections**

Image object detector base class/.

Inherits from *kwiver::vital::algorithm_def< refine_detections >*

Subclassed by kwiver::vital::algorithm_impl< refine_detections_write_to_disk, vital::algo::refine_detections >

Public Functions

virtual detected_object_set_sptr **refine**(image_container_sptr *image_data*, detected_object_set_sptr *detections*) **const** = 0

Refine all object detections on the provided image.

This method analyzes the supplied image and and detections on it, returning a refined set of detections.

Return vector of image objects refined

Parameters

- `image_data`: the image pixels
- `detections`: detected objects

Public Static Functions

static std::string **static_type_name**()

Return the name of this algorithm.

class `kwiver::vital::algo::`**split_image**

An abstract base class for converting base image type.

Inherits from *kwiver::vital::algorithm_def< split_image >*

Subclassed by kwiver::vital::algorithm_impl< split_image, vital::algo::split_image >

Public Functions

void **set_configuration**(kwiver::vital::config_block_sptr *config*)
Set this algorithm's properties via a config block.

bool **check_configuration**(kwiver::vital::config_block_sptr *config*) **const**
Check that the algorithm's currently configuration is valid.

Check that the algorithm's current configuration is valid.

virtual std::vector<kwiver::vital::image_container_sptr> **split**(kwiver::vital::image_container_sptr *img*) **const** = 0
Split image.

Public Static Functions

static std::string **static_type_name**()
Return the name of this algorithm.

class `kwiver::vital::algo::track_descriptor_set_input`
Read detected object sets.

This class is the abstract base class for the detected object set writer.

Detection sets from multiple images are stored in a single file with enough information to recreate a unique image identifier, usually the file name, and an associated set of track descriptors.

Inherits from *kwiver::vital::algorithm_def< track_descriptor_set_input >*

Public Functions

void **open**(std::string **const** &*filename*)
Open a file of track descriptor sets.

This method opens a track descriptor set file for reading.

Parameters

- `filename`: Name of file to open

Exceptions

- `kwiver::vital::path_not_exists`: Thrown when the given path does not exist.
- `kwiver::vital::path_not_a_file`: Thrown when the given path does not point to a file (i.e. it points to a directory).
- `kwiver::vital::file_not_found_exception`:

void **use_stream**(std::istream **strm*)
Read track descriptors from an existing stream.

This method specifies the input stream to use for reading track descriptors. Using a stream is handy when the track descriptors are available in a stream format.

Parameters

- `strm`: input stream to use

void **close**()
Close track descriptor set file.

The currently open track descriptor set file is closed. If there is no currently open file, then this method does nothing.

virtual bool **read_set**(kwiver::vital::track_descriptor_set_sptr &*set*, std::string &*image_name*) = 0
Read next detected object set.

This method reads the next set of detected objects from the file. **False** is returned when the end of file is reached.

Return **true** if track descriptors are returned, **false** if end of file.

Parameters

- `set`: Pointer to the new set of track descriptors. Set may be empty if there are no track descriptors on an image.
- `image_name`: Name of the image that goes with the track descriptors. This string may be empty depending on the source format.

bool **at_eof**() **const**
Determine if input file is at end of file.

This method reports the end of file status for a file open for reading.

Return **true** if file is at end.

Public Static Functions

static std::string **static_type_name**()
Return the name of this algorithm.

class `kwiver::vital::algo::track_descriptor_set_output`
Read and write detected object sets.

This class is the abstract base class for the detected object set reader and writer.

Detection sets from multiple images are stored in a single file with enough information to recreate a unique image identifier, usually the file name, and an associated wet of track descriptors.

Inherits from *kwiver::vital::algorithm_def< track_descriptor_set_output >*

Subclassed by kwiver::vital::algorithm_impl< track_descriptor_set_output_csv, vital::algo::track_descriptor_set_output >

Public Functions

void **open**(std::string **const** &*filename*)
Open a file of track descriptor sets.

This method opens a track descriptor set file for reading.

Parameters

- `filename`: Name of file to open

Exceptions

- `kwiver::vital::path_not_exists`: Thrown when the given path does not exist.
- `kwiver::vital::path_not_a_file`: Thrown when the given path does not point to a file (i.e. it points to a directory).

void **use_stream**(std::ostream *strm)
Write track descriptors to an existing stream.

This method specifies the output stream to use for reading track descriptors. Using a stream is handy when the track descriptors are available in a stream format.

Parameters

- `strm`: output stream to use

void **close**()
Close track descriptor set file.

The currently open track descriptor set file is closed. If there is no currently open file, then this method does nothing.

virtual void **write_set**(**const** kwiver::vital::track_descriptor_set_sptr *set*, std::string **const** &*image_path*) = 0
Write detected object set.

This method writes the specified detected object set and image name to the currently open file.

Parameters

- `set`: Detected object set
- `image_path`: File path to image associated with the track descriptors.

Public Static Functions

static std::string **static_type_name**()
Return the name of this algorithm.

class `kwiver::vital::algo::track_features`
An abstract base class for tracking feature points.

Inherits from *kwiver::vital::algorithm_def< track_features >*

Subclassed by kwiver::vital::algorithm_impl< track_features_core, vital::algo::track_features >

Public Functions

virtual feature_track_set_sptr **track**(feature_track_set_sptr *prev_tracks*, unsigned int *frame_number*, image_container_sptr *image_data*, image_container_sptr *mask* = image_container_sptr ()) **const** = 0
Extend a previous set of feature tracks using the current frame.

Return an updated set of feature tracks including the current frame

Exceptions

- `image_size_mismatch_exception`: When the given non-zero mask image does not match the size of the dimensions of the given image data.

Parameters

- `prev_tracks`: the feature tracks from previous tracking steps
- `frame_number`: the frame number of the current frame
- `image_data`: the image pixels for the current frame
- `mask`: Optional mask image that uses positive values to denote regions of the input image to consider for feature tracking. An empty sptr indicates no mask (default value).

Public Static Functions

static std::string **`static_type_name`**()
Return the name of this algorithm.

class `kwiver::vital::algo::train_detector`
An abstract base class for training object detectors.

Inherits from *kwiver::vital::algorithm_def< train_detector >*

Subclassed by kwiver::vital::algorithm_impl< darknet_trainer, vital::algo::train_detector >

Public Functions

virtual void **`train_from_disk`**(std::vector<std::string> *train_image_names*, std::vector<kwiver::vital::detected_object_set_sptr> *train_groundtruth*, std::vector<std::string> *test_image_names*, std::vector<kwiver::vital::detected_object_set_sptr> *test_groundtruth*) = 0
Train a detection model given a list of images and detections.

This varient is geared towards offline training.

Parameters

- `train_image_list`: list of train image filenames
- `train_groundtruth`: annotations loaded for each image
- `test_image_list`: list of test image filenames
- `test_groundtruth`: annotations loaded for each image

void **`train_from_memory`**(std::vector<kwiver::vital::image_container_sptr> *images*, std::vector<kwiver::vital::detected_object_set_sptr> *groundtruth*)
Train a detection model given images and detections.

This varient is geared towards online training, and is not required to be defined.

Exceptions

- `runtime_exception`: if not defined.

Parameters

- `images`: vector of input images

- `groundtruth`: annotations loaded for each image

Public Static Functions

static std::string **static_type_name**()
Return the name of this algorithm.

class `kwiver::vital::algo::`**triangulate_landmarks**
An abstract base class for triangulating landmarks.

Inherits from *kwiver::vital::algorithm_def< triangulate_landmarks >*

Subclassed by kwiver::vital::algorithm_impl< triangulate_landmarks, vital::algo::triangulate_landmarks >

Public Functions

virtual void **triangulate**(kwiver::vital::camera_map_sptr *cameras*,
kwiver::vital::feature_track_set_sptr *tracks*,
kwiver::vital::landmark_map_sptr &*landmarks*) **const** = 0
Triangulate the landmark locations given sets of cameras and feature tracks.

This function only triangulates the landmarks with indices in the landmark map and which have support in the feature tracks and cameras

Parameters

- `cameras`: the cameras viewing the landmarks
- `tracks`: the feature tracks to use as constraints
- `landmarks`: the landmarks to triangulate

Public Static Functions

static std::string **static_type_name**()
Return the name of this algorithm.

class `kwiver::vital::algo::`**uuid_factory**
Abstract base class for creating uuid's.

Inherits from *kwiver::vital::algorithm_def< uuid_factory >*

Subclassed by kwiver::vital::algorithm_impl< uuid_factory_uuid, vital::algo::uuid_factory >

Public Static Functions

static std::string **static_type_name**()
Return the name of this algorithm.

class `kwiver::vital::algo::`**video_input**
An abstract base class for reading videos.

This class represents an abstract interface for reading videos. Once the video is opened, the frames are returned in order.

