MWT Programmer's Guide

Introduction

The Multi-worm Tracker (MWT) is a population-level moving object analyzer. The name MWT describes a set of interacting hardware and software components that acquires images of biological samples at a high frame rate, analyzes those images in real-time, controls automated stimuli delivery systems, and produces data.

Algorithm

Algorithmically, the MWT follows the following steps: initial memory image creation, image correction through memory image subtraction, iterative flood fill object detection, tracked object registration and new object acquisition.

A memory image is created and maintained throughout the running of the algorithm. It is updated by adding a proportion of a newly acquired frame to a proportion of the existing memory image. These proportions are defined by the user and describe the rate at which objects are incorporated in to the memory image. The pixels that comprise tracked objects do not participate in the memory image update; thus tracked objects that cease to move do not disappear in to the memory image.

Image correction is a process of performing a pixel by pixel subtraction of the memory from the input image, with a grey value added. Any input pixels that are of a different intensity than those of the memory image represent something that is potentially of interest. If the resulting subtracted pixels are darker than grey, the object is new in the input frame. If they are lighter than grey, it is an object that is new in the memory image, and is subsequently ignored. This behavior is due to the intensity thresholding used in the object detector; any pixels outside the defined ranges are ignored.

Objects are detected via iterative flood filling. Every pixel is examined against a user-specified range. All pixels within this range have their neighbors checked, continuing in all directions until a pixel that falls outside the range is found. When all possible pixels have been exhausted, an object is returned, containing data on that object.

Every frame, every detected object is searched to locate ones that significantly overlap with currently tracked objects. All results are marked in the data stream: when the object no longer exists, exists and is in bounds, exists and is out of bounds, has collided with any number of other objects and has separated in to any number of other objects. Collisions and object nihility result in the destruction of tracked objects and, in the case of collisions, the creation of a single new object engirdling the collided objects. Tracked object separation also results in the destruction of a tracked object and creates new objects for each distinct separated object. Objects outside of user-set bounds are ignored, while objects in bounds and found have their data for that frame updated.

The final set of processes performed on any new frame is the detection of new objects and the updating of the memory image. Both of these processes are only performed within a subset of the image, a window with user-defined dimensions that is updated to a new part of the image each frame, wrapping when it reaches the bounds of the image. New objects are those sets of pixels detected by the flood fill algorithm and not registered to an existing object. The memory image is updated as described above.

The most recent version of this document is available at https://sourceforge.net/projects/mwt/files/mwt/1.1.0%20Release/documentation/MWT_Source_Code_Guide.pdf/download

Achievement of the algorithm through component interaction

The performance of the MWT algorithm is made possible by the interactions of three systems; a Labview to C interface, a C interface to C++ library, and an interface from the C++ library to the LabView system.

LabView is a procedural data flow description system. As such, the C-interface creates a procedural front-end on the object-oriented library. The functions which transmit data to LabView package LabView array data structures in to MWT library images and pass along user input from the front panel to the library, including all settings, definition of locations for reference objects, and specification of Regions of Interest (ROI).

The C interface to the C++ library controls the behavior of the class instances it instantiates. Functions are used to control the state of the objects, which change their behavior appropriately. The C interface to LabView repackages MWT library image data in to LabView arrays and reports data points requested about objects tracked upon request.

LabView stores array data as a pointer and an array of dimensions. The array data is flattened in to one dimension. The C interface translates this information in to an Image which does not control its own memory. These arrays represent image data generated by LabView, and as such can be changed by user manipulation of the LabView front panel. Fortunately, all data necessary to interpret the information is provided.

LabView Virtual Instruments

LabView (LV) is used as the conductor of the MWT. It is the coordinator between the input and output of hardware control signals and data streams, and governs the behavior of the MWT state machine. It would be naive of me to say the MWT is robust to failure, prone to dying gracefully rather than with a lot of kicking and screaming, though I can say we have tried to follow rigorous principles wherever possible. LabView is fail-soft; errors generated by VIs can be caught internally, meaning that, for example, required but not installed drivers cause the program simply not to load, rather than crash unexpectedly.

