VideoLab Manual

VideoLab is a tool for analyzing a series of consecutive images, such as videos. This tool is specifically designed for computer visioners, who needs to closely watch a series of images back and forth and do some basic processing to the images.

This tool supports the process of a series of image organized in a video file, a super ppm file, a folder of images, or even more complicated data structures like optical flow. These files can be from your local machine or from a remote server. With this tool, you can open as many image series as you want, browse them conveniently, process them with simple scripts, or convert them to different formats.

Table of Contents

Installation

To deploy VideoLab to your local machine, git clone the repository to your hard drive. You need to install conda (https://conda.io/projects/conda/en/latest/user-guide/install/index.html) and rsync (brew install rsync / sudo apt install rsync). Then, you can follow the commands listed below to establish the conda environment:

- conda create -n your_env_name python=3.8
- conda activate your_env_name
- pip install PyQt5
- pip install numpy
- pip install Pillow
- pip install opencv-python
- pip install opencv-contrib-python
- pip install decord
- pip install pexpect

To fast deploy VideoLab to Gypsum, you need to do the following steps:

- git clone the repository to Gypsum. Copy the path you save the repository in the server (referred as [processor_path]),
- run “mkdir -p tmp/scripts log” in the cloned folder on Gypsum,
- download all the third party modules (click to download), unzip them, and put them to [processor_path]/utils/transformers/thirdparty on Gypsum,
- on your local machine, use configs/serverConfigs/mygypsum.json as a template to create a server configuration file on the same path for your own need (or edit it directly). More specifically:
    - Type your user name into field “username”;
    - Paste processor_path to the corresponding field;
    - Type password or key to “auth_method” field depending on how you log into the server. If you use password to login to the server, put “password” to this field; if you use sshkey to log into Gypsum, you also need to add “key_file” field with the path pointing to the private key file;
    - You don’t need to edit “template_path” and its associated file since it has been adapted to Gypsum already.
- if you created a new server configuration file, you also need to register it to configs/serverConfigs/registeredServerConfigs.json. Just add an item to it. Field “name” is a nickname for your server configuration, and field “file” directs to the path of the configuration file.
- create the environment following what you did on your local machine. Notice: to use the third party module, you may need to install more packages. Please refer to section Advanced Topics.

If you are using other servers or want to add more server configurations, please refer to section Advanced Topics.