Use cases:

1) Reading video from a directory of images.

2) Reading video frames from a list of file names.

3) Reading video from mpeg/video file (one of many formats) (e.g. FMV)

4) Reading video from mpeg/video file (one of many formats) with cropping (e.g. WAMI). This includes Providing geostationary images by cropping to a specific region from an image. This may result in no data if the geo region and image do not intersect.

5) Reading video from network stream. (RTSP) This may result in unexpected end of video conditions and network related disruptions (e.g. missing frames, connection terminating, ...)

A note about the basic capabilities:

HAS_EOV - This capability is set to true if the video source can determine end of video. This is usually the case if the video is being read from a file, but may not be known if the video is coming from a streaming source.

HAS_FRAME_NUMBERS - This capability is set to true if the video source supplies frame numbers. If the video source specifies a frame number, then that number is used when forming a time stamp. If the video does not supply a frame number, the time stamp will not have a frame number.

HAS_FRAME_TIME - This capability is set to true if the video source supplies a frame time. If a frame time is supplied, it is made available in the time stamp for that frame. If the frame time is not supplied, then the timestamp will hot have the time set.

HAS_FRAME_DATA - This capability is set to true if the video source supplies frame images. It may seem strange for a video input algorithm to not supply image data, but happens with a reader that only supplies the metadata.

HAS_ABSOLUTE_FRAME_TIME - This capability is set to true if the video source supplies an absolute, rather than relative frame time. This capability is not set if an absolute frame time can not be found, or if the absolute frame time is configured as "none".

HAS_METADATA - This capability is set if the video source supplies some type of metadata. The metadata could be in 0601 or 0104 data formats or a different source.

HAS_TIMEOUT - This capability is set if the implementation supports the timeout parameter on the *next_frame()* method.

All implementations **must** support the basic traits, in that they are registered with a **true** or **false** value. Additional implementation specific (extended) traits may be added. The application should first check to see if a extended trait is registered by calling has_trait() since the actual implementation is set by a configuration entry and not directly known by the application.

Extended capabilities can be created to publish capabilities of non-standard video sources. These capabilities should be namespaced using the name (or abbreviation) of the concrete algorithm followed by the abbreviation of the capability.

Inherits from *kwiver::vital::algorithm_def< video_input >*

Subclassed by kwiver::vital::algorithm_impl< video_input_filter, vital::algo::video_input >, kwiver::vital::algorithm_impl< video_input_image_list, vital::algo::video_input >, kwiver::vital::algorithm_impl< video_input_pos, vital::algo::video_input >, kwiver::vital::algorithm_impl< video_input_split, vital::algo::video_input >, kwiver::vital::algorithm_impl< vidl_ffmpeg_video_input, vital::algo::video_input >

Public Functions

virtual void **open**(std::string *video_name*) = 0
Open a video stream.

This method opens the specified video stream for reading. The format of the name depends on the concrete implementation. It could be a file name or it could be a URI.

Capabilities are set in this call, so they are available after.

Note Once a video is opened, it starts in an invalid state (i.e. before the first frame of video). You must call *next_frame()* to step to the first frame of video before calling *frame_image()*.

Parameters

- `video_name`: Identifier of the video stream.

Exceptions

- `exception`: if open failed

virtual void **close**() = 0

Close video stream.

Close the currently opened stream and release resources. Closing a stream that is already closed does not cause a problem.

virtual bool **end_of_video**() **const** = 0

Return end of video status.

This method returns the end-of-video status of the input video. **true** is returned if the last frame has been returned.

This method will always return **false** for video streams that have no ability to detect end of video, such as network streams.

Return **true** if at end of video, **false** otherwise.

virtual bool **good**() **const** = 0

Check whether state of video stream is good.

This method checks the current state of the video stream to see if it is good. A stream is good if it refers to a valid frame such that calls to *frame_image()* and *frame_metadata()* are expected to return meaningful data. After calling *open()* the initial video state is not good until the first call to *next_frame()*.

Return **true** if video stream is good, **false** if not good.

virtual bool **next_frame**(kwiver::vital::*timestamp* &*ts*, uint32_t *timeout* = 0) = 0

Advance to next frame in video stream.

This method advances the video stream to the next frame, making the image and metadata available. The returned timestamp is for new current frame.

The timestamp returned may be missing either frame number or time or both, depending on the actual implementation.

Calling this method will make a new image and metadata packets available. They can be retrieved by calling *frame_image()* and *frame_metadata()*.

Check the HAS_TIMEOUT capability from the concrete implementation to see if the timeout feature is supported.

If the video input is already an end, then calling this method will return **false**.

Return **true** if frame returned, **false** if end of video.

Parameters

- `ts`: Time stamp of new frame.
- `timeout`: Number of seconds to wait. 0 = no timeout.

Exceptions

- `video_input_timeout_exception`: when the timeout expires.
- `video_stream_exception`: when there is an error in the video stream.

virtual kwiver::vital::image_container_sptr **`frame_image`**`()` = 0

Get current frame from video stream.

This method returns the image from the current frame. If the video input is already an end, then calling this method will return a null pointer.

This method is idempotent. Calling it multiple times without calling *next_frame()* will return the same image.

Return Pointer to image container.

Exceptions

- `video_stream_exception`: when there is an error in the video stream.

virtual kwiver::vital::video_metadata_vector **`frame_metadata`**`()` = 0

Get metadata collection for current frame.

This method returns the metadata collection for the current frame. It is best to call this after calling *next_frame()* to make sure the metadata and video are synchronized and that no metadata collections are lost.

Metadata typically occurs less frequently than video frames, so if you call *next_frame()* and *frame_metadata()* together while processing a video, there may be times where no metadata is returned. In this case an empty metadata vector will be returned.

Also note that the metadata collection contains a timestamp that can be used to determine where the metadata fits in the video stream.

In video streams without metadata (as determined by the stream capability), this method may return and empty vector, indicating no new metadata has been found.

Calling this method at end of video will return an empty metadata vector.

This method is idempotent. Calling it multiple times without calling *next_frame()* will return the same metadata.

Return Vector of metadata pointers.

Exceptions

- `video_stream_exception`: when there is an error in the video stream.

algorithm_capabilities **const** &**`get_implementation_capabilities`**`()` **const**

Return capabilities of concrete implementation.

This method returns the capabilities for the currently opened video.

Return Reference to supported video capabilities.

Public Static Functions

static std::string **static_type_name**()
Return the name of this algorithm.

Arrow Architecture

Arrows is the collection of plugins that provides implementations of the algorithms declared in Vital. Each arrow can be enabled or disabled in build process through CMake options. Most arrows bring in additional third-party dependencies and wrap the capabilities of those libraries to make them accessible through the Vital APIs. The code in Arrows also converts or wrap data types from these external libraries into Vital data types. This allows interchange of data between algorithms from different arrows using Vital types as the intermediary.

Capabilities are currently organized into Arrows based on what third party library they require. However, this arrangement is not required and may change as the number of algorithms and arrows grows. Some arrows, like core , require no additional dependencies. The provided Arrows are:

Core

Burnout

Ceres

Bundle Adjust Algorithm

class kwiver::arrows::ceres::**bundle_adjust**
A class for bundle adjustment of feature tracks using Ceres.

Inherits from kwiver::vital::algorithm_impl< bundle_adjust, vital::algo::bundle_adjust >

Public Functions

bundle_adjust()
Constructor.

~bundle_adjust()
Destructor.

config_block_sptr **get_configuration**() **const**
Get this algorithm's configuration block .

void **set_configuration**(vital::config_block_sptr *config*)
Set this algorithm's properties via a config block.

bool **check_configuration**(vital::config_block_sptr *config*) **const**
Check that the algorithm's currently configuration is valid.

void **optimize**(vital::camera_map_sptr &*cameras*, vital::landmark_map_sptr &*landmarks*, vital::feature_track_set_sptr *tracks*, vital::video_metadata_map_sptr *metadata* = nullptr) **const**
Optimize the camera and landmark parameters given a set of feature tracks.

Optimize the camera and landmark parameters given a set of tracks.

Parameters

- `cameras`: the cameras to optimize
- `landmarks`: the landmarks to optimize
- `tracks`: the feature tracks to use as constraints
- `metadata`: the frame metadata to use as constraints

void **`set_callback`**(callback_t *cb*)
Set a callback function to report intermediate progress.

bool **`trigger_callback`**()
This function is called by a Ceres callback to trigger a kwiver callback.

class `priv`
Private implementation class.

Inherits from kwiver::arrows::ceres::solver_options, kwiver::arrows::ceres::camera_options

Public Functions

`priv`()
Constructor.

