Vicente Farias
Ulas Onat Alakent

C++ OpenCV Image & Video Processing Library Utility

1) Introduction & Philosophy **2**

2) Setup **2**

3) Tutorial & User Documentation **3**

4) Developer Documentation **6**

4.1) Image Processing 7

4.1.1) Introduction 7

4.1.2) Design Choices in Image Processing 7

4.1.3) Functions 7

4.2) Video Processing 10

4.2.1) Introduction 10

4.2.2) Design Choices in Video Processing 10

4.2.3) Functions 11

4.3) User Interface 12

4.3.1) Introduction 12

4.3.2) Design Choices in User Interface 13

4.3.3) Functions 14

5) Benchmarking **16**

6) References **18**

1) Introduction & Philosophy

The goal of our project was to provide a simple and convenient way to perform various image and video processing operations using C++ and the OpenCV library. Image and video processing software is not readily or openly available. The most ubiquitous image processing software, Photoshop, costs about $240 dollars per year with no option for a lifetime subscription.

In order to complete our goal of producing image & video processing software, we hoped to create Image and Video modules that would provide useful functionalities through export statements. We had issues with module declaration with our compilers in development and were ultimately unable to implement them. Nevertheless, we were still able to create the Image and Video classes and import them to demonstrate our underlying philosophy and functionality.

Some of the reasons as to why we decided to implement this project:

1. **User-friendly:** Image and video processing can be complex and require specialized knowledge and skills. Our project aims to make these operations more accessible and user-friendly for a wider audience by simplifying input through a UI. We also reduced function arguments when importing our library to reduce functions to simple but useful use-cases (examples seen in user manual).
2. **Customizability:** With our project, users can customize and extend the functionalities of the Image and Video classes to suit their specific needs and requirements. This allows for greater flexibility and adaptability in image and video processing. Our ultimate vision included extended customizability through custom image labeling and custom training of the detection model. Room also lies for additional function implementations.
3. **Performance**: Given that OpenCV is available in both Python and C++, we were curious about testing the performance of our code across languages. We were aware that OpenCV's Python library served as a wrapper for the OpenCV C++ library, but still wanted to test the performance differences between the OpenCV Python Library and our OpenCV C++ Library.

2) Setup

*Note: The following instructions were completed on a macbook m1 macOS Monterey 12.3.1

1) Download a C++20 Compiler
2) Download CMake at https://cmake.org/install/
3) Download and Install the latest version of OpenCV for your respective machine from https://opencv.org/releases/
4) Extract and move the opencv to the proper library directory, in our case:
   /opt/homebrew/Cellar/opencv/4.6.0_1/lib/pkgconfig/opencv4.pc
   Alternatively 3-4 (on macOS):
   Install homebrew from https://brew.sh/
   Run: brew install opencv

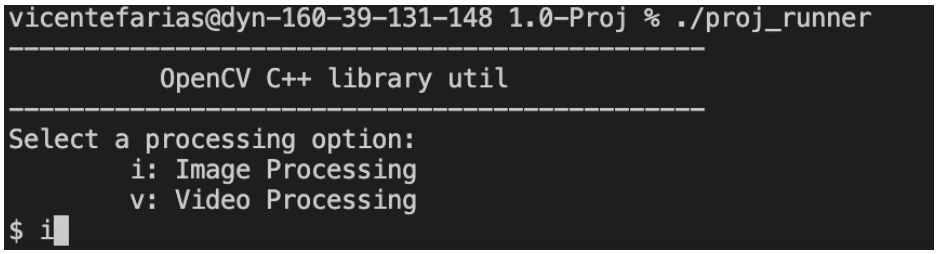
Check that brew has been installed at
/opt/homebrew/Cellar/opencv/4.6.0_1/lib/pkgconfig/opencv4.pc

5) Extract our project from the Github repository or a zip file.
   https://github.com/vicentefarias/4995_project/tree/Final
6) Change into the project directory
7) Run: Make from the project directory and ensure the code compiles (may require checking library path directory in Makefile)

3) Tutorial & User Documentation

The following section provides a guide to utilizing our Image & Video processing library, with our UI as an example use-case. The UI serves as a simple CLI which loads in videos or image files into their respective objects. The UI then provides intuitive access to functionalities by associating each function with input from std::in.

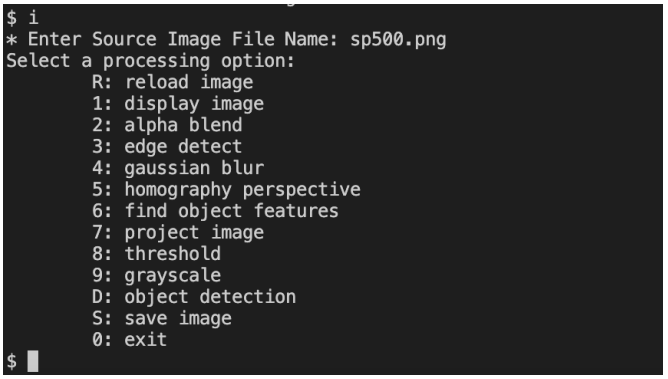
Running make from the project directory should now output an executable file, proj_runner. Execute Project Runner: ./proj_runner



The main menu of our user interface includes two options for image & video processing. Entering a processing option, char i or v, displays all possible functionalities for the option's respective class.