The program begins in the configuration state. In this state, the user has access to all the library configuration functions. These include the type of tracking, ROI and reference object specification, object size and intensity thresholds, and image binning. Image input can be interpreted either as bright field or dark field. This interpretation determines what intensity range within which important objects are expected to fall, and is used internally to calculate pixel intensity ranges from user input. Regions of Interest can either be rectangles or ellipses, but both share the following in common: they can be set multiple times, can be separate or overlap, and are removed one at a time, in a Last In, First Out fashion. Rectangular and elliptical ROIs are defined by two sets of (X,Y) coordinates. For both ROI types, these points, swapped if necessary, are the upper left and lower right corners of a rectangle. However, for elliptical ROIs, these points define the rectangle that encloses the ellipse and whose sides intersect it. The points can be entered as nonsense, such as negative values or rectangles with minuscule area, causing a nonsense ROI, but this should not cause the MWT to crash. Finally, reference objects, if set, define a single point, the centroidian of the reference object, that is a reference point for all other measurements made in the MWT algorithm. It is used when the entire image is expected to move a significant amount between frames.

During the configuration state, all changes made to the library are made immediately, with feedback of the object detection algorithm available. While even the most radical change to the library's configuration will eventually have its effect applied smoothly, some changes are most quickly enacted by reseting the MWT. These changes include changing the image binning size and the camera bit depth.

Feedback on the efficiency and behavior of the MWT algorithm is obtained through the image display section of the front panel. Its default behavior is to be on during initial setup and off during run. Note that there is a performance penalty for having display turned on due to the extra processing required to generate the image. A list provides access to the three images available for display: fixed, memory and raw. The fixed image is the result of the MWT algorithm. It will contain any annotations active during that state of the MWT; during initialization it will merely display the masks for tracked objects, however, during an experiment run it will display masks, bounding boxes, contours and skeletons, if they are enabled, the Region of Interest and any reference objects.

The graph displays are only active during an experimental run. This online information is not accessible through these graphs at any other time. Each of the graph displays is an independent view of the online statistics generated by the MWT. All data is stored on the local file system, the graphs represent a subsample of the data generated.

There are four implemented paths for image data input. These are IMAQ, IMAQdx, DirectShow and file input through an AVI. The IMAQ interface allows control of all CameraLink cameras, while the IMAQdx interface allows control of GigE and Firewire cameras. DirectShow is a beta library that permits image acquisition from USB cameras. AVI file input is possible, but requires more user input and interaction. LabView's libraries for AVI file handling are primitive, therefore the user is required to provide the bits per pixel and frame rate for the AVI. Then, the behavior of the instrument during initial setup should be changed such that the AVI file loops while in the setup state, allowing any necessary adjustments to the algorithm. Finally, the experimental duration of the instrument should be locked to the length of the AVI such that it will run through once for data acquisition and analysis.

LabView also has the responsibility performing any necessary data coercion. If the bit depth of the input image data is too small, it is translated to a higher bit depth using VIs from the NI Vision package.

National Instrument drivers are required for any hardware used in the MWT and for any hardware you seek to use. While the actual hardware devices are not required during a run of the MWT, and will not cause the system to crash if the VIs that control them are used, the program will not execute if the required drivers are not loaded.

There is a specific pit-fall that has become clear through experience that has to do with editing virtual instruments whose front panels are intended to be displayed. Any VIs front panel can be configured to display when it is called through a menu option available through the VIs icon in the wiring diagram. However, if this configuration is set when you edit the VI, it becomes unset. This requires you to reconfigure the VI's calling behavior after each edit.

Counter Card Control

The Measurement Computing Counter (MCC) card is used to generate a Transistor-Transistor Logic (TTL) signal; a pulse of 5V representing active and 0V being inactive. The outputs of the MCC card are used to create either a single, user-defined interval signal or a user-defined set of signals generated at a user-defined interval.

The outputs of this card are programmable, capable of holding a particular voltage level based on the behavior of the internal clocks and gate inputs. The clock frequency defines how long counting to a value takes.

A single pulse is achieved via a single counter with two values; a hold value and a load value. The hold value is how long the counter waits between counts while the load value is what it counts to. In terms of observable behavior, the load value is the time between a pulse and the hold value is the duration of the pulse.

A multi-pulse relies on two counter linked together. The output of one counter is used to control the behavior of the second counter. The first counter controls the time between pulse set generation; its output controls activation of the second counter, which will continue to generate a pulse as long as its gate is active. The first counter is programmed to continue its counting cycle indefinately, thus providing the desired output.

Both of these pulse generation behaviors are controlled through crafted virtual instruments. For single pulses, a counter is configured to only generate a single count. As such, only the hold value, or duration of the pulse, is set. This is also the case with the first counter, which controls time between pulse set generation, used in multi-pulse.