Public Members

bool **`verbose`**
verbose output

LossFunctionType **`loss_function_type`**
the robust loss function type to use

double **`loss_function_scale`**
the scale of the loss function

camera_map::map_camera_t **`cams`**
the input cameras to update in place

landmark_map::map_landmark_t **`lms`**
the input landmarks to update in place

std::map<track_id_t, std::vector<double>> **`landmark_params`**
a map from track id to landmark parameters

std::map<frame_id_t, std::vector<double>> **`camera_params`**
a map from frame number to extrinsic parameters

std::vector<std::vector<double>> **`camera_intr_params`**
vector of unique camera intrinsic parameters

std::map<frame_id_t, unsigned int> **`frame_to_intr_map`**
a map from frame number to index of unique camera intrinsics in camera_intr_params

StateCallback **`ceres_callback`**
the ceres callback class

vital::logger_handle_t **`m_logger`**
Logger handle.

Optimize Cameras Algorithm

class `kwiver::arrows::ceres::optimize_cameras`
A class for optimization of camera paramters using Ceres.

Inherits from kwiver::vital::algorithm_impl< optimize_cameras, vital::algo::optimize_cameras >

Public Functions

optimize_cameras()
Constructor.

~optimize_cameras()
Destructor.

optimize_cameras(**const** *optimize_cameras* &*other*)
Copy Constructor.

config_block_sptr **get_configuration**() **const**
Get this algorithm's configuration block .

void **set_configuration**(vital::config_block_sptr *config*)
Set this algorithm's properties via a config block.

bool **check_configuration**(vital::config_block_sptr *config*) **const**
Check that the algorithm's currently configuration is valid.

void **optimize**(kwiver::vital::camera_map_sptr &*cameras*, kwiver::vital::feature_track_set_sptr *tracks*, kwiver::vital::landmark_map_sptr *landmarks*, kwiver::vital::video_metadata_map_sptr *metadata* = nullptr) **const**
Optimize camera parameters given sets of landmarks and feature tracks.

We only optimize cameras that have associating tracks and landmarks in the given maps. The default implementation collects the corresponding features and landmarks for each camera and calls the single camera optimize function.

Exceptions

- `invalid_value`: When one or more of the given pointer is Null.

Parameters

- `cameras`: Cameras to optimize.
- `tracks`: The feature tracks to use as constraints.
- `landmarks`: The landmarks the cameras are viewing.
- `metadata`: The optional metadata to constrain the optimization.

void **optimize**(vital::camera_sptr &*camera*, **const** std::vector<vital::feature_sptr> &*features*, **const** std::vector<vital::landmark_sptr> &*landmarks*, kwiver::vital::video_metadata_vector *metadata* = kwiver::vital::video_metadata_vector()) **const**
Optimize a single camera given corresponding features and landmarks.

This function assumes that 2D features viewed by this camera have already been put into correspondence with 3D landmarks by aligning them into two parallel vectors

Parameters

- `camera`: The camera to optimize.
- `features`: The vector of features observed by `camera` to use as constraints.
- `landmarks`: The vector of landmarks corresponding to `features`.
- `metadata`: The optional metadata to constrain the optimization.

class priv

Private implementation class.

Inherits from kwiver::arrows::ceres::solver_options, kwiver::arrows::ceres::camera_options

Public Functions

priv()

Constructor.

Public Members

bool **verbose**

verbose output

LossFunctionType **loss_function_type**

the robust loss function type to use

double **loss_function_scale**

the scale of the loss function

vital::logger_handle_t **m_logger**

Logger handle.

Camera Position Smoothness Class

class `kwiver::arrows::ceres::`**camera_position_smoothness**

Ceres camera smoothness functor.

Public Functions

camera_position_smoothness(**const** double *smoothness*)

Constructor.

template <typename T>
bool **operator()**(**const** T ***const** *prev_pose*, **const** T ***const** *curr_pose*, **const** T ***const** *next_pose*, T **residuals*) **const**

Position smoothness error functor for use in Ceres.

Parameters

- `prev_pos`: Camera pose data block at previous time
- `curr_pos`: Camera pose data block at current time
- `next_pos`: Camera pose data block at next time

- `residuals`: Camera pose blocks contain 6 parameters: 3 for rotation(angle axis), 3 for center Only the camera centers are used in this function to penalize the difference between current position and the average between previous and next positions.

Public Static Functions

ceres::CostFunction ***create**(**const** double *s*)
Cost function factory.

Camera Limit Forward Motion Class

class `kwiver::arrows::ceres::`**camera_limit_forward_motion**
Ceres camera limit forward motion functor.

This class is to reglarize camera motion to minimize the amount of motion in the camera looking direction. This is useful with zoom lenses at long focal lengths where distance and zoom are ambiguous. Adding this constraint allows the optimization to prefer fast zoom changes over fast position change.

Public Functions

camera_limit_forward_motion(**const** double *scale*)
Constructor.

template <typename T**>**
bool **operator()**(**const** T ***const** *pose1*, **const** T ***const** *pose2*, T **residuals*) **const**
Camera forward motion error functor for use in Ceres.

Parameters

- `pose1`: Camera pose data block at time 1
- `pose2`: Camera pose data block at time 2
- `residuals`: Camera pose blocks contain 6 parameters: 3 for rotation(angle axis), 3 for center

Public Members

double **scale_**
the magnitude of this constraint

Public Static Functions

ceres::CostFunction ***create**(**const** double *s*)
Cost function factory.

Distortion Poly Radial Class

class `kwiver::arrows::ceres::`**distortion_poly_radial**
Class to hold to distortion function and traits.

Public Static Functions

template <typename T>
static void **apply**(**const** T **dist_coeffs*, **const** T **source_xy*, T **distorted_xy*)
Function to apply polynomial radial distortion.

Parameters

- `dist_coeffs`: radial distortion coefficients (2)
- `source_xy`: 2D point in normalized image coordinates
- `distorted_xy`: 2D point in distorted normalized image coordinates

Distortion Poly Radial Tangential Class

class `kwiver::arrows::ceres::`**`distortion_poly_radial_tangential`**
Class to hold to distortion function and traits.

Public Static Functions

template <typename T>
static void **apply**(**const** T **dist_coeffs*, **const** T **source_xy*, T **distorted_xy*)
Function to apply polynomial radial and tangential distortion.

Parameters

- `dist_coeffs`: radial (3) and tangential (2) distortion coefficients
- `source_xy`: 2D point in normalized image coordinates
- `distorted_xy`: 2D point in distorted normalized image coordinates

Distortion Ratpoly Radial Tangential Class

class `kwiver::arrows::ceres::`**`distortion_ratpoly_radial_tangential`**
Class to hold to distortion function and traits.

Public Static Functions

template <typename T>
static void **apply**(**const** T **dist_coeffs*, **const** T **source_xy*, T **distorted_xy*)
Function to apply rational polynomial radial and tangential distortion.

Parameters

- `dist_coeffs`: radial (6) and tangential (2) distortion coefficients
- `source_xy`: 2D point in normalized image coordinates
- `distorted_xy`: 2D point in distorted normalized image coordinates

Create Cost Func Factory

KWIVER_ALGO_CERES_EXPORT::ceres::CostFunction *`kwiver::arrows::ceres::`**`create_cost_func`**(LensDistortion *ldt*, double *x*, double *y*)

Factory to create Ceres cost functions for each lens distortion type.

Darknet

Darknet is an open source neural network framework written in C and CUDA.

The following algorithm implementations use Darknet

darknet_detector	**class** `kwiver::arrows::darknet::`**`darknet_detector`** Inherits from kwiver::vital::algorithm_impl< darknet_detector, vital::algo::image_object_detector >
darknet_trainer	**class** `kwiver::arrows::darknet::`**`darknet_trainer`** Darknet Training Utility Class. Inherits from kwiver::vital::algorithm_impl< darknet_trainer, vital::algo::train_detector >

In the pipe files, you can tune the algorithm with these variables :

- darknet:thresh
- darknet:hier_thresh
- darknet:gpu_index

FAQ

I am running out of memory in CUDA... Try one or both of these suggestions: - Change the darknet/models/virat.cfg variables height,weight to smaller powers of 32 - Change the darknet/models/virat.cfg variables batch and subdivisions (make sure they are still the same)

Matlab

OpenCV

This arrow is a collection of vital algorithms implemented with the OpenCV API

This arrow can be built by enabling the KWIVER_ENABLE_OPENCV CMake flag

This arrow implements the following algorithms:

image_io	*split_image*

Algorithm Configuration

Each algoithm implementation has the option to expose configuration parameters. These parameters help define the execution of the algoithm and are specific to this implementation.

Image I/O

No configuration options provied

Split Image

No configuration options provided

Proj4

UUID

VisCL

VXL

This arrow is a collection of vital algorithms implemented with the VXL API

This arrow can be built by enabling the KWIVER_ENABLE_VXL CMake flag

This arrow implements the following algorithms:

image_io	*split_image*

Algorithm Configuration

Each algoithm implementation has the option to expose configuration parameters. These parameters help define the execution of the algoithm and are specific to this implementation.