Image Processing

Selecting Image Processing in the main menu outputs all image processing techniques. The full list of image processing functionalities is listed below. Note that executing a function loads the Image output (executing a function loads the output on the next iteration). This serves to stream or pipe outputs of one function to another.



All image processing options are associated with a sample functional implementation of our Image class. Explanations of some associated function implementations follow below.

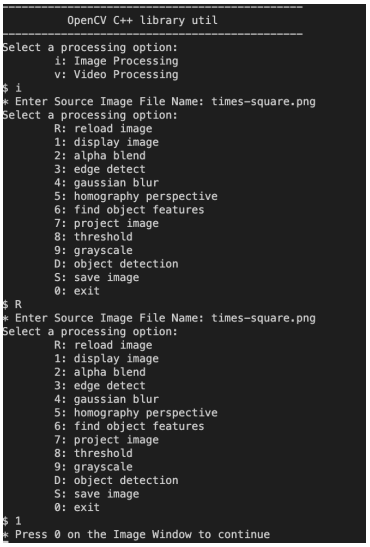
Reload Image: Calls the function get_image_from_user, which initializes and returns an image object from a filename. The function is used to load an image file into working memory, which can then be used to cascade outputs from various functions using the UI.

```
static Image get_image_from_user(const std::string& name = "") {
    if (name.empty())
        cout << "* Enter Source Image File Name: ";
    else
        cout << "* Enter Source " << name << " Image File Name: ";

    std::string img_filename;
    cin >> img_filename;

    return Image(img_filename);
```

Display Image: Calls the currently loaded Image object's built-in show() function. The display image case simply calls Image.show() on the Image object currently loaded into memory.
Sample Output:



Alpha Blend: calls the alpha_blend method which takes an Image object as argument and loads a second Image object from the user by get_image_from_user function. A double other_weight is obtained from the user, which denotes the relative pixel weight of the other image. The built-in alpha_blend method is called from the input Image object, which takes 2 parameters as arguments (as opposed to 4 in the original opencv library).

```
static Image alpha_blend(const Image& img_in) {
    Image other_img = get_image_from_user("image to blend");
    double other_weight = get_val_from_user<double>("weight for other img");

    other_img.fit_to_size(img_in);
    return img_in.alpha_blend(other_img, other_weight);
}
```

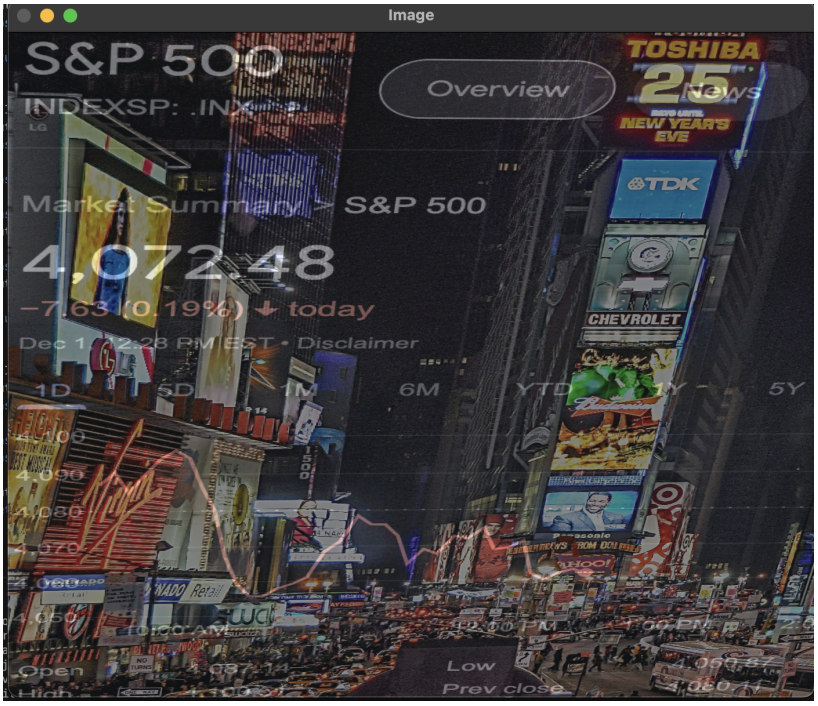
Sample Output for blend with sp500.png and 0.5 weight:

Homography Perspective: this case calls the homography_perspective function, which involves applying 2 built-in Image class functions. The first function provides an intuitive UI by collecting a set of input points from the user through clicks on a window. The second function shifts the perspective of the image onto the boundary enclosed by the four selected points.

```
static Image homography_perspective(const Image& img_in) {
    cout << "* Enter 4 points and continue\n";
    const auto pts = img_in.collect_points();

    return img_in.create_homography(pts);
}
```

