The IPOL Demo Description Lines (DDL)

Compiled on Monday 13th January, 2025 at 17:55

This document contains the technical documentation for the Demo Description Lines (DDL) for the IPOL Demo System 2.0 for the real-time demos generation from their textual description.

Contents

1 Introduction

The Demo Description Lines (DDL) define an abstract syntax, written in JSON (JavaScript Object Notation) format, that specifies the IPOL demos. Their main objective is to simplify as much as possible the creation of demos by describing them without the need of writing Python or HTML. This allows fast demo editing in the journal. The following sections describe each of the main keys of the DDL:

- *general*: general options (required);
- *build*: download and compile the source code (required);
- *inputs*: description of the inputs (optional);
- *params*: description of the parameters and user control (optional);
- *run*: script or binary which needs to be called for the execution, along with its parameters (required);
- *archive*: which parameters and files will be stored in the archive (optional);
- *results*: which elements will be displayed as results (required).

The IPOL control panel provides a JSON editor with a simple validator. Most of the syntax errors are detected in real time and reported by this graphical tool.

2 The *general* section

The general section describes global information about the demo. It is a set of (key, value) pairs, described in the following table. The column *req* refers to a required field. This type of tables will be used in all the sections of this document.

key	description	req
demo_title	Title of the demo.	no
description	Description to be shown at the beginning of the demo page. It contains HTML or plain text as a single string.	no

input_description	Description of the inputs. It contains HTML as a single string or as an array of string that will be concatenated and separated with spaces.	no
param_description	Description for the parameters. It contains HTML code as a single string or as an array of string that will be concatenated and separated with spaces.	no
xlink_article	Link to the article webpage	no
requirements	It specifies particular requirements needed for the execution of the demo, separated by commas. e.g. Matlab.	no
custom_js	It allows to give the URL of a custom Javascript file. This script contains extra JS code that allows to personalize the interaction and look of the front-end. It should only be used in very special cases there there is not any other alternative than overwriting the default behavior or look.	no
timeout	It specifies maximum time in seconds allows to execute the algorithm. If the execution takes longer than the specified time the system stops the exection.	no

Table 1: Fields in the *general* section.

3 The *build* section

The build section is a set of one or more sets, each one providing information to obtain and compile the source codes needed for a given demo. If the demo needs to compile several files from different links, the build sets must be indexed as build1, build2,..., build$n$ being n the total number of builds (see the example below). It is mandatory to write at least build1. If a demo needs to make use of a python package in order to execute the user will have to specify the requirements.txt file location inside the source binary compressed file. Three steps are needed to build a demo:

- Download the original sources codes (with optional userid and password for private demos);

- Build the executables after the download;
- Copy the needed files in a run context to execute the demo.

key	description	req
url	Link to download the source codes as a compressed file.	yes
username	Username for private demos.	no
password	Password for private demos.	no
construct	Shell command needed to compile the downloaded source code.	no
move	List of files needed to execute the demo separated with commas (see example below)	yes
virtualenv	Creates a python 3 virtualenv to install any needed python package inside the bin folder.	no

Table 2: Build fields

Example: In this case, the demo needs to compile from two different compressed files. The DDL uses the *move* statement to copy the obtained files after the compilation into a run context.

```
"build": {
    "build1": {
        "url": "http://www.ipol.im/pub/art/2014/82/
    sift_anatomy_20141201.zip",
        "construct": "cd sift_anatomy_20141201 && make",
        "move": "sift_anatomy_20141201/bin/sift_cli,
    sift_anatomy_20141201/bin/match_cli"
    },
    "build2":{
        "url": "http://dev.ipol.im/~monasse/orthoPose_1.0.tar
    .gz",
        "construct": "matlab -nodisplay -nosplash -nodesktop
    -r \"cd orthoPose_1.0/; mcc -m mainPoseEstimation.m -a lib
    /; exit;\"",
        "move": "orthoPose_1.0/mainPoseEstimation, orthoPose_
    1.0/run_mainPoseEstimation.sh"
        "virtualenv": "sift_anatomy_20141201/requirements.txt
    "
    }
}
```

4 The *inputs* section

The inputs section describes the characteristics of the input data for the algorithm.