Image I/O

No configuration options provied

Split Image

No configuration options provided

Sprokit Architecture

Sprokit is a "**S**tream **Pro**cessing Tool**kit**" that provides infrastructure for chaining together algorithms into pipelines for processing streaming data sources. The most common use case of Sprokit is for video processing, but Sprokit is data type agnostic and could be used for any type of streaming data. Sprokit allows the user to dynamically connect

and configure a pipeline by chaining together processing nodes called "processes" into a directed graph with data sources and sinks. Sprokit schedules the jobs to run each process and keep data flowing through pipeline. Sprokit also allows processes written in Python to be interconnected with those written in C++.

Getting Started with sprokit

The central component of KWIVER is vital which supplies basic data types and fundimental alrogithms. In addition, we use sprokit's pipelining facilities to manage, integrate and run many of KWIVER's modules and capabilities. To see what modules (called processes in sprockit) are available, run the following command:

```
$ plugin_explorer --process -b
```

Here's a typical list of modules (note that as KWIVER expands, this list is likely to grow):

—- All process Factories

Factories that create type "sprokit::process"

Process type: frame_list_input Reads a list of image file names and generates stream of images and associated time stamps

Process type: stabilize_image Generate current-to-reference image homographies

Process type: detect_features Detect features in an image that will be used for stabilization

Process type: extract_descriptors Extract descriptors from detected features

Process type: feature_matcher Match extracted descriptors and detected features

Process type: compute_homography Compute a frame to frame homography based on tracks

Process type: compute_stereo_depth_map Compute a stereo depth map given two frames

Process type: draw_tracks Draw feature tracks on image

Process type: read_d_vector Read vector of doubles

Process type: refine_detections Refines detections for a given frame

Process type: image_object_detector Apply selected image object detector algorithm to incoming images.

Process type: image_filter Apply selected image filter algorithm to incoming images.

Process type: image_writer Write image to disk.

Process type: image_file_reader Reads an image file given the file name.

Process type: detected_object_input Reads detected object sets from an input file. Detections read from the input file are grouped into sets for each image and individually returned.

Process type: detected_object_output Writes detected object sets to an output file. All detections are written to the same file.

Process type: detected_object_filter Filters sets of detected objects using the detected_object_filter algorithm.

Process type: video_input Reads video files and produces sequential images with metadata per frame.

Process type: draw_detected_object_set Draws border around detected objects in the set using the selected algorithm.

Process type: track_descriptor_input Reads track descriptor sets from an input file.

Process type: track_descriptor_output Writes track descriptor sets to an output file. All descriptors are written to the same file.

Process type: image_viewer Display input image and delay

Process type: draw_detected_object_boxes Draw detected object boxes on images.

Process type: collate Collates data from multiple worker processes

Process type: distribute Distributes data to multiple worker processes

Process type: pass Pass a data stream through

Process type: sink Ignores incoming data

Process type: any_source A process which creates arbitrary data

Process type: const A process wth a const flag

Process type: const_number Outputs a constant number

Process type: data_dependent A process with a data dependent type

Process type: duplicate A process which duplicates input

Process type: expect A process which expects some conditions

Process type: feedback A process which feeds data into itself

Process type: flow_dependent A process with a flow dependent type

Process type: multiplication Multiplies numbers

Process type: multiplier_cluster A constant factor multiplier cluster

Process type: mutate A process with a mutable flag

Process type: numbers Outputs numbers within a range

Process type: orphan_cluster A dummy cluster

Process type: orphan A dummy process

Process type: print_number Print numbers to a file

Process type: shared A process with the shared flag

Process type: skip A process which skips input data

Process type: tagged_flow_dependent A process with a tagged flow dependent types

Process type: take_number Print numbers to a file

Process type: take_string Print strings to a file

Process type: tunable A process with a tunable parameter

Process type: input_adapter Source process for pipeline. Pushes data items into pipeline ports. Ports are dynamically created as needed based on connections specified in the pipeline file.

Process type: output_adapter Sink process for pipeline. Accepts data items from pipeline ports. Ports are dynamically created as needed based on connections specified in the pipeline file.

Process type: template Description of process. Make as long as necessary to fully explain what the process does and how to use it. Explain specific algorithms used, etc.

Process type: kw_archive_writer Writes kw archives

Process type: test_python_process A test Python process

Process type: pyprint_number A Python process which prints numbers

This is the list of modules that can be included in a Sprokit pipeline. We're going to use the `numbers` module and the the `print_number` module to create a very simple pipeline. To learn more about the `numbers` module we'll again use `plugin_explorer` this time to get details on a particular module. For `numbers` we'll use the following command:

```
$ plugin_explorer --process --type numbers -d --config

Factories that create type "sprokit::process"

  Process type: numbers
  Description:        Outputs numbers within a range

    Properties: _no_reentrant,
    -- Configuration --
    Name       : end
    Default    : 100
    Description: The value to stop counting at.
    Tunable    : no

    Name       : start
    Default    : 0
    Description: The value to start counting at.
    Tunable    : no

  Input ports:
  Output ports:
    Name       : number
    Type       : integer
    Flags      : _required,
    Description: Where the numbers will be available.
```

And for `print_number`, we'll use:

```
$ plugin_explorer --process --type print_number -d --config

Factories that create type "sprokit::process"

  Process type: print_number
  Description:        Print numbers to a file

    Properties: _no_reentrant,
    -- Configuration --
    Name       : output
    Default    :
    Description: The path of the file to output to.
    Tunable    : no

  Input ports:
    Name       : number
    Type       : integer
    Flags      : _required,
    Description: Where numbers are read from.

  Output ports:
```

The output of these commands tells us enough about each process to construct a Sprockit ".pipe" file that defines a processing pipeline. In particular we'll need to know how to configure each process (the "Configuration") and how they can be hooked together (the input and output "Ports").

KWIVER comes with a sample sprokit/pipelines/number_flow.pipe file that configures and connects the pipeline so that the `numbers` process will generate a set of integers from 1 to 99 and the `print_number` process will write those to a file called `numbers.txt`. Of particular interest is the section at the end of the file that actually "hooks up" the pipeline.

To run the pipeline, we'll use the Sprokit `pipeline_runner` command:

```
$ pipeline_runner -p </path/to/kwiver/source>/sprokit/pipelines/number_flow.pipe
```

After the pipeline completes, you should find a file, `numbers.txt`, in your working directory.

Python Processes

One of KWIVER's great strengths (as provided by sprokit) is the ability to create hybrid pipelines which combine C++ and Python processes in the same pipeline. This greatly facilitates prototyping complex processing pipelines. To test this out we'll still use the `numbers` process, but we'll use a Python version of the `print_number` process called `kw_print_number_process` the code for which can be seen in sprokit/processes/python/kw_print_number_process.py. As usual, we can lean about this process with the following command:

```
$ plugin_explorer --process --type kw_print_number_process -d --config

Process type: kw_print_number_process
  Description: A Simple Kwiver Test Process
  Properties: _no_reentrant, _python
Configuration:
  Name        : output
  Default     : .
  Description: The path for the output file.
  Tunable     : no

Input ports:
  Name        : input
  Type        : integer
  Flags       : _required
  Description: Where numbers are read from.

Output ports:
```

As you can see, the process is very similar to the C++ `print_number` process. As a result, the [".pipe" file is very similar](sprokit/pipelines/number_flow_python.pipe).

In order to get around limitations imposed by the Python Global Interpreter Lock, we'll use a different Sprokit scheduler for this pipeline. The `pythread_per_process` scheduler which does essentially what it says: it creates a Python thread for every process in the pipeline:

```
pipeline_runner -S pythread_per_process -p </path/to/kwiver/source>/sprokit/pipelines/
↪number_flow_python.pipe>
```

As with the previous pipeline, the numbers will be written to an output file, this time `numbers_from_python.txt`

Process

detected_object_output

This sprokit process is used to ...

draw_detected_object_boxes

Pipefile Usage

The following sections describe the blocks needed to use this process in a pipe file

Pipefile block

```
# ================================================================
process draw # This name can be whatever you want
 :: draw_detected_object_boxes
 :default_line_thickness 3
# ================================================================
```

Pipefile connections

The following Input ports will need to be set

```
# ProcessX will provide a detected_object_set
connect from processX.detected_object_set
       to   draw.detected_object_set
# ProcessY will provide an image
connect from processY.image
       to draw.image
```

The follwing Output ports are available from this process

```
# This process will provide an image with boxes to any processes
connect from draw.image
       to   processZ.image
```

Class Description

class `kwiver::`**`draw_detected_object_boxes_process`**

Process to draw detected object boxes on an image.