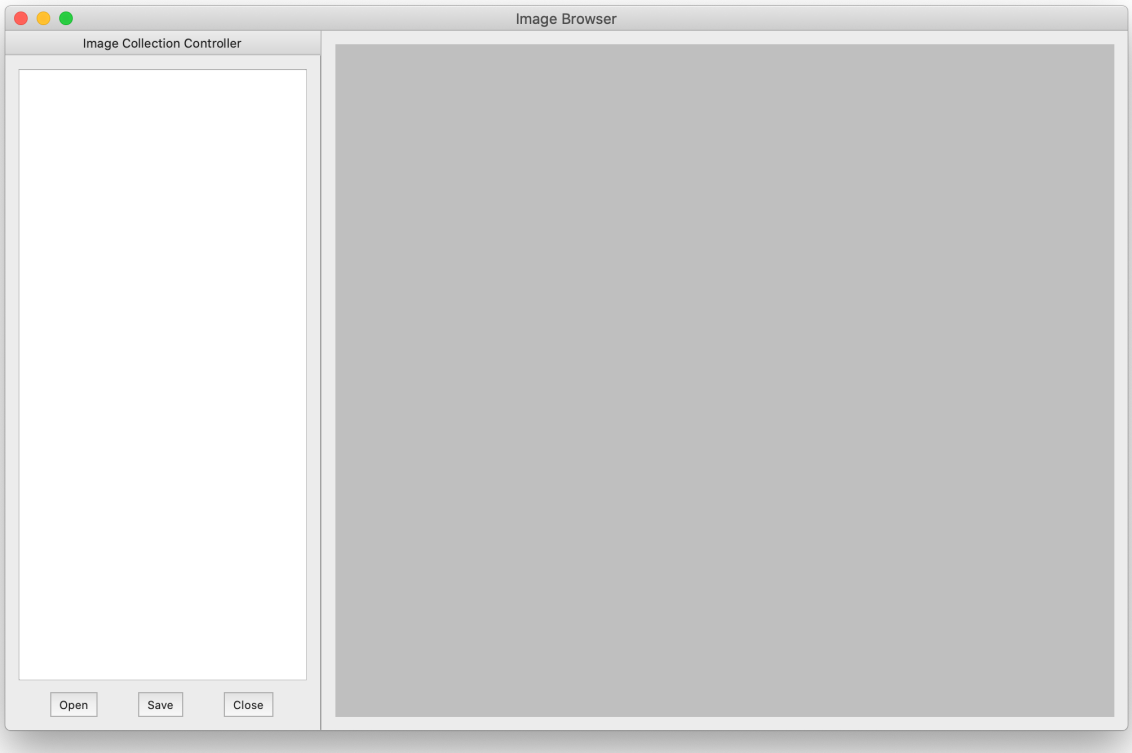
Usage

Launch VideoLab

To launch video lab, you need to open a terminal, activate the conda environment you deployed, and run the launch program:

- conda activate your_env_name
- cd to the root folder of the repository
- python ImageBrowser.py

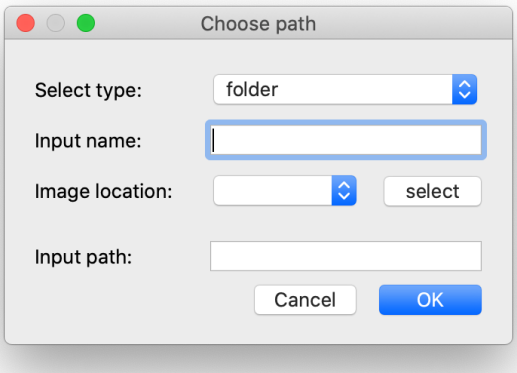
You will see a window like the following image:



The window has two parts: the left part is a dock widget which shows the image series you have opened. The right part is the area where you browse the image series. You can drag the left part to wherever you like or make it thinner or wider.

Open an image series

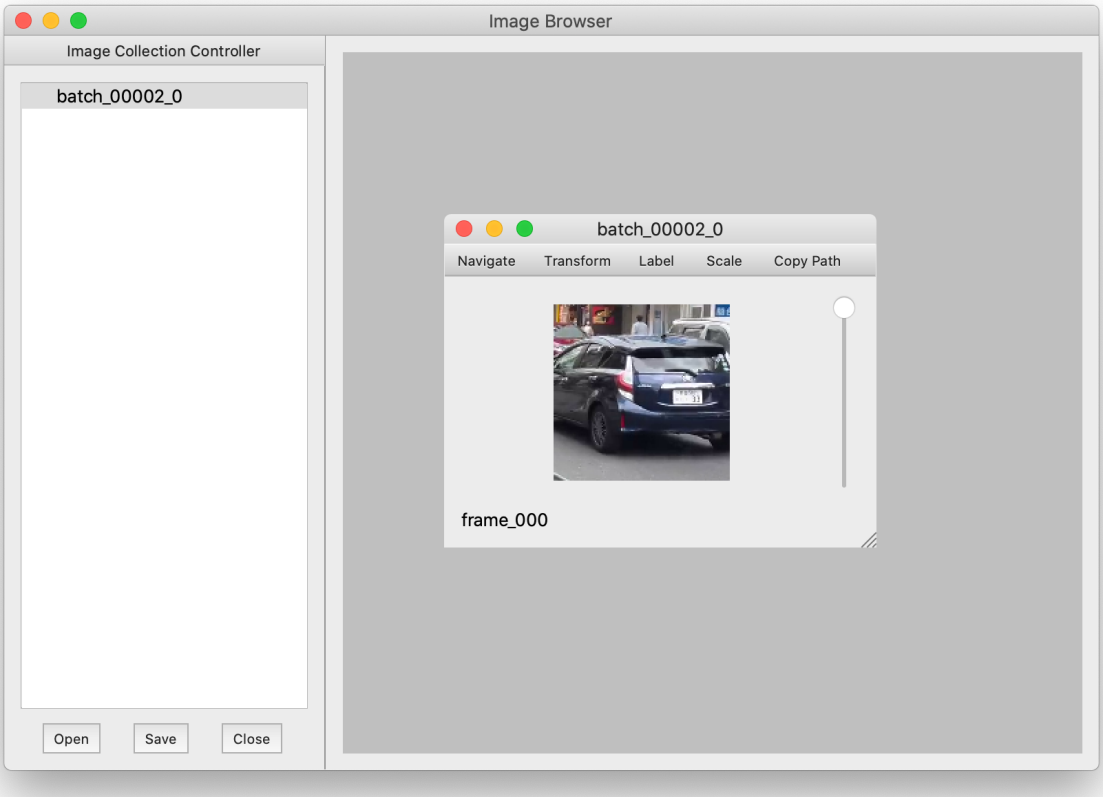
To open an image series, click the open button on the left bottom corner of the dock widget. You will see a dialog like this:



Select the format of the image series you want to open, input a unique nickname of the image series, which will be used to refer to the image series you opened, and type in the path of the image series (or use the file dialog to select a path by clicking the “select” button). For paths from the server, you have two choices: One is to leave Image location empty and input a full path in the server. The full path should be in scp format (e.g. usr@host:path). Another choice is to select the nickname of the server configuration from Image location and input the path of images to Input path field without the need of pointing out the username and the host. Currently, we support three basic types: folder, video, and ppm. Folder means a folder of images. If you select “folder” as the type, the corresponding path should be a folder which contains at least one image. We support videos of mp4 and avi format for now, so the path should be a video file of those two formats. PPM refers to super PPM format, which is a special format we designed for extending the original PPM format. Super PPM format can store and show a series of images in a compact way. The standalone super PPM format can also be opened like a normal PPM file. The corresponding path should be a file of ppm format if we select “ppm” as the type.

Besides the three basic types, we also support two compound types: optical flow and bbox. Basic types are RGB based, which means all the underlying data are RGB images. Compound types supports more complicated data structure. For example, you can visualize a series of optical flow files (.flo) from a folder with optical flow type. Bbox is a type for the result of object tracking which includes both RGB images and the bounding box associated to these images.

Clicking OK button, the image series will be opened to the dock widget like the following screenshot (VideoLab may ask you for the password of the server when the input path is on the server end). Double click the nickname of the image series, you can see a subwindow is opened for you to display the image series.

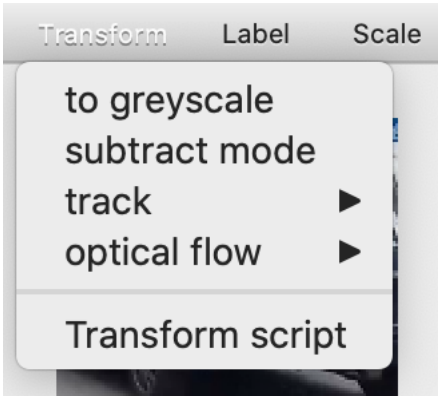


Browse the image series

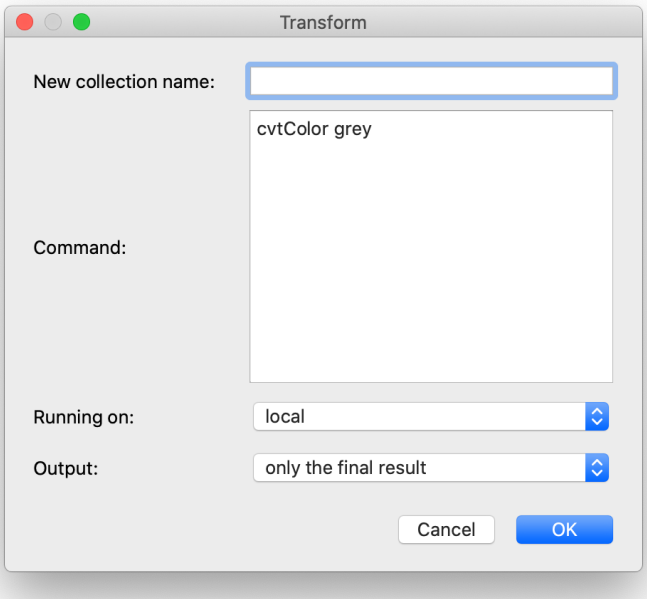
You can drag the slider of the sub to the right of the image to browse forward and backward rapidly. The left bottom corner shows the name of the image. There is a toolbox under the title of the subwindow. “Navigate” button can give you better control of browsering the image series, “Transform” button will help you process the image series you opened. You can refer to Process images subsection for details. “Label” button will pop up a menu, which can help you label some images of interest and browse the labeled subset of images in a separate window and save them to a separate path. Please refer to Label Images subsection for more detail. “Scale” button can help you zoom in/out the image you are viewing. “Copy Path” button will copy the path of the currectly displayed image to your clipboard.