4.1 image

key	description	req
type	type of the input: image	yes
description	Short name or description. This is used by the web interface.	no
max_pixels	This value sets the maximum number of pixels allowed for the input. If the size of the image is over this the limit it will be resized. The value can be a number or an arithmetic expression (ex: "1000*1000" = 1 Mpx).	yes
max_weight	Maximum weight (in bytes) of an input file. This prevents uploading too large files. The value can be a number or an arithmetic expression (ex: "100*1024*1024"= 100 Mb).	no
dtype	Final format for the image. Some examples: • *1x8i*: gray, unsigned integer 8 bits; • *3x8i*: color, RGB unsigned integer 8 bits; • *1x16i*: gray, unsigned integer 16 bits; • *3x16i*: color, RGB unsigned integer 16 bits.	yes
ext	input extension (ie. file format)	yes
forbid_preprocess	Must be a boolean value. Forbids any pre-processing of the input data. Submitted image is kept as-is. Used by algorithms like noise estimation or modification detection, where re-sampling will affect results. If a processing is needed according to the expected properties, an error message will be displayed to the user. This will also remove the crop feature from the interface.	no
control	String to include an interactive control. Possible values are "mask", "dots" and "lines" each for a different kind of mask drawing behaviour.	no

Table 3: Fields for an *image* as input.

4.2 video

key	description	req
type	type of the input: video	yes
description	Short name or description. This is used by the web interface.	no
as_frames	Boolean value. The input video will be converted to png images for each frame according to the max_frames field. Frames will be stored in a temporal folder inside the execution directory with the name. (ex: ./input_0/frame_000.png)	no
max_pixels	This value sets the maximum number of pixels allowed for the input video per frame. If the size of the input is over, the video frames will be resized. The value can be a number or an arithmetic expression (ex: "1000*1000" = 1 Mpx).	yes
max_frames	Maximum number of frames after conversion, either as frames or video.	yes
max_weight	Maximum weight (in bytes) of an input file. This prevents uploading too large files. The value can be a number or an arithmetic expression (ex: "100*1024*1024"= 100 Mb).	no
forbid_preprocess	Forbids any pre-processing of the input data by the IPOL system. Submitted video is kept as-is. Used by algorithms like noise-estimation or modification detection, where re-sampling will affect results.If a processing is needed according to the expected properties, an error message will be displayed to the user.	no

Table 4: Fields for a *video* as input.

4.3 data

The *data* type is used when the input type is other than an image or a video. Submitted data is kept as it is. The extension of your data file should be defined in the "ext" column.

key	description	req
type	type of the input: data	yes
description	Short name or description. This is used by the web interface.	no
max_weight	Maximum weight (in bytes) of an input file. This prevents uploading too large files. The value can be a number or an arithmetic expression (ex: "100*1024*1024"= 100 Mb).	no
ext	input extension (ie. file format, eg: .txt, .tiff)	yes

Table 5: Fields for the *data* as input.

Example: An example of the *data* input for a demo is shown below. An input file with the .txt format is required in this case.

```
"inputs": [
    {
    "description": "Text file containing the curve points",
    "max_weight": 524288000,
    "ext": ".txt",
    "required": true,
    "type": "data"
    }
]
```

4.4 map

The *map* type the interface makes the demo show a map of the Earth where the user can drawn one or more polygons interactively. The selection is passed to the demo's code as a list of GeoJSON features containing geometric coordinates.