Draws boxes around detected objects.

{detected_object_set} List of detections to draw.

{image} Input image where boxes are drawn.

{image} Updated image with boxes and other annotations.

{threshold} Detections with concidence values below this value are not drawn. (float)

{alpha_blend_prob} If this item is set to **true**, then detections with a lower probability are drawn with more transparency.

{default_color} The default color specification for drawing boxes if no other more specific color spec is provided.

{custom_class_color}

Inherits from sprokit::process

class priv

Public Functions

void **draw_box**(cv::Mat *&image*, **const** vital::detected_object_sptr *dos*, std::string *label*, double *prob*, bool *just_text* = false, int *offset_index* = 0) **const**

Draw a box on an image.

This method draws a box on an image for the bounding box from a detected object.

When drawing a box with multiple class names, draw the first class_name with the `just_text` parameter **false** and all subsequent calls with it set to **true**. Also the `offset` parameter must be incremented so the labels do not overwrite.

Parameters

- `image`: Input image updated with drawn box
- `dos`: detected object with bounding box
- `label`: Text label to use for box
- `prob`: Probability value to add to label text
- `just_text`: Set to true if only draw text, not the bounding box. This is used when there are multiple labels for the same detection.
- `offset`: How much to offset text fill box from text baseline. This is used to offset labels when there are more than one label for a detection.

vital::image_container_sptr **draw_detections**(vital::image_container_sptr *image_data*, vital::detected_object_set_sptr *in_set*) **const**

Draw detected object on image.

This method draws the detections on a copy of the supplied image. The detections are drawn in confidence order up to the threshold. For each detection, the most likely class_name is optionally displayed below the box.

Return Updated image.

Parameters

- `image_data`: The image to draw on.
- `input_set`: List of detections to draw.

bool **name_selected**(std::string **const** *&name*) **const**

See if name has been selected for display.

Return **true** if name should be rendered

Parameters

- `name`: Name to check.

frame_list_input

You can find the available option for the image_reader *here*

Reads a list of image file names and generates stream of images and associated
time stamps

Configuration

Variable	Default	Tunable	Description
frame_time	0.03333333	NO	Inter frame time in seconds. The generated timestamps will have the specified number of seconds in the generated timestamps for sequential frames. This can be used to simulate a frame rate in a video stream application.
image_list_file	(no default value)	NO	Name of file that contains list of image file names. Each line in the file specifies the name of a single image file.
image_reader	(no default value)	NO	Algorithm configuration subblock
path	(no default value)	NO	Path to search for image file. The format is the same as the standard path specification, a set of directories separated by a colon (':')

Input Ports

There are no input ports for this process.

Output Ports

Port name	Data Type	Flags	Description
image	kwiver:image	(none)	Single frame image.
image_file_name	kwiver:image_file_name	(none)	Name of an image file. The file name may contain leading path components.
timestamp	kwiver:timestamp	(none)	Timestamp for input image.

Pipefile Usage

The following sections describe the blocks needed ot use this process in a pipe file.

Pipefile block

```
# ================================================================
process <this-name>
  :: frame_list_input
# Inter frame time in seconds. The generated timestamps will have the specified
# number of seconds in the generated timestamps for sequential frames. This can
# be used to simulate a frame rate in a video stream application.
  frame_time = 0.03333333
# Name of file that contains list of image file names. Each line in the file
# specifies the name of a single image file.
  image_list_file = <value>
# Algorithm configuration subblock
  image_reader = <value>
# Path to search for image file. The format is the same as the standard path
# specification, a set of directories separated by a colon (':')
```

```
  path = <value>
# ============================================================
```

Process connections

The following Input ports will need to be set

```
# There are no input port's for this process
```

The following Output ports will need to be set

```
connect from <this-proc>.image
         to   <downstream-proc>.image
connect from <this-proc>.image_file_name
         to   <downstream-proc>.image_file_name
connect from <this-proc>.timestamp
         to   <downstream-proc>.timestamp
```

Class Description

class `kwiver::`**`frame_list_process`**
Reads a series of images.

{image}

{frame} {time}

Inherits from sprokit::process

image_object_detector

This sprokit process is used to ...

image_viewer

Display input image and delay

class `kwiver::`**`image_viewer_process`**
Display images.

Inherits from sprokit::process

Configuration

annotate_image = false *Not tunable* Add frame number and other text to display.

footer = (no default value) *Not tunable* Footer text for image display. Displayed centered at bottom of image.

header = (no default value) *Not tunable* Header text for image display.

pause_time = 0 *Not tunable* Interval to pause between frames. 0 means wait for keystroke, Otherwise interval is in seconds (float)

title = Display window *Not tunable* Display window title text..

Input Ports

image Single frame image.

Data type : kwiver:image Flags : _required

timestamp Timestamp for input image.

Data type : kwiver:timestamp Flags : (none)

Output Ports

Pipefile Usage

The following sections describe the blocks needed to use this process in a pipe file

Pipefile block

```
# ================================================================
process disp
  :: image_viewer
:annotate_image         true
:pause_time             2.0
:footer                 footer_footer
:header                 header-header
# ================================================================
```

Pipefile connections

The following Input ports will need to be set

```
connect from processX.timestamp
        to   disp.timestamp
connect from processX.image
        to   disp.image
```

The follwing Output ports are available from this process

```
# There are no output port's for this process
```

Class Description

class `kwiver::image_viewer_process`
Display images.

Inherits from sprokit::process

image_writer

This sprokit process is used to ...

How To Make a Process

Plugins

Pipeline design

Overview

The design of the new pipeline is meant to address issues that have come up before and to add functionality that has been wanted for a while including Python support, interactive pipeline debugging, better concurrency support, and more.

Type Safety

The codebase strives for type safety where possible. This is achieved by using `typedef` to rename types. When applicable, `typedef` types also expose objects through only a `shared_ptr` to prevent unintentional deep copies from occurring and simplify memory management.

The use of `typedef` within the codebase also simplifies changing core types if necessary (e.g., replacing `std::shared_ptr` with a different managed pointer class).

Some of the core classes (i.e., `sprokit::datum` and `sprokit::stamp`) are immutable through their respective `typedef` and can only be created with static methods of the respective class which enforce that they can only be constructed in specific ways.

doxygenclass:: sprokit::datum

project kwiver

members

Introspection

Processes are designed to be introspected so that information about a process can be given at runtime. It also allows processes to be created at runtime and pipelines created dynamically. By abstracting out C++ types, language bindings do not need to deal with templates, custom bindings for every plugin, and other intricacies that bindings to C++ libraries usually entail.

Thread safety

Processes within the new pipeline are encouraged to be thread safe. When thread safety cannot be ensured, it must be explicitly marked. This is so that any situation where data is shared across threads where more than one thread expects to be able to modify the data is detected as an error.

Error Handling

Errors within the pipeline are indicated with exceptions. Exceptions allow the error to be handled at the appropriate level and if the error is not caught, the message will reach the user. This forces ignoring errors to be explicit since not all compilers allow decorating functions to warn when their return value is ignored.

Control Flow

The design of the ref sprokit::process class is such that the heavy lifting is done by the base class and specialized computations are handled as needed by a subclass. This allows a new process to be written with a minimum amount of boilerplate. Where special logic is required, a subclass can implement a c virtual method which can add supplemental logic to support a feature.

For example, when information about a port is requested, the ref sprokit::process::input_port_info method is called which delegates logic to the ref sprokit::process::_input_port_info method which can be overwritten. By default, it returns information about the port if it has been declared, otherwise it throws an exception that the port does not exist. To create ports on the fly, a process can reimplement ref sprokit::process::_input_port_info to create the port so that it exists and an exception is not thrown.

The rationale for not making ref sprokit::process::input_port_info c virtual is to enforce that API specifications are met. For example, when connecting edges, the main method makes sure that the edge is not c NULL and that the process has not been initialized yet.

Data Flow

Data flows within the pipeline via the ref sprokit::edge class which ensures thread-safe communication between processes. A process communicates with edges via its input and output ports. Ports are named communication sockets where edges may be connected to so that a process can send and receive data. Input ports may have at most one edge sending data to it while output ports may feed into any number of edges.

Ports

Ports are declared within a process and managed by the base ref sprokit::process class to minimize the amount of code that needs to be written to handle communication within the pipeline.

A port has a "type" associated with it which is used to detect errors when connecting incompatible ports with each other. These types are em logical types, not a type within a programming language. A c double can represent a distance or a time interval (or even a distance is a different unit!), but a port which uses a c double to a distance would have a type of c distance_in_meters, em not c double. There are two special types, one of which indicates that any type is accepted on the port and another which indicates that no data is ever expected on the port.

