

# BOT: A C++ Library for Biomedical Object Tracking

## Design Document and User Guide

### Abstract

Biomedical Object Tracking (BOT) is a C++ library for tracking a massive number of objects from time-lapse image sequences. It features (I) object association based pairwise tracking, (II) interactive structured learning for optimal feature parametrization, (III) generic and extensible feature design, and (IV) configurable workflow for different applications (e.g. tracking cells and tracking worm). In this report, we provide the technical details on the implementation of the aforementioned algorithms, features and detailed guide to installing and using this library.

## 1 Introduction

Reliable multiple object tracking is fundamental to the interpretation of time-lapse microscopic image data in biomedical research. Recently, object association based on mathematical programming for object tracking has drawn much attention due to its high efficiency in solving large-scale, complex tracking problems such as lineage-tree reconstruction [2] and cell culture study [4, 3]. We recently developed structured learning for cell tracking, an approach for learning the optimal parameterization of the high-dimensional association features from a training set of ground truth associations [1]. Experimental results show that this approach not only provides superior tracking performance but also enables biologists to contribute their expertise in an intuitive fashion.

Despite the popular use of object association in many tracking problems, there has not been an open implementation of this method, let alone a well-designed library with high extensibility and standard interfaces. We attempt to address this situation by introducing BOT, a extensible C++ library for biomedical object tracking. Though initially designed for cell tracking, we hope to make BOT a generic algorithmic library adaptable to various biomedical tracking applications. Briefly, BOT features:

**Solver** An efficient pairwise object association solver based on integer linear programming (ILP);

**Feature** A rich set of generic features and extensible feature design using factory pattern;

**Learning** Structured learning for high-dimensional feature parametrization;

**Diversity** Configurable workflow for supporting diverse applications;

**GUI** A user-friendly GUI by integration with the Interactive Learning and Segmentation Toolkit (ilastik, <http://www.ilastik.org/>) [6].

## 2 Implementation Details

### 2.1 Prerequisites

BOT has two prerequisites: data and solver system.

Regarding data, BOT assumes that the objects of interest have been segmented or detected from the raw images. The corresponding segmentation or detection algorithms are usually problem-specific. The result is a labeling, in which BOT accepts value zero as background and considers all pixels with identical label (other than zero) as a single object.

Regarding solver systems, BOT relies on at least an integer linear programming (ILP) solver for predicting the tracking. To perform structured learning for parameter optimization, quadratic programming (QP) or linear programming (LP) solver is required. By default, BOT uses IBM ILOG CPLEX<sup>1</sup> which provides solvers for ILP, QP and LP problems. It is nevertheless possible and easy to interface BOT with alternative solvers such as lpsolve<sup>2</sup> and Gurobi<sup>3</sup>. More details on installing CPLEX and on using customized solver systems will follow in Section 3.1.2 and Section 2.5.3, respectively.

## 2.2 Concepts

### 2.2.1 Object, Singlet and Multiplet

An object literally refers to an object of interest whose behavior is to be tracked. Object is an abstract concept and has two concrete instances: *singlet* and *multiplet*. A singlet is exactly a segment or detection which consists of pixels with the same label (nonzero). It represents the very rudimentary entity but is not sufficient for explaining sophisticated biomedical events. We therefore introduce multiplet which is a combination of several singlets (usually restricted to a neighborhood system). Multiplets capture events in which more than one singlet are involved, such as the two daughter cells from a cell division and the multiple parts from an over-segmented worm. In its current implementation, BOT only considers multiplet consisting of two singlets, see Section 2.7 for more details.

In BOT, class `Object` is the base class for class `Singlet` and class `Multiplet`. It employs a sparse representation (i.e. a point cloud) of the pixels within the object, i.e. their coordinates, intensity values and labels. It also stores the object features (see Section 2.2.3) as a vector of `Matrix2D` (i.e. `std::vector<Matrix2D>`). Class `Singlet` and class `Multiplet` instantiate class `Object`. To generate all singlets and multiplets in a frame with configurable neighborhood constraints, use class `SingletGenerator` and class `MultipletGenerator`, respectively. These generators assign an id to each singlet/multiplet that is unique within this frame.

### 2.2.2 Event and Hypothesis

An event is one particular type of association between at least two and possibly more objects respectively from a pair of neighboring frames. An event has a particular biological or technical interpretation. For example, cell division is a biological event and the involved objects are the father cell from one frame and the two daughter cells from the next frame. For another example, a sudden over-segmentation (i.e. split) is an event caused by a technical limitation.

Multiple events can be defined to handle more complex tracking problems. For example of cell tracking with imperfect segmentation, we can define six events as in Table 1. The first four events express the usual cell behaviors and the last two capture segmentation errors, i.e. split for over-segmentation and merge for under-segmentation. In the header of the table, Pairing specifies the types of objects involved, e.g.  $1 \rightarrow 1$  for singlet to singlet,  $1 \rightarrow 2$  for singlet to multiplet, and  $1 \rightarrow \emptyset$  for singlet to *nothing* (considered as a special type