Figure 1 shows the control and its key elements. To draw a polygon there is a toolbox (1) which allows to start drawing a polygon and to remove completely the last one. Click on the upper icon to start drawing and click on the map

Figure 1: The IPOL's map interface. 1) The controls to draw and remove polygons, 2) A polygon already draw, 3) A polygon being drawn, and 4) the information about the current polygon.

to add as many vertices as needed. You can also start adding vertices by clicking the right button of the mouse. When done, click the right button of the mouse. After that, the polygon will appear as finished (2). You can drawn more than one polygon, if needed. After finishing with the first, you can draw a second one (3). The information about the current polygon is show above (4).

The map is an interactive 3D projection. You can move around and change the location using the mouse and dragging with the left button, or using the keyboard cursors. With the right button of the mouse you can rotate the map and change the orientation of the camera. The zoom can be adjusted with the wheel of the mouse or with '+'/'-' in the keyboard.

The demo will receive a GeoJSON file containing the coordinates as a list of geometric features. For example, a selection made of two polygons of three and four vertices would be encoded as follows:

```
{
    "type": "FeatureCollection",
    "features": [
        {
            "id": "1a3e0b6db723596db6da77b80ea0904f",
            "type": "Feature",
            "properties": {},
            "geometry": {
                "coordinates": [
```

```
                    [
                        [
                            -3.69022875121982,
                            40.41938883192799
                        ],
                        [
                            -3.6927607565289122,
                            40.41605617841688
                        ],
                        [
                            -3.6802294760118457,
                            40.41563141659978
                        ],
                        [
                            -3.69022875121982,
                            40.41938883192799
                        ]
                    ]
                ],
                "type": "Polygon"
            }
        },
        {
            "id": "ca0e17328f3186e57d84fd8c535f6ab5",
            "type": "Feature",
            "properties": {},
            "geometry": {
                "coordinates": [
                    [
                        [
                            -3.7089827566515794,
                            40.42069570981127
                        ],
                        [
                            -3.695464423216549,
                            40.41857202035922
                        ],
                        [
                            -3.70220213226159,
                            40.4154026975869
                        ],
                        [
                            -3.712544730223499,
                            40.417526487080124
                        ],
                        [
                            -3.7089827566515794,
                            40.42069570981127
                        ]
```

```
                    ]
                ],
                "type": "Polygon"
            }
        }
    ]
}
```

Note that a polygon of N vertices is encoded as a list of $N + 1$ coordinates, where the first equals the last. This is the GIS standard to represent a topologically closed curve.

Here it follows an example to read the GeoJSON file in Python:

```
#!/usr/bin/env python3
# -*- coding: UTF-8 -*-

import json
import argparse

def print_polygons(polygons):
    '''
    Print the information of given polygons
    '''
    if not polygons:
        print("No polygons were drawn in the map")
        return

    print(f"{len(polygons)} polygon(s) were drawn in the map:")
    for polygon in polygons:
        print(f"\t- polygon with {len(polygon)} vertices:")
        for coord in polygon:
            print(f"\t\t{coord}")

parser = argparse.ArgumentParser(description='GeoJSON example.')
parser.add_argument('--json', type=str, help='Input filename', default='input_0.json')
args = parser.parse_args()

# Load GeoJSON
with open(args.json, "rt") as f:
```

```
    D = json.load(f)

# Store here the list of polygons found in the GeoJSON file
polygons = []

# Parse the GeoJSON
# The coordinates are in feature['geometry']['coordinates']
for feature in D['features']:
    if 'geometry' not in feature:
        continue

    if 'coordinates' not in feature['geometry']:
        continue

    coordinates = feature['geometry']['coordinates'][0]
    polygons.append(coordinates)

# Finally, print the information of the polygons found
print_polygons(polygons)
```

When a demo uses a map, it can only contain that single input in the DDL.

key	description	req
type	type of the input: map	yes
center	longitude and latitude (example: [-3.703790, 40.416775] to center the map in Madrid)	no
ext	input extension (for example, .json)	yes

Table 6: Fields of *map* input type.