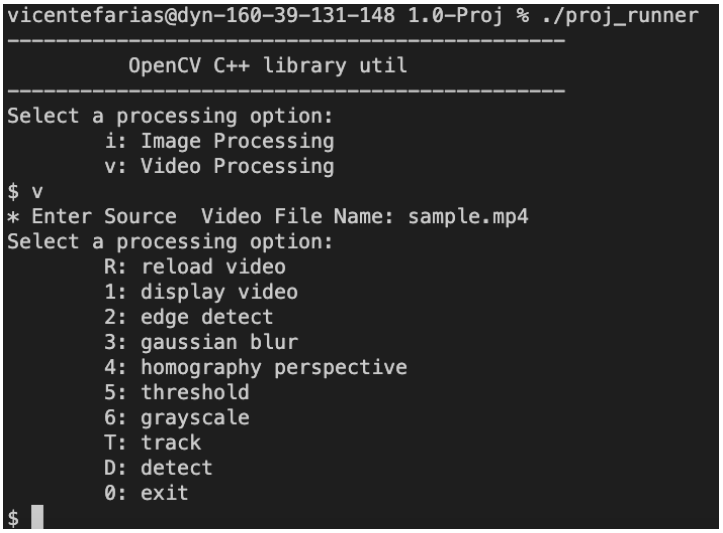
Sample output, after clicking on the TopLeft, TopMiddle, BottomMiddle, and BottomLeft points on the output of the previously alpha blended image:



As seen through the examples above, our Image class provides simple and intuitive ways to process images. The homography_perspective example abstracts a layer away from mouse_call_back and pointer_casts when collecting points. This idea of abstracting a layer of complexity drove our UI implementations as well as implementations of the Image & Video classes. For more examples on utilizing the Image class refer to 4) Developer Documentation.

Video Processing

Selecting Video Processing in the main menu outputs all video processing techniques. The full list of video processing functionalities is listed below. The video processing options work much the same as those for Images, in that they create new Video objects from their associated function. A sample implementation of a Video object function is shown below.



Threshold case calls a function, threshold_mask, that takes a Video object as input and returns another Video object. The Video input, cap_in, calls a built-in two argument threshold function from the type and thresholding value obtained from the user. The function thresholds an images pixels based on the type and threshold value.

```
static Video threshold_mask(Video& cap_in) {

    threshold_options();
    const int type = get_val_from_user<int>("threshold type");
    const int val = get_val_from_user<int>("threshold value");

    return cap_in.threshold(type, val);
}
```

Sample output frame for type=1, val=100:

4) Developer Documentation

We implemented an image and video library that can be imported (ideally module import) and utilized the implementation to produce common image and video processing functions. Image.h and Video.h describe the implementation of the Video and Image Classes, which contain the underlying processing functionalities.

4.1) Image Processing

4.1.1) Introduction

Image processing is done through the Image class we created, which is a wrapper around the OpenCV cv::Mat class that provides some basic image processing functionality. It includes methods for loading an image from a file, displaying an image on the screen, saving an image to a file, resizing an image to match the size of another image, blending two images together, detecting edges in an image, blurring an image using a Gaussian blur, and collecting user-defined points from an image to project to a new image. These methods can be used to perform various image processing tasks using the OpenCV library.

4.1.2) Design Choices in Image Processing

A design choice in this implementation is the use of a struct to represent errors that can be thrown by the Image class. In this case, the NotEnoughPointsErr and FailedToLoadImgErr structs are used to indicate that not enough points were collected from the user, or that the image failed to load from a specified file, respectively. This allows for a more structured approach to error handling, as the structs can be caught and handled separately in the code.

Apart from that, we use the debug_assert function, which is used to check for certain conditions and raise an error if those conditions are not met. This function is used, for example, to ensure that the kernel size passed to the gaussian_blur method is an odd number, greater than 1, and less than 1000. This type of checking can help prevent errors and ensure that the code behaves as expected.

Also note that we use a static method, destroy_all_windows, for closing all currently displayed image windows. This method can be called without an instance of the Image class, making it convenient to use when working with multiple Image objects.

Finally, the collect_points method uses a callback function, add_point_cb, to handle mouse clicks and add the selected points to a vector. This allows for a clean separation of concerns between the main body of the collect_points method, which displays the image and waits for user input, and the callback function, which handles the actual collection of points. This can make the code easier to read and maintain.