Arbitrary Function Generator control

As part of the expansion of the MWT to track Drosophila larvae, a Tektronix Arbitrary Function Generator was attached to the system. It is used to generate wave forms that drive a speaker, providing a new type of stimuli to the larvae. It is triggered from the MCC card, as all stimuli are, and controlled through virtual instruments that allows the setting of frequency and amplitude, with changes to these settings at user-defined intervals.

Experimental Coordination

The LabView virtual instruments define how the experiments are coordinated. They trigger the necessary events in the control flow; from acquiring images, to loading them, to processing them, relocating tracked objects and finding new ones, to finalization, with memory deallocation and initial state restoration.

C DLL, MWT_DLL.h

This header defines the procedural interface which passes LabView requests in to the MWT C++ library. It is a wrapper around an instantiation of an object-oriented class. The methods defined perform translations from data types provided by LabView, such as arrays, in to MWT data types, such as Image.

C++ Library

All of the work described in the above algorithm is performed by the code described in the C++ Library. This library is broken down in to a set of header files, each of which describes a set of classes designed to solve a sub-set of the MWT tracking problem.

Internally, the bit depth of the input image data is assumed to be at least 9 bits.

MWT_Blob.h

This header defines the core classes for collecting data on moving objects differing from the background. Data are collected at three levels; the Blob, which describes a connected group of pixels, the Dancer, which describes a single Blob over time, and the Performance, which describes a group of dancers over time. While there are other classes contained in this header, they are ancillary to the above described classes.

Blobs manage their own counters, skeletons, masks, and flood information. Blobs can be used as temporary storage during different stages of the tracking algorithms or to permanently describe a found object. All image-derived object information is stored as pixel strips.

Along with keeping track of every frame an active Blob is tracked, Dancers contain the necessary temporary storage for candidate Blobs.

Performances keep track of the population of Dancers. It resolves conflicts between Dancers, such as when they collide or disappear or split from a single blob in to multiple or move from within the Region of Interest.

MWT_Geometry.h

This header contains class descriptions of basic geometry concepts. These classes include operators and accessor functions. Point, Rectangle, and Ellipse are defined herein.

MWT_Image.h

This header contains the class descriptions most important for image processing. This includes strips, masks, contours, ranges, images, and intermediate data structures.

Strips represent an x-coordinate and y-range. The class includes functions for performing Strip arithmetic.

A Mask is a sorted set if Strips. Its class contains functions for, among other things, basic Mask operations, such as dilation, erosion, and intersection. Masks perform their own bounds checking.

A Contour represents a closed curve of points.

A Range is a single inclusive pair of numbers, while a DualRange is two ranges of numbers that are expected to overlap. These ranges are an initial range and hysteresis value.

An Image is used for both unprocessed input data and memory images. It can control its own memory, responsibly destroying itself, or act through a pointer to externally controlled memory. Images can either be binned or not, with binning performed through local area averaging techniques at access time. Any Image can store and generate raw, corrected, and memory images, assuming the data was previously set. They can also dump stored data to TIFF images created on the local disk.

FloodInfo and FloodData are intermediate data structures. FloodInfo stores information about Blobs while the flood filling algorithm is running, while FloodData keeps useful information about a Blob after filling.

MWT_Library.h

The purpose of this library is to describe metadata about an experiment, with procedural wrappers for Performances. In order to accomplish this purpose, it defines the following classes; SummaryData, TrackerEntry, and TrackerLibrary. SummaryData contains meta information that should not conceptually exist in a Performance. Such data includes the population-level statistical information generated from a frame and a list of “events” that have occurred. TrackerEntry is a wrapper for a single

Performance, including all necessary data input and output settings. Finally, TrackerLibrary operates as a Performance factory, allowing C-style function interfaces to an internally-managed instance of a Performance.

MWT_List.h

This library contains the descriptions for templated linked list and stack class data structures. These classes are defined such that any other class can be cast used as the internal nodes of a list or stack.

MWT_Model.h

The classes herein defined are used to simulate input data to the MWT; ModelWorm, that will generate worm-like data, and ModelCamera, which will create images for the system.

MWT_Storage.h

Within this header are defined various factories for the allocation of memory for data structures. The factories are all defined as templates. Storage is the base-class for factory style object instantiation. Classes which support these instantiation factories must be made compatible with or wrapped by a Listable class. ManagedList is a list that can de/allocate its own memory. DualList is a pair of ManagedList that refer to the same data. Array1D, Array2D, Dualable and Dualled are support classes for the described functionality.

This Document was written by Nicholas A. Swierczek