Process images

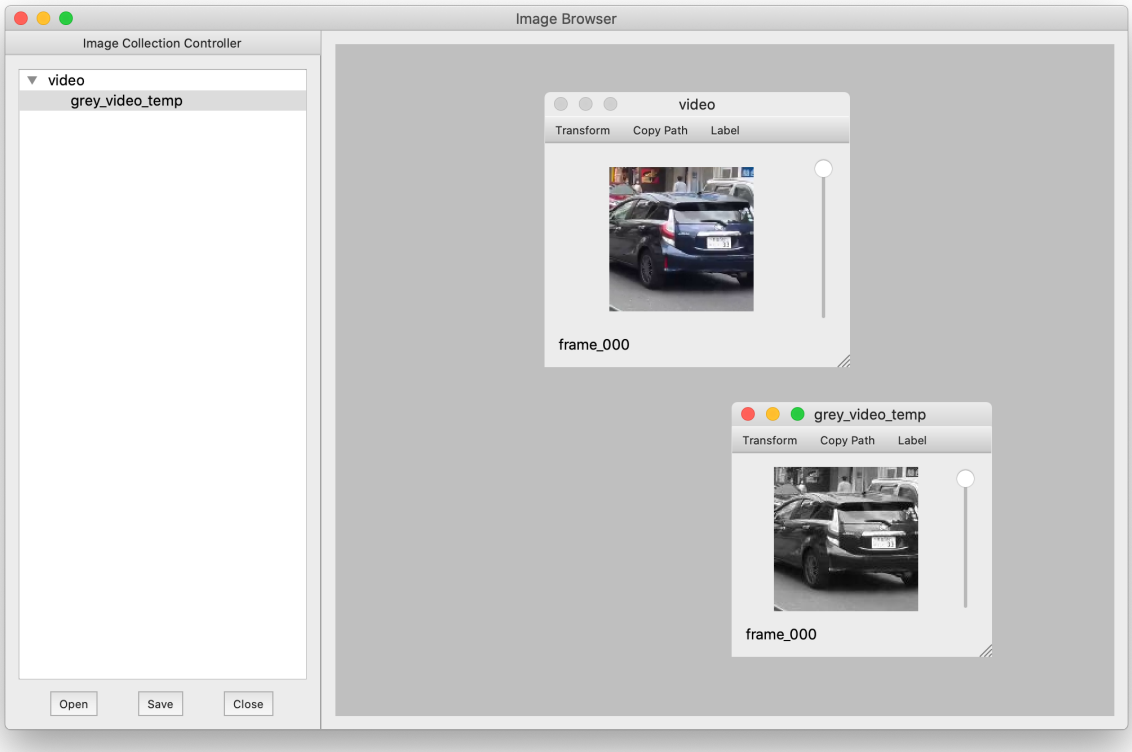
Click “Transform” button on the subwindow, you can see the following menu:



The items above the separator are preset image processing functions, these are common image functions like converting images to grey or computing the optical flows. Click one of it, you will see the following window popped out:



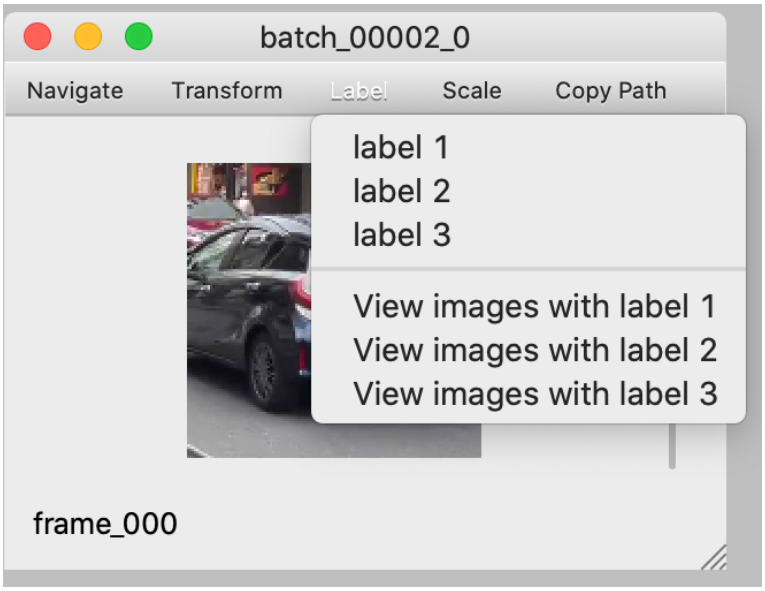
All the functions are script-based. The preset will type in the corresponding script for you automatically. If you choose “Transform script”, the command field will be empty and wait for your input. Type in a script you want to use for image processing, and the nickname for the new image series which will be generated after processing, and click OK button, VideoLab will run the script to process the image series and open the new image series as a sub-series of the original image series:



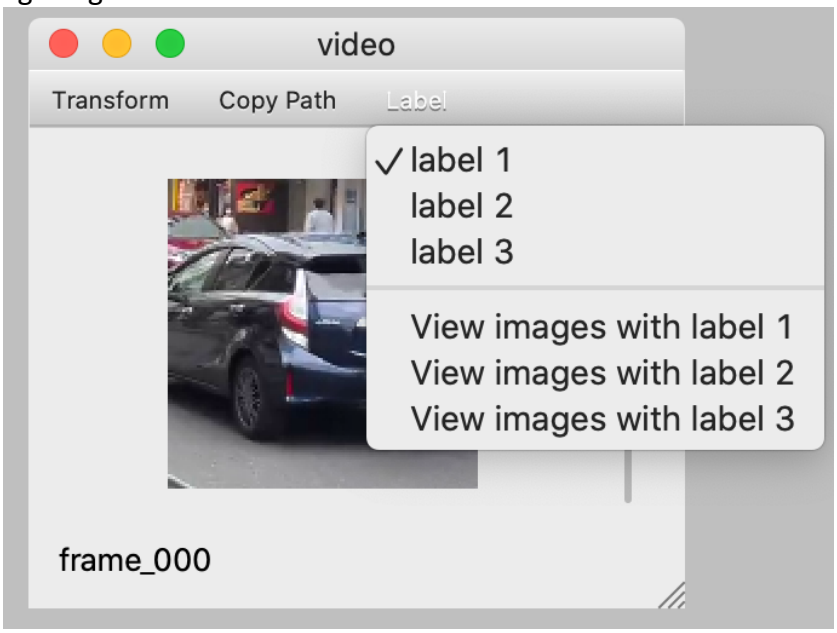
You can also select the machine you want to use to run the script by clicking the combo box next to Running on, and select the nickname of the server, which refers to the server you registered to the software in registeredServerConfigs.json. “local” means running on your local machine. You can also select to output all intermediate results of every command in the script with Output combo box.
Notice that we added “_temp” to the nickname, meaning this newly generated image series are stored in a temporary folder. You need to save it to avoid losing it after you close VideoLab. Please refer to Save Image Series subsection for more detail.
The grammar of the script VideoLab supports will be described in Scripts section.

Label images

By clicking Label button on the subwindow, a menu like the following image will pop out.



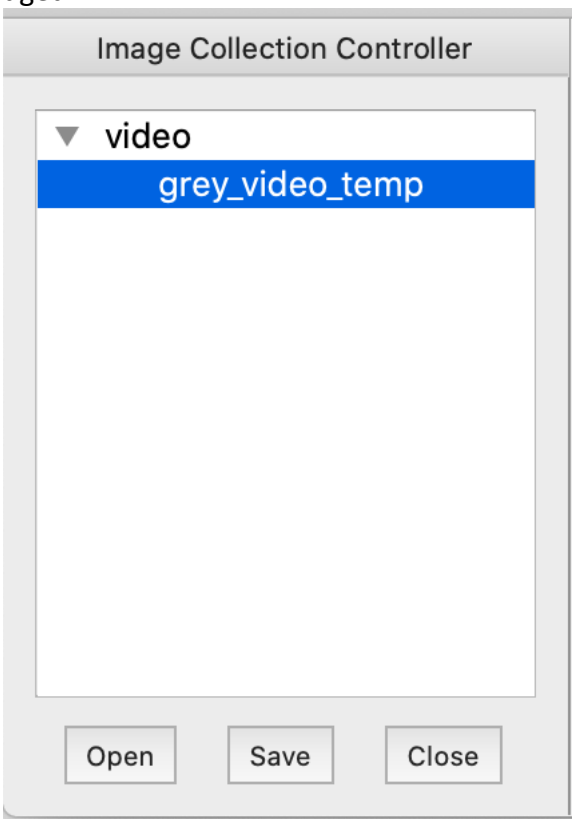
By selecting “label x” in the menu, the currently displayed image will be added to the image subseries of “label x”, and you will see a tick on the item if the current image has been added to the corresponding image subseries:



By selecting “View images with label x”, a new subwindow will be opened to show all images with label x. You can save this set of images by clicking “Save” button under the window title. Please refer to Save image series subsection for details.

Save image series

You can save any image series you opened to another path in another format. To do this, you need to select an image series by clicking it in the dock widget, and then click Save button on the bottom of the dock widget.



A dialog will pop up. You should select the format you want to save the selected image series, input the name for it and input the path you want to save it.

If the type of the image series you want to save is a basic type, you can select any other basic types, our software will convert the format for you. If the type is a compound type, you may

also select basic types for saving. However, this will only keep the RGB component and discard other information. We suggest using the same type for saving in this case.
Notice that, if the selected type is folder or any folder-like compound type (e.g. optical flow and bbox), the input name is the folder name the image series will be saved, and the input path is the path where the newly created folder will be placed. If the selected type is a file like video or ppm, the input name must include the extension (e.g. abc.mp4), and the input path is the folder path where the newly created file (i.e. abc.mp4) will be placed.
You can type in a path located in a server to save the image series to the server directly. In this case, you need to type in a path like "sam@gypsum.cs.umass.edu:~/images.mp4", which is the same as the scp command's format. VideoLab may ask you for the password for the server.
Also, you can use image location to select a server configuration and save your time typing your username and the server address like what you did during opening an image series.
By the way, it is a little different to save an image sub series with labels since those subwindows will not be registered to the dock widget. Please refer to Label images subsection.

Close image series

Selecting an image series by clicking its nickname in the dock widget and clicking Close button in the bottom of the widget, you can close the image series you have opened. Notice that, the image series and its associated sub series will all be closed together.

Scripts

To process images in VideoLab, you can just type in a script with some simple commands without the need of writing an individual program. We provide many off-the-shelf commands which should suffice the common use. You can also easily design your own command and extend this powerful tool to suit your need. We will introduce how you can achieve this in section Advanced Topics.

Commands

A script is formed by several lines of commands. VideoLab will parse the script and run it as a pipeline. The basic grammar of each command is like the following format:

command args

For example, in "cvtColor grey", cvtColor is the command part, and grey is the args part. These two parts are separated by a space.

The args part is a string of the format supported by the built-in argparse module in python. VideoLab uses argparse to parse the arguments input by users. Please refer to https://docs.python.org/3/library/argparse.html for the use of argparse. Generally speaking, args can be something like:

--color grey --shape rectangle –workers 2

This is pretty like what we daily use in Linux, so the learning cost is minimized.

Of course, different commands need different arguments. The following table lists some off-the-shelf commands VideoLab currently supports and their associate information. You can also define your own command as I mentioned.

Command	**Arguments**	**Function**
pwcnet		Compute optical flow with PWC-NET
	Example: pwcnet	
	Source: https://github.com/sniklaus/pytorch-pwc	
	Extra packages: pytorch, cupy(pip install cupy-cuda111), flow_viz(pip install flow_viz)	
cvtColor	grey	Convert the image series to greyscale
	Example: cvtColor grey	
raft	see raft github repo	Compute optical flow with RAFT
	Example: raft	
	Source: https://github.com/princeton-vl/RAFT	
	Extra packages: pytorch, scipy	
stabilizedTrack	--bbox x y w h	OpenCV stabilized tracker
	Example: stabilizedTrack --bbox 100 80 100 80 x and y are the index of the left top corner of the bounding box. w and h are the width and the height of the bounding box, respectively.	

Another thing you need to notice is that different commands have different input/output type. For example, raft takes an image series of basic types as input and outputs an image series of optical flow types. If the type is not correct, the processing of image series will fail. We also designed some commands like bbox2RGB for type conversion.