4.1.3) Functions

- **Image():** The default constructor for the Image class. It initializes an empty Image object.
- **Image(const std::string& filename):** This constructor takes a filename as input and loads the image from that file using OpenCV's imread function. If the image cannot be loaded, it throws a FailedToLoadImgErr error.
- **Image(const cv::Mat _img):** This constructor takes a cv::Mat object as input and initializes an Image object with the given cv::Mat object.
- **void show(const std::string& filename = "Image"):** This method displays the image on the screen using OpenCV's namedWindow and imshow functions. It also waits for the user to press the "0" key on the keyboard before closing the window.
- **void save(const std::string filename):** This method saves the image to a file using OpenCV's imwrite function.
- **static void destroy_all_windows():** This method destroys all currently displayed windows using OpenCV's destroyAllWindows function.
- **void fit_to_size(const Image& other):** This method resizes the current image to match the size of another image. It does this by calling OpenCV's resize function, passing in the current image, other image, and size of the other image.
- **Image alpha_blend(const Image& other, const double other_weight):** This method blends the current image with another image at a specified weight. It does this by calling OpenCV's addWeighted function, passing in the current image, the specified weight for the current image, the other image, and other image weight = (1 - current image weight). The design reason for selecting the other image weight as (1 - current image weight) is to not oversaturate pixels in the resulting blending image. The result of the blending operation is returned as a new Image object.
- **Image edge_detect(const int lower_threshold, const int upper_threshold):** This method detects edges in the image using the Canny edge detection algorithm. It does this by calling OpenCV's Canny function, passing in the current image, the lower and upper thresholds for the edge detection, and the output image. The output image is returned as a new Image object.
- **Image gaussian_blur(const int kernel_sz):** This method blurs the image using a Gaussian blur. It does this by calling OpenCV's GaussianBlur function, passing in the current image, the kernel size for the blur, and the output image. The kernel size must be an odd number greater than 1 and less than 1000. The output image is returned as a new Image object.
- **std::vector<cv::Point> collect_points(const std::string& window_name = "_tmp_collect"):** This method allows the user to collect points from the image by displaying the image on the screen and allowing the user to click on points of interest. It does this by calling OpenCV's setMouseCallback function, which sets a callback function that is called whenever the user clicks on the image. The callback function collects the clicked points and stores them in a vector. Once the specified number of points have been collected, the image is closed and the vector of points is returned.

- **Image create_homography(const std::vector<cv::Point>& points):** This method computes a homography that can be used to align or stitch images together. It takes a vector of points as input, which defines a subimage of the current image. It then uses OpenCV's findHomography function to compute a homography that maps the points in the subimage to a destination image of the same size as the current image. The destination image is defined by four points: (0, 0), (image width, 0), (image width, image height), and (0, image height). The homography is then applied to the current image using OpenCV's warpPerspective function, which produces a new image that is aligned with the destination image. The resulting image is returned as a new Image object.
- **int get_0th_moment(const std::vector<cv::Point>& points):** This method computes the 0th moment of a set of points in the image. The 0th moment is simply the area of the convex polygon defined by the points. It does this by creating a binary mask of the image, where all pixels outside the convex polygon defined by the points are set to 0 and all pixels inside the polygon are set to 1. It then uses OpenCV's countNonZero function to count the number of non-zero pixels in the mask, which is equivalent to the area of the convex polygon. The area is returned as an integer value.
- **static cv::Point get_1st_moment(const std::vector<cv::Point>& points):** This method computes the first moment of a set of points in the image. The first moment is the mean (or average) position of the points. It does this by summing the x and y coordinates of all the points, and then dividing by the total number of points to compute the mean x and y coordinates. The mean coordinates are returned as a cv::Point
- **Image get_mask(const std::vector<cv::Point>& points):** This function creates a binary image mask based on the specified points. This mask is a white polygon with the same shape as the specified points, with black pixels outside of the polygon. The function then combines this mask with the original image using a bitwise AND operation, which sets the pixels outside of the mask to black. This effectively crops the image to the region defined by the mask.
- **Image proj_img(const Image& other, const std::vector<cv::Point>& other_points, const std::vector<cv::Point>& this_points):** This function projects a portion of one image (the other image) onto a portion of this image. This is done by first masking out the projection site on the destination image (this image), then masking out the rest of the source image and projecting the masked source image onto the masked destination image using a perspective transformation. The resulting image is the combination of the two masked images.
- **Image grayscale():** The function converts this image to grayscale using the OpenCV cvtColor function. It then converts the grayscale image back to an RGB image, which is necessary because OpenCV stores grayscale images as single-channel images, while the Image class stores images as 3-channel RGB images.
- **Image threshold(const int type, const int value):** This applies a thresholding operation to this image, which sets all pixels below a certain value to black and all pixels above the value to white. The type of thresholding to use and the threshold value are specified as arguments to the function. This can be useful for image segmentation, where the goal is to separate different objects in an image based on their intensity values.
- **std::vector<std::string> loadClassifications():** This function reads a list of class names from a file on disk and returns them as a std::vector<std::string>. This is used to associate the class names with the predictions of a trained object detection model, so that the class names can be displayed along with the predicted bounding boxes.
- **cv::dnn::Net loadDNN():** This function loads a pre-trained object detection model from a file on disk using the OpenCV readNet function. It then sets the preferred backend and target for the model, which specifies which hardware and software should be used to run the model. This function returns the loaded model as a cv::dnn::Net object, which can be used to perform object detection on images.
- **cv::Mat formatInput(cv::Mat &image):** This function resizes an image to have the same width and height, with any additional pixels set to black. This is used to ensure that the input

images are of a consistent size that is compatible with the trained object detection model. The function returns the resized image as a cv::Mat object.