5 The *params* section

The *params* section describes the set of parameters needed by a demo, their constraints and the visual appearance of the user control. It is defined as an array of sets, where each set contains (key, value) pairs. In this section, we show examples of the expected appearance of these parameters in the web interface. The look of the controls might differ depending on the operating system and the browser used.

5.1 range

The *range* type is used as an horizontal slider constrained by a minimum and a maximum numeric values. It can be moved with the mouse or by using the arrow keys according to the step value fixed in the DDL. The user control is similar to the one at Figure 2.

key	description	req
type	range	yes
id	Used to identify the parameter.	yes
label	A name and/or description of the parameter. It appears on the left side in the web interface.	no
comments	A description of the parameter. It appears on the right side in the web interface.	no
visible	Javascript expression evaluated as a boolean.	no
values	Sets min, max, step and default values using a key/value scheme { "min":val, "max":val, "step":val, "default":val }. Ex: to select a value included in (-1, -0.5, 0, 0.5, 1) write `"values": {"min": -5, "max": 5, "step": 0.5, "default": 0}`	yes

Table 7: Fields for the properties of the *range* type.

Figure 2: Range type example. It shows a slider with values from 0.02 to 0.2.

5.2 selection_collapsed

The *selection_collapsed* type returns one string selected by a key (for example, a color code selected by name). The user control is a dropdown select similar to the one in Figure 3.

key	description	req
type	selection_collapsed	yes
id	Used to identify the parameter.	yes

label	A name and/or description of the parameter. It appears on the left side in the web interface.	no
comments	A description of the parameter. It appears on the right side in the web interface.	no
visible	Javascript expression evaluated as a boolean.	no
values	set of (key, value) pairs, where the key is the displayed text and the value is the string returned, for example `"values": {"black": "000000", "white": "FFFFFF"}`	yes
default_value	defines the default value for this parameter, should be one the values defined in 'values'.	yes

Table 8: Fields for the properties of the *selection_collapsed* type.

Figure 3: Selection collapsed example. In this case, the selection offers five options to choose.

5.3 selection_radio

The *selection_radio* returns one string selected by a key (for example, a color code selected by name). The user control is a set of radio buttons as in Figure 4.

key	description	req
type	selection_radio	yes
id	Used to identify the parameter.	yes
label	Name and/or description of the parameter. It appears on the left side in the web interface.	no
comments	Description of the parameter. It appears on the right side in the web interface.	no
visible	Javascript expression evaluated as a boolean.	no
values	set of (key, value) pairs, where the key is the displayed text and the value is the string returned, for example `"values": {"black": "000000", "white": "FFFFFF"}`	yes
default_value	defines the default value for this parameter, should be one the values defined in 'values'.	yes
vertical	It is boolean value. The button distribution is vertical when the value is activated (true), otherwise, the visualization is horizontal as default.	no

Table 9: Fields for the properties of the *selection_radio* type.

Figure 4: Radio buttons example. The label description is Mode and the parameter offers two radio buttons. The vertical option is disabled.

5.4 label

The *label* type can be used to separate groups of parameters or to include html fields (images, external links, etc.) in the web interface.

key	description	req
type	label	yes
label	HTML text to display, as a single string or as an array of strings.	yes
visible	Javascript expression evaluated as a boolean.	no

Table 10: Fields for the properties of the *label* type.