Macros

Sometimes, we need to interact with users to make an image processing function work. For example, a tracker needs an initial bounding box selected by the user. If you work only on your local machine, it is fine to implement the interactive logic directly in your own command (see section Advanced Topic for the detail of how to implement your own command). However, if you want your command also work on a server, the interactive logic will no longer function since a server usually does not support GUI. To circumvent this problem, we introduce macros to the VideoLab script.

A macro is a special line of command which will only run on your local machine. The format of a macro is like this:

#define macro_name interact_command

interact_command is a special command which only runs on your local machine. You can use these commands to interact with your user, like letting them select a bounding box from an image. We provide a list of interact command but you also can design your own. Interact command will return a string indicating the user's choice.

The macro above defines a special symbol named macro_name. This symbol is bound with the string returned by interact_command. In the args of ordinary commands, [macro_name] will all be replaced by the string returned by interact_command.

Let's use an example to explain this:

#define bbox selectROI
stabilizedTrack --bbox [bbox]

The first line, a macro, defined a symbol named bbox. selectROI is an interact command which helps users select an initial bounding box and return "x y w h" as a string. Thus, bbox is now bound with the string "x y w h" which describes the bounding box users selected. In the second line, an ordinary command, we will replace [bbox] by the string. Then, VideoLab will remove all macros and run the rest of the script locally or remotely.

Notice that, selectROI only runs on your local machine, [bbox] is replaced by the true parameters. The server can run the ordinary command without disturbing the users. We will discuss about how you can design your own interact command in section Advanced Topics.

Advanced Topics

Create your own command

You can easily design any command you need for VideoLab. Basically, you just need to write a python class derived from some parent class depending on the output type of your command, implement the abstract methods, and then register the class to utils/transformers/registeredTransformers.json. Let's use two situations to describe the implementation:

The first situation is for commands which will produce models of basic types (folder, video, and super ppm). We use Transform_cvtColor as an example. This class is defined in utils/transformers/Transform_cvtColor.py. As you see, it derives from Transform_toBasicModel class. Every command whose output is a basic model should derive from this class. Let's take a look at its component: It has a class variable named command. You need also define this variable in your class and assign it with a string, which refers to the command you want to use. Three other methods are also defined. __init__ should not have any argument besides self. The main work for this method is to call the parent's __init__ method and initialize your own code. The other two methods, getArgParser and processImageCollection, are the abstract methods you need to implement.

- getArgParser takes no argument aside from self but outputs a parser from argparse module. In this method, we just use argparse to customize our own parser, which will be used to parse the args part of the command. If you don't need any argument for your command, just return None.
- processImageCollection is the core of the command. It processes the image and output the result. It takes three arguments, i.e. self, model, and args.
  model is a special type to store image series in VideoLab. You don't need to know the details of the type. The only thing you need to know for now is that it has five essential methods: **model.length()**, **model.getData(idx)**, **model.getImg(idx)**, **model.getImgName(idx)** and **model.get(idx)**:
  - length() will give you an integer indicating the number images in the image series to be processed.
  - getData(idx) will give you the underlying representation of the i-th image in the model. For basic models, this returns what getImg(idx) returns. For compound models, the data returns depends on the specific model type. For example, optical flow model will return the flow; bbox model will return a tuple including the RGB image and its associated bounding box (x, y, w, h), where (x, y) is the left corner, x and y are for width and height, respectively.
  - getImg(idx) will give you the RGB representation of the i-th image. The image is an **uint8** numpy array. The image is of shape **(Height, Width, Channel)**, where channel is of the sequence **RGB**. Each compound type has defined its own way to convert its underlying representation to a RGB image. For example, optical flow model returns the visualization of the flow. Bbox model returns the image with the bounding box.
  - getImgName(idx) returns the name of the i-th image.
  - get(idx) returns a tuple combining the result of getImg and getImgName

Args is the result parsed by the parser you returned from getArgParser with the args part of the command as the input. Thus you can use something like args.target to get the user's input. Don't forget to examine if the input is valid and raise an exception if needed.

The final thing you need to pay attention to is that we return images and its associated name as a tuple by **yield**, instead of return. The image should be of the same format as the input. You can change its name if you'd like.