- **std::vector<Detection> obtainOutput(cv::dnn::Net &net, cv::Mat &input, const std::vector<std::string> labels):** This function performs object detection on an input image using a pre-trained object detection model. It first uses the blobFromImage function from OpenCV's deep learning module to convert the input image into a format that can be used as input to the model. It then runs the model on the input image, extracts the predicted bounding boxes, scores, and class labels from the model's output, and applies non-maximal suppression to filter out overlapping bounding boxes. The function returns the resulting detections as a std::vector<Detection> object, where Detection is a struct containing the bounding box coordinates, the class label, and the confidence score for each detected object.
- **void putDetection(std::vector<Detection> &output, cv::Mat &frame, const std::vector<std::string> labels):** This function draws the bounding boxes and labels for the detected objects on the input image. It iterates over each detection in the input output vector, draws a rectangle around the bounding box, and adds the class label and confidence score to the image. It then returns the resulting image with the detections drawn on it.
- **Image detection():** This function performs object detection on the current image using a pre-trained object detection model. It first loads the class labels and the model itself using the loadClassifications and loadDNN functions. It then formats the input image using the formatInput function, and performs object detection on the input image using the obtainOutput function. Finally, it draws the bounding boxes and labels for the detected objects on the input image using the putDetection function, and returns the resulting image.

4.2) Video Processing

4.2.1) Introduction

Video processing is done through the Video class that is designed to make working with videos in C++ easier by providing a convenient interface for common video processing tasks. The class serves as a wrapper around the cv::VideoCapture class from the OpenCV library.
When a Video object is created, it is initialized with either a filename or a cv::VideoCapture object much like an Image class is initialized from a file or cv::Mat. The Video class provides several methods for processing and displaying the video, such as show for displaying the video, tracking to set consistent bounding boxes on a labeled object, and edge_detect for detecting edges in the video. As examples, the show method can be used to iterate and display a video's frames in a window, and the grayscale method can be used to convert the video to grayscale and save the result to a new video file. These methods make it easier to work with videos in C++, as they provide a simple interface for common video processing tasks.

4.2.2) Design Choices in Video Processing

One design choice in the Video class is the use of method chaining, where the result of one method is used as the input to another method. For example, the grayscale method returns a new Video object containing the grayscale version of the original video, and this Video object can be passed to another method, such as show or thresholding. This design

allows for easy composition of video processing operations, making it easy to apply multiple transformations to a video.

A design choice made in both Video (and Image) class was to set the destination points in the create_homography function according to the corner points of the underlying image or video frame. Our intention with this decision was to reduce the number of inputs by mapping a selected subimage to the original image frame (essentially focusing part of an image), as well as creating a useful use-case of extracting parts of an image.

Finally, take a look at our use of separate functions for each of the main tasks performed by the class: thresholding, tracking, and object detection. This allows each task to be performed independently and makes it easy to understand and modify the code for each task.

4.2.3) Functions

- **Video():** This is the default constructor for the Video class. It initializes an empty Video object.
- **Video(const std::string& filename):** This constructor initializes a Video object with a video file specified by the filename argument. The video file is opened using the cv::VideoCapture class, and an error is thrown if the file cannot be opened.
- **Video(const cv::VideoCapture cap):** This constructor initializes a Video object with a cv::VideoCapture object specified by the cap argument. This allows the Video class to be used with videos that are not stored in files, such as videos captured from a camera.
- **void show(const std::string& filename = "Video"):** This method displays the video in a window with the specified filename. The video is displayed frame by frame, and the user can press the ESC key to exit the video.
- **Video saveAs(const std::string& filename="save.avi"):** This method saves the video to a file with the specified filename. The saved video has the same dimensions as the original video, and is encoded using the MJPG codec.
- **Video grayscale():** This method converts the video to grayscale and returns a new Video object containing the grayscale version of the original video. The grayscale video is displayed in a window as it is being saved to a file.
- **Video edge_detect(const int lower_threshold, const int upper_threshold):** This method detects edges in the video using the Canny edge detection algorithm, with the specified lower_threshold and upper_threshold values. The resulting video with detected edges is displayed in a window as it is being saved to a file.
- **Video gaussian_blur(const int kernel_sz):** This method applies Gaussian blur to the video using the specified kernel size, and returns a new Video object containing the blurred video.
- **static void add_point_cb(int event, int x, int y, int flags, void* data):** This method is a callback function that is called when the user clicks on the window while collect_points is running. It adds the clicked point to the std::vector<cv::Point> object that is passed as the data argument.
- **std::vector<cv::Point> collect_points(const std::string& window_name = "_tmp_collect"):** This method displays the video in a window and allows the user to select points on the video by clicking on the window. The selected points are returned as a std::vector<cv::Point> object.
- **Video create_homography(const std::vector<cv::Point>& points):** This method creates a homography matrix from the specified points, which can be used to transform the video by warping its perspective. The method applies the transformation to the video and returns a new Video object containing the transformed video.
- **Video threshold(const int type, const int value)** This method applies a thresholding operation to the video, and returns a new Video object containing the thresholded video. The