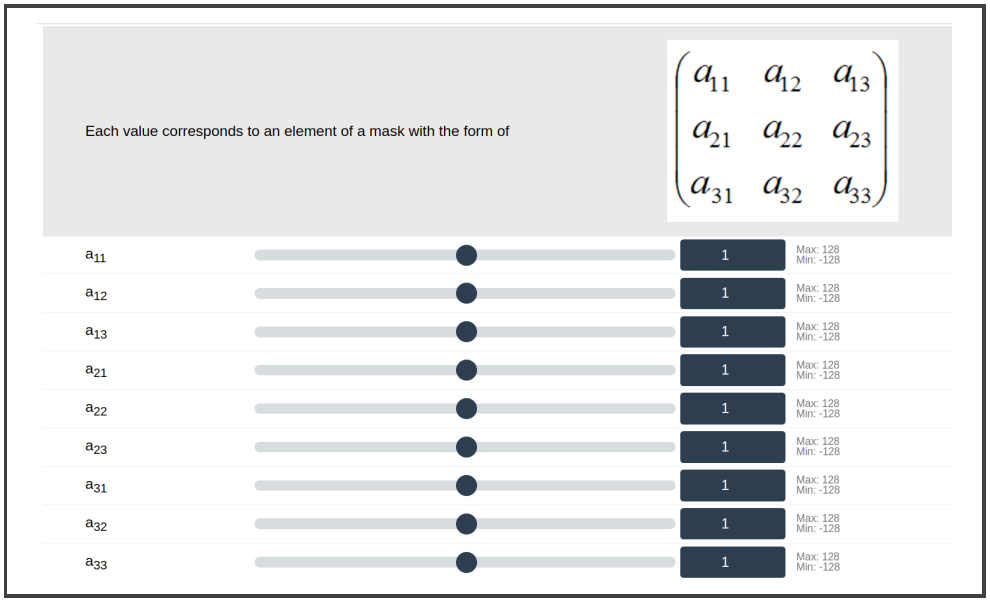

Figure 5: Label example. The label explains that the sliders below represent matrix values according to the image depicted in the label.

5.5 checkbox

The *checkbox* type returns a boolean value. The user control is a checkbox similar to the one in Figure 6.

key	description	req
type	checkbox	yes
id	Used to identify the parameter.	yes
label	A name and/or description of the parameter. It appears on the left side.	no

comments	A description of the parameter. It appears on the right side in the web interface.	no
visible	Javascript expression evaluated as a boolean.	no
default_value	boolean: True for checked	

Table 11: Fields that manages the properties of the *checkbox* type.

Figure 6: Checkbox example. This can be used in the demos that need to activate or not an option.

5.6 numeric

The *numeric* type returns a numeric value validated against constraints (min, max). The user control is an input field with numbers. Note that this is quite similar to the *range* type but without the slider. You can see an example in Figure 7.

key	description	req
type	numeric	yes
id	Used to identify the parameter.	yes
label	A name and/or description of the parameter. It appears on the left side.	no
comments	A description of the parameter. It appears on the right side in the web interface.	no
visible	Javascript expression evaluated as a boolean.	no
values	Set min, max, and default values using the following key/value scheme `"values": {"min": -5, "max": 5, "default": 0}`	yes

Table 12: Fields for the properties of the *numeric* type.

Figure 7: Numeric example. The label explains that the sliders below represent matrix values according to the image depicted in the label.

5.7 text

The *text* type returns a string. The user control is an input field.

key	description	req
type	text	yes
id	Used to identify the parameter.	yes
label	A name and/or description of the parameter. It appears on the left side.	no
comments	A description of the parameter. It appears on the right side in the web interface.	no
visible	Javascript expression evaluated as a boolean.	no
values	set maxlength in characters and default values using the following key/value scheme `"values": {"maxlength": 3, "default": "fr"}`	no

Table 13: Fields for the properties of the *text* type.

Figure 8: Text example. The user can write some text as parameter for the demo.

5.8 textarea

This param allows including textual information as a parameter. The text must be written in the DDL with the correct format. This means that the text area can show your message with new lines, skip lines and the normal ways of a file if the encoding format is correct. For instance, if you want that your text area looks like in Figure 9, the default value must be as in the following example:

Examples: Example of a DDL when using a text area.

```
1 {
2   "default_value": "INFORMATION ABOUT FIRST RECTANGLE
     CONTAINER\r\nNORMALIZED IMAGE DIMENSION\r\nwidth_float = 1
     .413793\r\n",
3           "wrap":false,
```

```
            "height": 5,
            "type": "textarea",
            "id": "file_1",
            "label": "Parameter file of the model.",
            "comments":"<b>You can also change the parameters
    in the text.<b>",
}
```

key	description	req
type	textarea	yes
label	name and/or description of the parameter. It appears on the left side.	no
id	Used to identify the parameter.	yes
default_value	Text to include in the text area	no
visible	Javascript expression evaluated as a boolean.	no
height	Set the height of your textarea. The maximum value is 2000px.	no
width	Set the width of your textarea. If you do not include the parameter it will be 100%.	no
wrap	This attribute specifies how the text in a text area is wrapped. False means that the line is not adapted to the textarea. True the opposite.	no

Table 14: Fields for the properties of the *textarea* type.

Figure 9: textarea example. The label explains that the sliders below represent matrix values according to the image depicted in the label.

6 The *run* section

The *run* section specifies which script or binary needs to be called to run a demo, along with its parameters. The input files defined in the *input* section

are available as arguments with a normalized name input_{0..n}.{extension} (ex: input_0.png). The parameters define in *params* section are available by their id with $ as a prefix (ex: "id": "width", $width).

In this example, the demo is executed by the binary file jpegblocks (compiled and moved in the *build* section), with input_0.png as an input and $block_size as a parameter.

```
1  "run": "jpegblocks input_0.png $block_size"
```

The execution is then passed to the *run.sh* script, provided in the optional demoextras.zip, with input_0.png $width as arguments.

```
1  "run": "${demoextras}/run.sh input_0.png $width"
```

In adition, there are other variables that will be substituted before execution (run section). As shown before, to use a variable just insert the name of the variable between curly brackets preceded by a dollar sign.

In the previous example the demoExtras path will be replaced in order that the run section executes a script inside the demoExtras folder. It follows a list of all available variables for the run section:

- **demoextras**: will be replaced by the demoExtras' path of the current demo,
- **matlab_path**: will be replaced by the path to the current MATLAB installation,
- **bin**: will be replaced by the directory with the compiled code, or any moved element,
- **virtualenv**: will be replaced by the path to the virtualenv, if any. This folder contains the scripts needed to activate the virtualenv.

7 The *archive* section

The *archive* section defines the data (files, parameters, running time, ...) to be stored for each experiment performed with original data uploaded by the user. The normal behaviour is to archive only the original data uploaded by the users. However, a demo editor can also allow to store all the experiments

done even if the data does not come from an upload (see the *archive_always* field).

key	description	req
files	(key, value) pairs where key is the file to archive and value is the name.	no
hidden_files	This field contains files as in the above one. This is used to store files required for a correct reconstruction of an experiment but that the demo editor does not want to show in the archive.	no
params	List of parameters to archive.	no
enable_reconstruct	Show a button to reconstruct an experiment stored in the archive.	no
archive_always	The archive will store the experiments even if they are performed with the data proposed by the demo (if the private mode is not set).	no

Table 15: The *archive* section, properties

Example: Here, we see DDL's needed to activate the reconstruct and archive_always options. They also specify the files and params that must be stored in a particular order. The running time for each execution is also stored.

```
"archive":
  {
    "enable_reconstruct": true,
    "archive_always": true,
    "files" :
      { "input_0.png"                 : "input image",
        "primitives.txt"              : "Primitives"
      },
    "params" :
      [ "high_threshold_canny",
        "initial_distortion_parameter",
        "angle_point_orientation_max_difference" ],
    "info"   : { "run_time": "run time" }
  }
```

8 The *results* section

The results specifies what to display as a result of an experiment. It is an array of sets, where each entry describes one type of output from the algorithm. There are displayed sequentially one below the other.