Ports can also have flags associated with them. Flags give extra information about the data that is expected on a port. A flag can indicate that the data on the port must be present to make any sense (either it's required for a computation or that if the result is ignored, there's no point in doing the computation in the first place), the data on the port should not be modified (because it is only a shallow copy and other processes modifying the data would invalidate results), or that the data for the port will be modified (used to cause errors when connected to a port with the previous flag).

Flags are meant to be used to bring attention to the fact that more is happening to data that flows through the port than normal.

Packets

Each data packet within an edge is made up of two parts: a status packet and a stamp. The stamp is used to ensure that the various flows through the pipeline are synchronized.

The status packet indicates the result of the computation that creates the result available on a port. It can indicate that the computation succeeded (with the result), failed (with the error message), could not be completed for some reason (e.g., not enough data), or complete (the input data is exhausted and no more results can be made). Having a status message for each result within the pipeline allows for more fine-grained data dependencies to be made. A process which fails to get some extra data related to its main data stream (e.g., metadata on a video frame) does not have to create invalid objects nor indicate failure to other, unrelated, ports.

A stamp consists of a step count and an increment. If two stamps have the same step count. A stamp's step count is incremented at the source for each new data element. Step counts are unitless and should only be used for ordering information. In fact, the ref sprokit::stamp interface enforces this and only provides a comparison operator between stamps. Since step counts are unitless and discrete, inserting elements into the stream requires that the step counts change.

The base ref sprokit::process class handles the common case for incoming and outgoing data. The default behavior is that if an input port is marked as being "required", its status message is aggregated with other required inputs:

- If a required input is complete, then the current process' computation is considered to be complete as well.
- Otherwise, if a required input is an error message, then the current process' computation is considered an error due to an error as input (following the GIGO principle).
- Otherwise, if a required input is empty, then the current process' computation is considered empty (the computation is missing data and cannot be completed).
- Then, since all of the required inputs are available, the stamps are checked to ensure that they are on the same step count.

If custom logic is required to manage ports or data, this control flow can be disabled piecemeal and handled manually. The status can check can be disabled on a per-process basis so that it can be managed in a special way.

Pipeline Execution

The execution of a pipeline is separate from the construction and verification. This allows specialized schedulers to be used in situations where some resource is constrained (one scheduler to keep memory usage low, another to minimize CPU contention, another for an I/O-heavy pipeline, and others).

Pipeline Declaration Files

Pipeline declaration files allow a pipeline to be loaded from a plain text description. They provide all of the information necessary to create and run a pipeline and may be composed of files containing pipeline specification information that are included into the main file

The '#' character is used to introduce a comment. All text from the '#' to the end of the line are considered comments.

A pipeline declaration file is made up of the following sections:

- Configuration Section
- Process Definition Section

- Connection Definition

Configuration Entries

Configuration entries are statements which add an entry to the configuration block for the pipeline. The general form for a configuration entry is a key / value pair, as shown below:

```
key = value
```

The key specification can be hierarchical and be specified with multiple components separated by a ':' character. Key components are described by the following regular expression `[a-zA-Z0-9_-]+`.

```
key:component:list = value
```

Each leading key component (the name before the ':') establishes a subblock in the configuration. These subblocks are used to group configuration entries for different sections of the application.

The value for a configuration entry is the character string that follows the '=' character. The value has leading and trailing blanks removed. Embedded blanks are preserved without the addition of enclosing quotes. If quotes are used in the value portion of the configuration entry, they are not processed in any way and remain part of the value string. That is, if you put quotes in the value component of a configuration entry, they will be there when the value is retrieved in the program.

Configuration items can have their values replaced or modified by subsequent configuration statements, unless the read-only flag is specified (see below).

The value component may also contain macro references that are replaced with other text as the config entry is processed. Macros can be used to dynamically adapt a config entry to its operating environment without requiring the entry to be hand edited. The macro substitution feature is described below.

Configuration entry attributes

Configuration keys may have attributes associated with them. These attributes are specified immediately after the configuration key. All attributes are enclosed in a single set of brackets (e.g. []). If a configuration key has more than one attribute they are all in the same set of brackets separated by a comma.

Currently the only understood flags are:

`flag{ro}` Marks the configuration value as read-only. A configuration that is marked as read only may not have the value subsequently modified in the pipeline file or programatically by the program.

`flag{tunable}` Marks the configuration value as tunable. A configuration entry that is marked as tunable can have a new value presented to the process during a reconfigure operation.

Examples:

```
foo[ro] = bar # results in foo = "bar"
foo[ro, tunable] = bar
```

Macro Substitution

The values for configuration elements can be composed from static text in the config file and dynamic text supplied by macro providers. The format of a macro specification is `$TYPE{name}` where **TYPE** is the name of macro provider and **name** requests a particular value to be supplied. The **name** entry is specific to each provider.

The text of the macro specification is only replaced. Any leading or trailing blanks will remain. If the value of a macro is not defined, the macro specification will be replaced with the null string.

Macro Providers

The macro providers are listed below and discussed in the following sections.

- LOCAL - locally defined values
- ENV - program environment
- CONFIG - values from current config block
- SYSENV - system environment

LOCAL Macro Provider

This macro provider supplies values that have been stored previously in the config file. Local values are specified in the config file using the ":=" operator. For example the config entry `mode := online` makes `$LOCAL{mode}` available in subsequent configuration entries.:

```
mode := online
...
config_file = data/$LOCAL{mode}/model.dat
```

This type of macro definition can appear anywhere in a config file and becomes available for use on the next line. The current block context has no effect on the name of the macro.

ENV Macro Provider

This macro provides gives access to the current program environment. The values of environment variables such as "HOME" can be used by specifying `$ENV{HOME}` in the config file.

CONFIG Macro Provider

This macro provider gives access to previously defined configuration entries. For example:

```
config foo
  bar = baz
```

makes the value available by specifying `$CONFIG{foo:bar}` to following lines in the config file as shown below.:

```
value = mode-$CONFIG{foo:bar}ify
```

SYSENV Macro Provider

This macro provider supports the following symbols derived from the current host operating system environment.

- curdir - current working directory
- homedir - current user's home directory
- pid - current process id
- numproc - number of processors in the current system
- totalvirtualmemory - number of KB of total virtual memory

- availablevirtualmemory - number of KB of available virtual memory
- totalphysicalmemory - number of KB of total physical memory
- availablephysicalmemory - number of KB of physical virtual memory
- hostname - name of the host computer
- domainname - name of the computer in the domain
- osname - name of the host operating system
- osdescription - description of the host operating system
- osplatform - platorm name (e.g. x86-64)
- osversion - version number for the host operating system
- iswindows - TRUE if running on Windows system
- islinux - TRUE if running on Linux system
- isapple - TRUE if running on Apple system
- is64bits - TRUE if running on a 64 bit machine

Block Specification

In some cases the fully qualified configuration key can become long and unwieldy. The block directive can be used to establish a configuration context to be applied to the enclosed configuration entries. `block alg` Starts a block with the *alg* block name and all entries within the block will have `alg:` prepended to the entry name.:

```
block alg
   mode = red       # becomes alg:mode = red
endblock
```

Blocks can be nested to an arbitrary depth with each providing context for the enclosed entries.:

```
block foo
  block bar:fizzle
    mode = yellow      # becomes foo:bar:fizzle:mode = yellow
  endblock
endblock
```

Including Files

The include directive logically inserts the contents of the specified file into the current file at the point of the include directive. Include files provide an easy way to break up large configurations into smaller reusable pieces.

`include filename`

If the file name is not an absolute path, it is located by scanning the current config search path. The manner in which the config include path is created is described in a following section. If the file is still not found, the stack of include directories is scanned from the current include file back to the initial config file. Macro substitution, as described below, is performed on the file name string before the searching is done.

Block specifications and include directives can be used together to build reusable and shareable configuration snippets.:

```
block main
  block alg_one
    include alg_foo.config
  endblock

  block alg_two
    include alg_foo.config
  endblock
endblock
```

In this case the same configuration structure can be used in two places in the overall configuration.

Include files can be nested to an arbitrary depth.

Relativepath Modifier

There are cases where an algorithm needs an external file containing binary data that is tied to a specific configuration. These data files are usually stored with the main configuration files. Specifying a full hard coded file path is not portable between different users and systems.

The solution is to specify the location of these external files relative to the configuration file and use the *relativepath* modifier construct a full, absolute path at run time by prepending the configuration file directory path to the value. The relativepath keyword appears before the *key* component of a configuration entry.:

```
relativepath data_file = ../data/online_dat.dat
```

If the current configuration file is `/home/vital/project/config/blue/foo.config`, the resulting config entry for **data_file** will be `/home/vital/project/config/blue/../data/online.dat`

The *relativepath* modifier can be applied to any configuration entry, but it only makes sense to use it with relative file specifications.