type argument specifies the type of thresholding to use, and the value argument specifies the threshold value. The possible values for the type argument are:
    - 1: Binary thresholding
    - 2: Binary inverse thresholding
    - 3: Truncate thresholding
    - 4: zero thresholding
    - 5: zero inverse thresholding
- **Video track():** This method applies object tracking to the video, and returns a new Video object containing the tracked video. The method uses the KCF (Kernelized Correlation Filters) tracker from the OpenCV library to track the object. The user must select the object to track by clicking on it in the video. The method displays the video in a window and overlays a rectangle on the tracked object. The method also displays the current frames per second (FPS) of the tracking operation.
- **std::vector<std::string> loadClassifications():** This function reads a text file containing the list of class labels (one label per line) and returns a std::vector<std::string> containing the labels. This list of labels is used to map the labelId of a detected object to its corresponding class label.
- **cv::dnn::Net loadDNN():** This function loads the pre-trained object detection model from a file and returns an instance of cv::dnn::Net representing the model. This model is used to make predictions on input images.
- **cv::Mat formatInput(cv::Mat &image):** This function takes an input image and resizes it to a square shape, with the same size as the input size of the object detection model. This is done by padding the original image with black pixels to the sides to make it a square shape. The resized image is returned. This step is necessary because the object detection model requires a fixed input size.
- **std::vector<Detection> obtainOutput(cv::dnn::Net &net, cv::Mat &input, const std::vector<std::string> labels):** This is a helper function used by the detect function to process the output of a neural network object detection model. This function receives the output of the neural network in the form of a std::vector<cv::Mat> object, along with a list of labels for the classes of objects that the model is trained to recognize, and returns a list of Detection objects containing information about the detected objects in the input image.
- **void putDetection(std::vector<Detection> &output, cv::Mat &frame, const std::vector<std::string> labels):** It takes a list of detections, an image frame, and a list of labels and draws a rectangle around each detected object, labels the object with its corresponding class label, and colors the rectangle with a unique color.
- **Video detection():** It applies object detection on each frame of a video using a pre-trained object detection neural network. It first loads the classification labels and the neural network, then applies object detection on each frame and displays the resulting frames with the detected objects highlighted. Finally, it saves the video with the detected objects to a file.

4.3) User Interface

4.3.1) Introduction

UI is a user interface for the Image and Video classes, which are intended to provide image and video processing functionalities. The UI class provides utility functions for taking input from the user and performing various image processing operations such as alpha blending, edge detection, Gaussian blur, perspective transformation, and object feature

extraction. These operations are accessed by the user through a command-line interface. The UI class uses the Image and Video classes to perform the underlying image processing operations.

For example, the *alpha_blend* function prompts the user to enter the name of another image to blend with the input image, and a weight for the other image. It then creates an Image object for the other image, resizes it to the same size as the input image, and uses the Image class's alpha_blend method to blend the two images together.

The UI class also provides functions for perspective transformation and object feature extraction. The *homography_perspective* function prompts the user to select four points on the input image, and uses these points to create a perspective transformation using the Image class's create_homography method. The *get_object_features* function prompts the user to select points on the input image that enclose an object, and then uses the Image class's get_mask, get_1st_moment, and get_0th_moment methods to extract the object's mask, centre of mass, and area.

Overall, our UI class provides a convenient interface for accessing the image and video processing functionalities provided by the Image and Video classes. The user can perform various image processing operations by entering input through the command line, and the UI class uses the Image and Video classes to perform the underlying image processing operations.

4.3.2) Design Choices in User Interface

One of the design choices in this class is the use of static member functions for the image and video processing functions. This means that these functions do not need to be called on an instance of the UI class and can be invoked directly.

This design choice may be useful because it allows these functions to be used more easily and flexibly, without the need to create an instance of the UI class. Additionally, because the functions are static, they can access only static members of the UI class, which means that they do not have access to any state that may be stored in a non-static member variable. This can simplify the implementation and prevent potential conflicts between different instances of the UI class. The main reasoning behind each static function was to demonstrate individual use-case functionalities.

Another design choice in this class is the use of templates for some of the functions that require a numeric input from the user, such as the get_val_from_user function. This allows these functions to be used with different numeric types, such as int or double, without the need to write separate versions of the function for each numeric type. This can make the code more reusable and flexible.

4.3.3) Functions

- **static Image get_image_from_user(const std::string& name = ""):** This is a utility function that prompts the user to enter the filename of an image. The name parameter is optional and is used to provide a custom prompt for the filename input. The function returns an Image object constructed using the entered filename.