8.1 gallery

The results *gallery* type displays images. These ones can be displayed in different rows and columns. The example of this section shows the Demo Description Lines required for the visualization showed in the Figure 10. Notice that each row implies an array for each image. In the case that you only want to display one image, the DDL only required an expresion like: `"label":{ "img": "name_of_file.extension"}`

key	description	req
type	gallery	yes
visible	A Javascript expression evaluated as a boolean.	no
label	HTML label for the gallery, can be either a single string or a list of string that will be concatenated.	no
contents	A set of sets, each entry describes one or more images with a key and properties: • *key*, required, a label for the entry, could be a string or an evaluated expression in case of repeat; • *img*, required, a string with a filename or an array of strings with filenames; • *visible*, optional, a Javascript expression evaluated to a boolean; • *repeat*, optional, a Javascript expression, will create a loop in the form idx=0..range-1	yes

Table 16: Properties of the *gallery* type in the results section.

Example: The next example shows the DDL's needed for displaying three images per row in an image gallery.

```
{
 "contents": {
    "IPOL colors (scaled, no level lines)": {
       "img": ["rof_ipoln.png", "ground_truth_ipoln.png", "
   color_wheel_ipoln.png"]
    },
    "IPOL colors (unscaled, with level lines)":{
```

```
7         "img": [ "rof_ipol1.png", "ground_truth_ipol1.png", "
      color_wheel_ipol1.png"]
8       },
9       "Middlebury Colors":{
10        "img": ["rof_middlebury.png", "ground_truth_middlebury
      .png", "color_wheel_middlebury.png"]
11      },
12      "Arrows":{
13        "img": ["rof_arrows.png", "ground_truth_arrows.png", "
      color_wheel_arrows.png"]
14      },
15      "Input images (I1,I2) ": {
16        "img": [ "input_0.png", "input_1.png"]
17      },
18      "label": "<h3>Optical Flow (Calculated flow, Ground Truth
      )</h3>",
19      "type": "gallery",
20      "visible" : "info.gt"
21  },
```

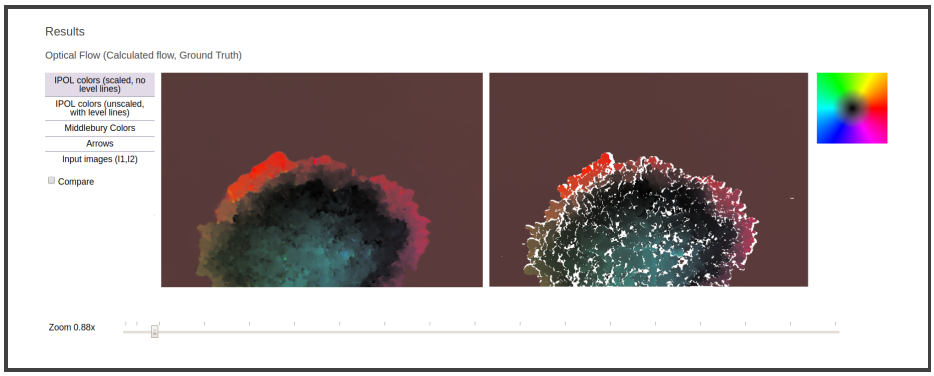

Figure 10: Example of an image gallery. In this example, we see three images per row.

8.2 gallery_video

The results *gallery_video* type displays video files. This type is quite similar to the previous one but related to the visualization of video contents.

key	description	req
type	gallery_video	yes

visible	A Javascript expression evaluated as a boolean.	no
label	HTML label for the gallery, can be either a single string or a list of string that will be concatenated.	no
contents	A set of sets, each entry describes one or more images with a key and properties: • *key*, required, a label for the entry, could be a string or an evaluated expression in case of repeat; • *img*, required, a string with a filename or an array of strings with filenames; • *visible*, optional, a Javascript expression evaluated to a boolean; • *repeat*, optional, a Javascript expression, will create a loop in the form idx=0..range-1	yes

Table 17: Properties of the *gallery_video* type in the results section.