Configuration Section

Configuration sections introduce a named configuration subblock that can provide configuration entries to runtime components or make the entries available through the $CONFIG{key} macro.

The configuration blocks for *_pipeline* and *_scheduler* are described below.

The form of a configuration section is as follows:

```
config <key-path> <line-end>
      <config entries>
```

Examples

todo Explain examples.:

```
config common
  uncommon = value
  also:uncommon = value
```

Creates configuration items:

```
common:uncommon = value
common:also:uncommon = value
```

Another example:

```
config a:common:path
  uncommon:path:to:key = value
  other:uncommon:path:to:key = value
```

Creates configuration items:

```
a:common:path:uncommon:path:to:key = value
a:common:path:other:uncommon:path:to:key = value
```

Process definition Section

A process block adds a process to the pipeline with optional configuration items. Processes are added as an instance of registered process type with the specified name. Optional configuration entries can follow the process declaration. These configuration entries are made available to that process when it is started.

Specification

A process specification is as follows. An instance of the specified process-type is created and is available in the pipeline under the specified process-name:

```
process <process-name> :: <process-type>
  <config entries>
```

Examples

An instance of my_processes_type is created and named my_process:

```
process my_process :: my_process_type

process another_process
  :: awesome_process
     some_param = some_value
```

Non-blocking processes

A process can be declared as non-blocking which indicates that input data is to be dropped if the input port queues are full. This is useful for real-time processing where a process is the bottleneck.

The non-blocking behaviour is a process attribute that is specified as a configuration entryin the pipeline file. The syntax for this configuration option is as follows:

```
process blocking_process
  :: awesome_process
   _non_blocking = 2
```

The special "_non_blocking" configuration entry specifies the capacity of all incoming edges to the process. When the edges are full, the input data are dropped. The input edge size is set to two entries in the above example. This capacity specification overrides all other edge capacity controls for this process only.

Static port values

Declaring a port static allows a port to be supplied a constant value from the config in addition to the option of it being connected in the normal way. Ports are declared static when they are created by a process by adding the c flag_input_static option to the c declare_input_port() method.

When a port is declared as static, the value at this port may be supplied via the configuration using the special static/ prefix before the port name. The syntax for specifying static values is:

```
:static/<port-name> <key-value>
```

If a port is connected and also has a static value configured, the configured static value is ignored.

The following is an example of configuring a static port value.:

```
process my_process
  :: my_process_type
     static/port = value
```

Instrumenting Processes

A process may request to have its instrumentation calls handled by an external provider. This is done by adding the _instrumentation block to the process config.:

```
process my_process
  :: my_process_type
  block _instrumentation
     type = foo
     block  foo
       file = output.dat
       buffering = optimal
     endblock
  endblock
```

The type parameter specifies the instrumentation provider, "foo" in this case. If the special name "none" is specified, then no instrumentation provider is loaded. This is the same as not having the config block present. The remaining configuration items that start with "_instrumentation:<type>" are considered configuration data for the provider and are passed to the provider after it is loaded.

Connection Definition

A connection definition specifies how the output ports from a process are connected to the input ports of another process. These connections define the data flow of the pipeline graph.:

```
connect from <process-name> . <input-port-name> to <process-name> . <output-port-name>
```

Examples

This example connects a timestamp port to two different processes.:

```
connect from input.timestamp      to   stabilize  .timestamp
connect from input.timestamp      to   writer     .timestamp
```

Pipeline Edge Configuration

A pipeline edge is a connection between two ports. The behaviour of the edges can be configured if the defaults are not appropriate. Note that defining a process as non-blocking overrides all input edge configurations for that process only.

Pipeline edges are configured in a hierarchical manner. First there is the _pipeline:_edge config block which establishes the basic configuration for all edges. This can be specified as follows:

```
config _pipeline:_edge
       capacity = 30      # set default edge capacity
```

Currently the only attribute that can be configured is “capacity”.

The config for the edge type overrides the default configuration so that edges used to transport specific data types can be configured as a group. This edge type configuration is specified as follows:

```
config _pipeline:_edge_by_type
       image_container:capacity = 30
       timestamp:capacity = 4
```

Where *image_container* and *timestamp* are the type names used when defining process ports.

After this set of configurations have been applied, edges can be more specifically configured based on their connection description. An edge connection is described in the config as follows:

```
config _pipeline:_edge_by_conn
        <process>:<up_down>:<port> <value>
```

Where:

- <process> is the name of the process that is being connected.
- <up_down> is the direction of the connection. This is either “up” or “down”.
- <port> is the name of the port.

For the example, the following connection:

```
connect from input.timestamp
        to   stabilize.timestamp
```

can be described as follows:

```
config _pipeline:_edge_by_conn
   input:up:timestamp:capacity = 20
 s  tabilize:down:timestamp:capacity = 20
```

Both of these entries refer to the same edge, so in real life, you would only need one.

These different methods of configuring pipeline edges are applied in a hierarchial manner to allow general defaults to be set, and overridden using more specific edge attributes. This order is default capacity, edge by type, then edge by connection.

Scheduler configuration

Normally the pipeline is run with a default scheduler that assigns one thread to each process. A different scheduler can be specified in the config file. Configuration parameters for the scheduler can be specified in this section also.:

```
config _scheduler
   type = <scheduler-type>
```

Available scheduler types are:

- sync - Runs the pipeline synchronously in one thread.
- thread_per_process - Runs the pipeline using one thread per process.
- pythread_per_process - Runs the pipeline using one thread per process and supports processes written in python.
- thread_pool - Runs pipeline with a limited number of threads (not implemented).

The pythread_per_process is the only scheduler that supports processes written python.

Scheduler specific configuration entries are in a sub-block named as the scheduler. Currently these schedulers do not have any configuration parameters, but when they do, they would be configured as shown in the following example.

Example

The pipeline scheduler can selected with the pipeline configuration as follows:

```
config _scheduler
 type = thread_per_process

 # Configuration for thread_per_process scheduler
 thread_per_process:foo = bar

 # Configuration for sync scheduler
 sync:foos = bars
```

Clusters Definition File

A cluster is a collection of processes which can be treated as a single process for connection and configuration purposes. Clusters are defined in a slngle file with one cluster per file.

A cluster definition starts with the *cluster* keyword followed by the name of the cluster. A documentation section must follow the cluster name definition. Here is where you describe the purpose and function of the cluster in addition to any other important information about limitations or assumptions. Comments start with −− and continue to the end of the line.

The body of the cluster definition is made up of three types of declarations that may appear multiple times and in any order. These are:

- config specifier
- input mapping
- output mapping

A description is required after each one of these entries. The description starts with "–" and continues to the end of the line. These descriptions are different from typical comments you would put in a pipe file in that they are associated with the cluster elements and serve as user documentation for the cluster.

After the cluster has been defined, the constituent processes are defined. These processes are contained within the cluster and can be interconnected in any valid configuration.

config specifier

A configuration specification defines a configuration key with a value that is bound to the cluster. These configuration items are available for use within the cluster definition file and are referenced as <cluster-name>:<config-key>:

```
cluster_key = value
-- Describe configuration entry
```

Input mapping

The input mapping specification creates an input port on the cluster and defines how it is connected to a process (or processes) within the cluster. When a cluster is instantiated in a pipeline, connections can be made to these ports.:

```
imap from cport
     to   proc1.port
     to   proc2.port
-- Describe input port expected data type and
-- all other interesting details.
```

Output mapping

The output mappinc specification creates an output port on the cluster and defines how the data is supplied. When a cluster is instantiated, these output ports can be connected to downstream processes in the usual manner.:

```
omap from proc2.oport   to  cport
-- Describe output port data type and
-- all other interesting details.
```

An example cluster definition is as follows:

```
cluster <name>
  -- Description fo cluster.
  -- May extend to multiple lines.

  cluster_key = value
  -- Describe the config entry here.

  imap from cport
       to   proc1.port
       to   proc2.port
  -- Describe input port. Input port can be mapped
  -- to multiple process ports

  omap from proc2.oport    to  coport
  -- describe output port
```

The following is a more complicated example:

```
cluster configuration_provide
  -- Multiply a number by a constant factor.

  factor = 20
  -- The constant factor to multiply by.

  imap from factor  to   multiply.factor1
  -- The factor to multiply by.

  omap from multiply.product    to   product
  -- The product.

 # The following defines the contained processes
process const
  :: const_number
  value[ro]= $CONFIG{configuration_provide:factor}

process multiply
  :: multiplication

connect from const.number        to   multiply.factor2
```

Pipeline Example

How To Make a Pipeline

Vital	A set of data types and algorithm interfaces
Arrows	Various implementations of vital algorithms
Sprokit	An infrastructure for chaining together algorithms

CHAPTER 3

Tools

KWIVER provides the following tools

Process Explorer

Pipeline Runner

Process Explorer	Provides information about Sprokit Processes
#:doc:*Pipeline Runner</tools/pipeline_runner>*	Executes a pipe file

CHAPTER 4

Tutorials

The following links describe a set of kwiver tutorials. All the source code mentioned here is provided by the repository.