- **static Video get_video_from_user(const std::string& name = ""):** This is similar to the above function, but it prompts the user to enter the filename of a video instead of an image. It returns a Video object constructed using the entered filename.
- **template <typename Num_T> static Num_T get_val_from_user(const std::string& name = ""):** This is a function template that prompts the user to enter a value of a specified numeric type. The name parameter is optional and is used to provide a custom prompt for the value input. The function returns the entered value of the specified type.
- **static Image alpha_blend(const Image& img_in):** This is a utility function that performs alpha blending of the given img_in image with another image entered by the user. It calls the Image::alpha_blend() method to blend the two images, using a weight for the second image entered by the user. The blended image, Image object, is returned.
- **static Image edge_detect(const Image& img_in):** This is a utility function that performs edge detection on the given img_in image. It prompts the user to enter lower and upper threshold values for the edge detection operation, and calls the Image::edge_detect() method to detect the edges using the entered thresholds. The edge-detected image is returned.
- **static Image gaussian_blur(const Image& img_in):** This is a utility function that performs Gaussian blurring on the given img_in image. It prompts the user to enter the size of the blur kernel, and calls the Image::gaussian_blur() method to blur the image using the entered kernel size. The blurred image is returned.
- **static Image homography_perspective(const Image& img_in):** This is a utility function that performs perspective transformations on the given img_in image. It prompts the user to enter 4 points on the image, and calls the Image::collect_points() method to collect the entered points. It then calls the Image::create_homography() method to create a homography matrix using the collected points, and applies the matrix to the image to perform the transformation. The transformed Image is returned.
- **static Image get_object_features(const Image& img_in):** This is a utility function that detects and outputs various features of an object in the given img_in image. It prompts the user to enter points of an enclosing polygon around the object, and calls the Image::collect_points() method to collect the entered points. It then calls the Image::get_mask() method to create a binary mask of the object using the collected points. It also calculates and outputs the object's centre of mass using the Image::get_1st_moment() method, and the object's area using the Image::get_0th_moment() method. An Image object constructed from the selected mask is returned.
- **static Image proj_img(const Image& img_in):** This is a utility function that projects a cutout from one image onto another image. It prompts the user to enter 4 points on the img_in image, and calls the Image::collect_points() method to collect the entered points. It then prompts the user to enter the filename of another image, and creates an Image object using the entered filename. It prompts the user to enter 4 points on this new image, and calls the Image::collect_points() method to collect the entered points. Finally, it prompts the user to enter 4 points on the img_in image again, and calls the Image::proj_img() method to project the cutout from the new image onto the img_in image using the collected points as the source and destination points. The resulting image is returned.
- **static void threshold_options():** This is a utility function that outputs the available thresholding options for the Image::threshold() method. These options are binary thresholding, inverted binary thresholding, truncated thresholding, thresholding to zero, and inverted thresholding to zero.
- **static Image threshold_mask(const Image& img_in):** This is a utility function that applies thresholding to the given img_in image to create a binary mask. It calls the threshold_options() function to output the available thresholding options, and prompts the user to enter the desired thresholding type and value. It then calls the Image::threshold() method to apply the chosen thresholding to the image using the entered value, and returns the resulting binary mask.

- **static Image grayscale(const Image& img_in):** This is a utility function that converts the given img_in image to grayscale. It calls the Image::grayscale() method to convert the image to grayscale and returns the resulting image.
- **static Image detection(const Image& img_in):** This is a utility function that performs object detection on the given img_in image. It calls the Image::detection() method to detect objects in the image and returns the resulting image with the detected objects highlighted.
- **static Video edge_detect(Video& cap_in):** This is a utility function that performs edge detection on the given cap_in video. It prompts the user to enter lower and upper threshold values for the edge detection operation, and calls the Video::edge_detect() method to detect the edges in the video using the entered thresholds. The edge-detected video is returned.
- **static Video gaussian_blur(Video& cap_in):** This is a utility function that performs Gaussian blurring on the given cap_in video. It prompts the user to enter the size of the blur kernel, and calls the Video::gaussian_blur() method to blur the video using the entered kernel size. The blurred video is returned.
- **static Video homography_perspective(Video& cap_in):** This is a utility function that performs homography and perspective transformations on the given cap_in video. It calls the Video::collect_points() method to collect 4 points on the first frame of the video. It then calls the Video::create_homography() method to create a homography matrix using the collected points, and applies the matrix to the video to perform the transformation. The transformed video is returned.
- **static Video threshold_mask(Video& cap_in):** This is a utility function that applies thresholding to the given cap_in video to create a binary mask. It calls the threshold_options() function to output the available thresholding options, and prompts the user to enter the desired thresholding type and value. It then calls the Video::threshold() method to apply the chosen thresholding to the video using the entered value, and returns the resulting binary mask.
- **static Video grayscale(Video& cap_in):** This is a utility function that converts the given cap_in video to grayscale. It calls the Video::grayscale() method to convert the video to grayscale and returns the resulting video.
- **static Video tracking(Video& cap_in):** This is a utility function that performs object tracking on the given cap_in video. It calls the Video::track() method to track objects in the video and returns the resulting video with the tracked objects highlighted.
- **static Video detection(Video& cap_in):** This is a utility function that performs object detection on a video stream. The function takes a Video object as input and returns a Video object containing the processed video stream with the detected objects highlighted. This function uses the underlying image processing capabilities of the Image class to perform the object detection operation.
- **static void print_header():** This is a utility function that prints a header to the command line, identifying the program as an "OpenCV C++ library util". This function does not take any inputs and does not return any output.
- **static void print_main_menu():** This is a utility function that prints a menu of options to the command line for the user to choose from. The menu presents the user with two options: "Image Processing" and "Video Processing". This function does not take any inputs and does not return any output.
- **static void print_image_menu(), static void print_video_menu():** They are simple functions that print a menu of options to the command line for the user to choose from when performing image or video processing tasks. The menu presents the user with a list of operations that can be performed on an image, such as reloading the image, displaying the image, applying alpha blending, applying edge detection, applying Gaussian blur, applying homography perspective transformation, finding object features, projecting the image, applying thresholding, converting the image to grayscale, performing object detection, and saving the image.