Examples: Advanced example, mixing repeat, visible, using an array of filenames.

```
{
    "type": "video_gallery",
    "label": "<b>Video gallery</b>",
    "display": "grid",
    "visible": "1==1",
    "contents": {
        "Input_0": {
            "video":  "'input_0.mp4'",
            "visible": "1==1"
        },
        "'Scale_'+idx": {
            "video":  "'scaled_'+idx+'.mp4'",
            "repeat": "4"
        }
    }
}
```

8.3 file_download

The results *file_download* type proposes a link to download a file.

key	description	req

type	file_download	yes
visible	A Javascript expression evaluated as a boolean.	no
repeat	range expression (evaluated in Javascript): will create a loop in the form idx=0..range-1	no
label	HTML title associated to the file to download. In case of repeat, evaluated as an expression with idx variable, otherwise, can be evaluated if it starts with a single quote.	yes
contents	either a single string of the filename to download, or a list of label:filename pairs for files to download. In case of repeat, evaluated as an expression with idx variable.	yes

Table 18: Properties of the *file_download* type in the results section.

We show two examples: the first one is to download one result and the second is to download several results in the same line.

Examples :

```
{
    "type"     : "file_download",
    "label"    : "Download Hough result",
    "contents" : "output_hough.png"
}
```

```
{
    "type"     : "file_download",
    "label"    : "<h3>Download computed optical flow:</h3>",
    "contents" : {
        "tiff": "stuff_tvl1.tiff",
        "flo" : "stuff_tvl1.flo",
        "uv"  : "stuff_tvl1.uv"
    }
}
```

Example using *repeat*:

```
{
    "type"      : "file_download",
    "repeat"    : "params.scales",
    "label"     : "'Download the estimations obtained at scale
    '+idx",
    "contents" : "'estimation_s'+idx+'.txt'"
}
```

8.4 html_text

It displays the given HTML-encoded content.

key	description	req
type	html_text	yes
visible	A Javascript expression evaluated as a boolean.	no
contents	An array of strings, that will be concatenated to form the HTML content. This content can contain Javascript expression if it starts with a single quote.	yes

Table 19: Properties of the *html_text* type in the results section.

Example :

```
{
    "type"          : "html_text",
    "contents"      : [
        "'<p style=\"font-size:85%\">",
        "* &ldquo;Exact&rdquo; is computed with FIR, ",
        "DCT for &sigma; &gt; 2 ",
        "(using '+params.sigma<=2?'FIR':'DCT'+",
        "'</p>'"
    ]
}
```

8.5 html_file

It displays the given HTML file.

key	description	req
type	html_file	yes
visible	A Javascript expression evaluated as a boolean.	no
contents	A string with a filename.	yes

Table 20: Properties of the *html_file* type in the results section.

Example :

```
{
    "type"          : "html_file",
    "contents"      : "output.html"
}
```

8.6 text_file

It displays the contents of a text file.

key	description	req
type	text_file	yes
visible	Javascript expression evaluated as a boolean.	no
label	HTML label.	yes
contents	A text filename to display.	yes
style	CSS rules written in a JSON string, ex `"style": "{'font-weight': 'bolder', 'color': 'red'}"`	yes

Table 21: Properties of the *text_file* type in the results section.

Example :

```
{
    "type"          : "text_file",
    "label"         : "<h2>Output<h2>",
    "contents"      : "stdout.txt",
    "style"         : "{'width': '40em', 'height': '16em', '
   background-color': '#FFE'}"
```

```
}
```

8.7 message

The *message* type displays a text message with a predefined color. This can be used for warning or error messages.

key	description	req
type	message	yes
visible	Javascript expression evaluated as a boolean.	no
contents	A string which will be evaluated by Javascript to get the message.	yes
textColor	The name of a color or a CSS-compatible color.	no

Table 22: Properties of the *message* type in the results section.

Examples :

```
{
    "contents": "'Image too small: the input image needs to be at least 42000 pixels to get a reliable estimate<br>Forced to use one bin for the estimation.'",
    "type": "message",
    "textColor": "red",
    "visible": "info.sizeX * info.sizeY < 42000"
}
```