Visit the repository on how to get and build the KWIVER code base

As always, we would be happy to hear your comments and receive your contributions on any tutorial.

Fundamental Types and Algorithms

The following tutorials will demonstrate the basic functionality provided in kwiver. They will focus on the vital types available in kwiver and the various algorithm interfaces currelty supported. Each example highlights an area of functionality provied in KWIVER. The KWIVER examples directory contains executable code demonstrating the use of these types with various arrow implementations of the highligted algorithms.

Images	Learn about the fundamental image types and some basic I/O and algorithms
Detection	Focus on the data structures and algorithms used by object detection

Sprokit Pipelines

The following tutorials will use Sprokit pipeline files to chain together various algorithms to demonstrate applied examples. The KWIVER examples directory contains executable pipe files for each of the table entries below. In order to execute the provided pipeline file, follow the steps to set up KWIVER here

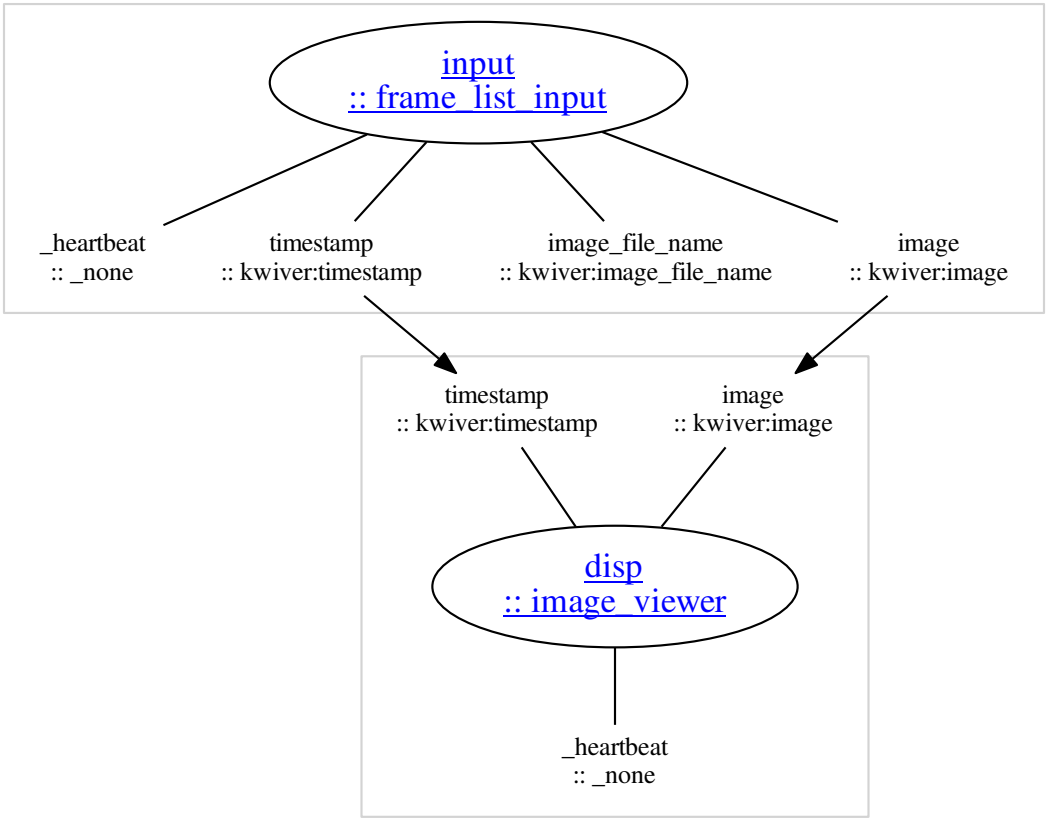
Numbers Flow	A simple 'Hello World' pipeline that outputs numbers to a file
Image Display	A pipe that loads and displays several images
Video Display	A pipe that loads and displays a video file
Hough Detection	Detect circles in images using a hough detector
Darknet Detection	Object detection using the Darnket library
Image Stabilization	Something cool that Matt Brown has done

Hello World

The examples/pipelines/number_flow.pipe and exaples/pipelines/number_flow_python are associated with this tutorial.

Simple Image

The examples/pipelines/image_display.pipe is associated with this tutorial.




```
# ================================================================
process input
  :: frame_list_input
# Input file containing new-line separated paths to sequential image
# files.
  image_list_file = @EXAMPLE_DIR@/pipelines/image_list.txt
  frame_time = .9

# Algorithm to use for 'image_reader'.
# Must be one of the following options:
#         - ocv
#         - vxl
```

```
  image_reader:type = ocv

# ================================================================
process disp
  :: image_viewer
:annotate_image         true
:pause_time             2.0
:footer                 footer_footer
:header                 header-header

# ================================================================
# global pipeline config
#
config _pipeline:_edge
       capacity = 10

# ================================================================
# connections
connect from input.timestamp
        to   disp.timestamp
connect from input.image
        to   disp.image

# -- end of file --
```

Simple Video

The examples/pipelines/video_display.pipe is associated with this tutorial.

Hough Detection

The examples/pipelines/hough_detector.pipe is associated with this tutorial.

Darknet Detection

Setup

In order to execute pipeline files, follow these steps to set up KWIVER

In order to run the pipelines associated with this tutorial you will need to download the associated data package. The download process is done via targets created in the build process. In a bash terminal in your KWIVER build directory, make the following targets:

```
make external_darknet_example
make setup_darknet_example
```

If you are using Visual Studio, manually build the external_darknet_example project, followed by the setup_darknet_example project.

This will pull, place, and configure all the data associated with thise exampe into <your KWIVER build directory>/examples/pipeline/darknet folder

The following files will be in the <build directory>/examples/pipelines/darknet folder:

- images - Directory containing images used in this example
- models - Directory containing configuration and weight files needed by Darknet
- output - Directory where new images will be placed when the pipeline executes
- video - Directory containing the video used in this example
- configure.cmake - CMake script to set configure *.in files specific to your system
- darknet_image.pipe - The pipe file to run Darknet on the provided example images
- darknet_image.pipe.in - The pipe file to be configured to run on your system
- darknet_video.pipe - The pipe file to run Darknet on the provided example video
- darknet_video.pipe.in - The pipe file to be configured to run on your system
- image_list.txt - The images to be used by the darknet_image.pipe file
- image_list.txt.in - The list file to be configured to run on your system
- readme.txt - This tutorial supersedes content in this file

Execution

Run the following command from the kwiver buildbin directory (binrelease on windows) Relativly point to the darknet_image.pipe or darknet_video.pipe file like this:

```
# Windows Example :
pipeline_runner -p ..\..\examples\pipelines\darknet\darknet_image.pipe
# Linux Example :
./pipeline_runner -p ../examples/pipelines/darknet/darknet_image.pipe
```

The darknet_image.pipe file will put all generated output to the examples/pipelines/darknet/output/images

The darknet_video.pipe file will put all generated output to the examples/pipelines/darknet/output/video

We will dig into more details for each pipeline file in the following sections.

Image Detection

The darknet_image.pipe file will run a pre-trained YOLO v2 object detector from darknet against the provided image files. The detector is trained to identify people and vehicles in images.

Follow these links for more information about pipeline design and files.

This pipefile will execute the following processes for each image specified:

Processes
Name input **Type** *frame_list_input* **Description** Reads the images in the image_list.txt file
Name yolo_v2 **Type** *image_object_detector* **Description** Configured to use the darknet implementation of image_object_detector
Name draw **Type** *draw_detected_object_boxes* **Description** Creates a copy of the current image, then draw the detection boxes on it created by the yolo_v2 process
Name disp **Type** *image_viewer* **Description** Shows the new image copy with detection boxes in a window as the pipeline runs
Name write **Type** *image_writer* **Description** Writes the new image copy with detection boxes to the specified directory
Name yolo_v2_kw18_writer **Type** *detected_object_output* **Description** Writes the detected_object_set object to an ascii file in kw18 format
Name yolo_v2_csv_writer **Type** *detected_object_output* **Description** Writes the detected_object_set object to an ascii file in csv format

Video Detection

TODO

Image Stabilization

The examples/pipelines/image_display.pipe is associated with this tutorial.

CHAPTER 5

Extending Kwiver

This section discusses the various ways KWIVER can be extended with vital types, algorithms (Arrows) and processes.

Creating a new Algorithm

Adding Algorithm Implementations

How to configure an Algorithm

How to Instantiate an Algorithm

How to Wrap an Algorithm with a Process

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Index

I

K