- **static void image_cases(), static void video_cases():** They are functions that take input from the user to select a corresponding static image or video processing function, defined in the UI file. Both functions take user input in the form of a switch in a do-while loop. Each case in the switch statement corresponds to a processing option defined in the static Video or Image utility functions.

5) Benchmarking

Language Benchmark:

We were interested in determining whether our wrapper for OpenCV would be faster than the Python code using OpenCV, which actually runs C++ in the background. To measure the time taken by each function call, we created a *Benchmark.cpp* file. Some of the functions of this class includes:

- *void load_benchmark()*
- *void alpha_blend_benchmark()*
- *void edge_detect_benchmark()*
- *void gaussian_blur_benchmark()*
- *void threshold_benchmark()*
- *void grayscale_benchmark()*
- *void detection_benchmark()*

They are pretty self explanatory. They simply run the relevant functions, time them and output the results. To give you an idea, let's take a look at the alpha_blend_benchmark():

```
//  benchmark blending two images
void alpha_blend_benchmark(const int ITERATIONS){
    Image img1 = Image("sp500.png");
    Image img2 = Image("times-square.png");
    auto totalTime = 0;
    for(int i=0; i<ITERATIONS; i++){
        auto start = std::chrono::high_resolution_clock::now();
        img2.fit_to_size(img1);
        img1.alpha_blend(img2, 0.5);
        auto end = std::chrono::high_resolution_clock::now();
        auto time = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
        totalTime += time.count();
    }
     std::cout  << "Alpha Blend (average time, ms): " << (totalTime/float(ITERATIONS))/1000.0 << std::endl;
}
```

As can be seen above, it starts the timer right before the process for alpha blending begins, and terminates right after. As a design choice, we are using the ITERATIONS constant instead of hard-coding it. This gives us the freedom to test different numbers of iterations easily. It is important to run the benchmark multiple times and average the results to ensure a reliable measurement.

We also created a benchmark script *Benchmark.py* written in Python to compare the efficiency of our implemented library against the Python code using OpenCV. We created an Image class that imitates the Image class we created in C++. Note that, we created that class in Python for the sake of benchmarking and testing, therefore it only implements certain portions of the functionalities in our C++ code, just enough to get them compared.

The equivalent method for the alpha blending benchmark in Python can be seen below:

```python
def image_alpha_blend_benchmark(self, iterations=100):
    img1 = Image("sp500.png")
    img2 = Image("times-square.png")

    total_time = 0
    for i in range(500):
        start = time.time()
        img2.fit_to_size(img1)
        img1.alpha_blend(img2, 0.5)
        end = time.time()
        total_time += (end - start)

    print('Alpha Blend (in ms): ' + str((total_time / iterations) * 1000))
```

Here are some comparison of some functionalities on our project against the OpenCV functions through Python):

	Edge Detection	Gaussian Blurring	Alpha Blend
Our library & utility	2.58	2.88	1.48
OpenCV with Python	2.67	3.04	1.89
Results	**we are 3% faster**	**we are 5% faster**	**we are 22% faster**

* Results are in milliseconds.

To ensure the results were hardware-independent, the tests have been conducted on the same machine. Furthermore, the benchmarking methods use the same approach and call the same OpenCV functions. To get the above results, we ran through the benchmarks by passing the same images and properties in the functions 500 iterations, averaged the results.

To conclude, you can observe that even with the additional functionality, it still runs faster than Python.

6) References

OpenCV Documentation
https://docs.opencv.org/4.6.0/

OpenCV Tracking API
https://docs.opencv.org/3.4/d9/df8/group__tracking.html

Object Detection Paper: YOLO (You Only Look Once)
https://homes.cs.washington.edu/~ali/papers/YOLO.pdf

A Deep Learning Object Detection Method for an Efficient Clusters Initialization
https://arxiv.org/pdf/2104.13634.pdf

Object Detection Repository
https://github.com/ultralytics/yolov5

Image Processing Examples - Creating Object Masks
https://www.researchgate.net/figure/Segmentation-a-Originalimage-b-grayscale-image-c-Binary-mask_fig2_281783829

Homography Techniques
https://towardsdatascience.com/understanding-homography-ak-a-perspective-transformation-cacaed5ca17

Alpha Blending
https://developer.nvidia.com/content/alpha-blending-pre-or-not-pre

Edge Detector
https://docs.opencv.org/4.x/da/d22/tutorial_py_canny.html

Gaussian Blurring
https://datacarpentry.org/image-processing/06-blurring

Video Detection
https://medium.com/mlearning-ai/detecting-objects-with-yolov5-opencv-python-and-c-c7cf13d1483c