The second situation is to create a command which produces compound types. In this case, you need to derive from the corresponding base class. For commands with optical flow output, use Transform_toOpticalFlowModel; for commands with bbox output, use Transform_toBboxModel. As you can see from these two classes, there embodies another abstract method named generateProcessModel. This method collects result from processImageCollection method, and return the produced output model. You don't need to do anything here since the model creator will take care of this method. If you take a look at Transform_toBasicModel, you'll see the same method also defined there.

Let's describe how to create a command for optical flow model output as an example. As you can see from utils/transformers/optical_flow folder, there are two base classes, Transform_opticalFlowBase and Transform_opticalFlowFramework. The latter derives from the former. You have two choices here. One is to create the former class and build the command from scratch. Another is to use the framework we provide, which can save you from writing some redundant code. We highly recommend the second choice and will only introduce the second choice.

Open Transform_Farneback as an example. There are four methods and one field, which you need to define. Command and __init__ are the same as commands for basic models. The other two methods are initOpticalFlowAlgorithm(self, model, args) and computeOpticalFlow(self, img1, img1_name, img2, img2_name). The first method is for model initialization or other preparation. The two arguments are the same as described before. The second method is the core method which inputs the images and their associated name (all are the same format as described above), and returns the flow and the new name for this flow. You can open Transform_opticalFlowFramework to take a look at how its processImageCollection method use these two methods you defined.

Creating a command for bbox model is very similar. Open Transform_trackerFramework, you'll see two abstract methods waiting for implementation. You need to derive this class and implement them besides command and __init__. initTracker is for model initialization; update gets a new image, and compare it with the images previously input to it, and output the bounding box (x, y, w, h) associated with this image.

After you implemented your own class, don't forget to register this class to registeredTransformers.json. The format of the item to add is *"command": [module path, class name]*.

Interact with users

Sometimes we need to interact with users before we process an image series. For example, a tracker may need its user to select an initial bounding box for it. You have two choices to achieve this interactive function.

You can incorporate interactive logics directly to your transformers, i.e. the image processing module. This is the easiest and conveniently way of interacting with users. However, this will limit your transformers to running only on your local machine (MacOS only) since a server usually doesn't have GUI.
There is only one thing you need to notice: in transformers, the interactive logics must be run in a sub process. This is because transformers are running in a sub thread, which should not do any UI related work. You can refer to Transform_stabilizedTrack as an example.
Another way to interact with users is to use interact commands and macros. You can design your own interact commands. The steps are pretty similar to the way you create an ordinary command. What you need to do is to create a class derived from utils/interactTools/Interact_base, implement the abstract method interact, and register your class to utils/interactTools/registeredInteractCmds.json. The abstract method only takes one argument, model, besides self. The interface of model is the same as what we introduced in the previous subsection.
One pitfall is that OpenCV's GUI tools cannot work with Qt in Linux environment. Thus, in the implementation of the method interact, you need to create an independent program, use os.system to fork a new terminal and run the independent program. Please refer to utils/interactTools/interact_selectROI.py.

Use your own server

VideoLab is designed to be flexible enough to communicate with your server freely. To use your own server, you need to create your own server configuration file, and change configs/serverConfigs/registeredServerConfig.json to point to the configuration file you wrote.
To create your own server configuration file, you can use configs/serverConfigs/mygypsum.json as a template. You need to change filed server, username, auth_method, processor_path to your corresponding server as described in section Installation. If your auth_method is "key", you also need to add "key_file" field to refer to the private key file for your ssh login. We described these fields in section Installation. The only field we haven't explored is template_path.
You need to create your own template file and change this field to point to the file. The template file writes all commands you need to run if you want to run a program.
Let's use gypsum_template.txt as an example. On Gypsum (a slurm-based server) we need to apply for a new node for computation instead of running programs on the main node, we input srun to apply for a new node. Then, we activate our deployed conda environment. The third and the fourth lines are special symbols which will be replaced automatically by VideoLab. [GENSCRIPT] will be replaced by a command which generates the script VideoLab will run for image processing. [RUN] will be replaced by a command which run the generated script. Finally, we need to type in exit to release the computing node we are using.
You need to customize your own template file to adapt to your own server. The template file is just like the way you operate your server daily. You only need to add [GENSCRIPT] and [RUN] to the template file.