<sup>1</sup><http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/>

<sup>2</sup><http://sourceforge.net/projects/lpsolve>

<sup>3</sup><http://www.gurobi.com>

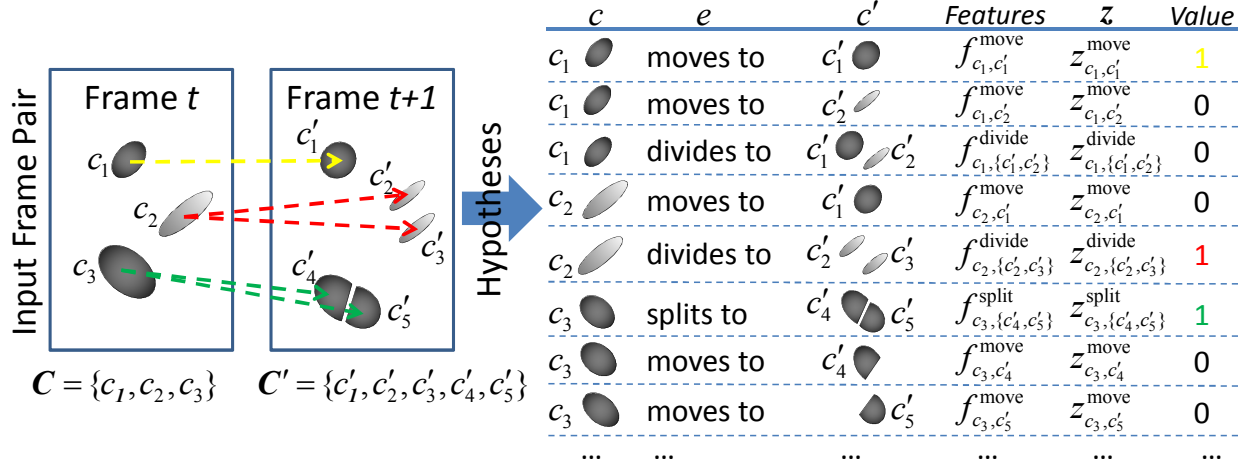


Figure 1: Toy example: two frames of objects and a list of the possible associations as hypotheses. One particular interpretation of the scene is indicated by colored arrows (left) or equivalently by a configuration of binary indicator variables  $z$  (rightmost column in table).

of “object”). BOT uses class EventConfiguration to import the definition of events from an INI file<sup>4</sup> and manages them. Please refer to Section 2.4 for more details on configuring the workflow and on preparing the INI file.

Name	Pairing	Feature 1	Feature 2	Feature ...
Move	$1 \rightarrow 1$	Change of size	Spatial displacement	...
Division	$1 \rightarrow 2$	Angle pattern	Father cell intensity	...
Appearance	$\emptyset \rightarrow 1$	Distance to the border	Overlap with the border	...
Disappearance	$1 \rightarrow \emptyset$	Distance to the border	Overlap with the border	...
Split	$1 \rightarrow 2$	Shape compactness	Mass evenness	...
Merge	$2 \rightarrow 1$	Shape compactness	Mass evenness	...

Table 1: Event definition for cell tracking (six types of events).

Given the definition of events, hypotheses are essentially their instantiations. As shown in the toy example in Fig. 1, hypotheses are generated throughout the entire sequence for each type of event, subject to a pre-defined neighborhood constraints including the  $k$  nearest neighbor and the spatial distance threshold. BOT uses class HypothesisSpace to generate and manage all hypotheses throughout the entire sequence. For a particular pair of frames, its hypotheses and the corresponding joint features (see Section 2.2.3) are represented using class FramePair.

### 2.2.3 Object Features and Joint Features

An object feature characterizes the object only and is extracted from the object itself (e.g. position, size and principal components) or together with the global context information (e.g. distance to the border and overlap with the border). Object features are normally generic and primitive.

A joint feature, on the other hand, characterizes a hypothetical association. Extracted from the object features, it is a measure of the compatibility of the raw information and the hypothetical association. For example, spatial displacement is a joint feature computed from the position of the two objects and it is

<sup>4</sup>[http://en.wikipedia.org/wiki/INI\\_file](http://en.wikipedia.org/wiki/INI_file)

intended to respect the speed limitation of movement (e.g. cell migration). Appropriate distance measure should be selected according to the object feature, such as using earth mover’s distance [5] for comparing two intensity histograms (see class `MeasureEarthMoversDistance`). Note that joint features are not always extracted from both objects in the hypothesis. For example of cell division, the father cell usually appears brighter than the usually ones so that we directly take the mean intensity of the father cell as a “joint feature” for the division event.

BOT has already included a rich set of generic object features and a handful of joint features. Furthermore, it adopts the factory method pattern to allow for easy extension of the feature set and dynamic feature loading. More details are provided in Section 2.5. To avoid possible confusion of these two concepts, in BOT we refer to the joint feature as *measure* since it essentially provides a measure of the compatibility. We refer to the object feature simply as *feature*. Class `JointFeatureExtractor` extracts the object features and class `MeasureExtractor` extracts the measures.

## 2.3 Data Structure and Representation

### 2.3.1 Primitive Data Types

To enforce consistency throughout the library, we always use `int32` for indices and object ids. Furthermore, we define `Matrix2D`, a data type for representing the raw images, segmentations and feature matrices. The advantage of such representation is that we can use powerful linear algebra operations to facilitate the feature computation such as those in VIGRA (see namespace `vigra::linalg`).

```
// data type definition for indices , ids , etc .
typedef int int32;

// data type definition for images , segmentations , features , etc .
typedef double MatrixElem;
typedef vigra::MultiArray<2, MatrixElem > Matrix2D;
```

### 2.3.2 Objects, Hypotheses and Features

We use point cloud to sparsely represent the raw pixels that belongs to an object such as their coordinates, intensity values and labels. The id (`int32` type) of a singlet is unique within a frame. The same applies to multiplet as well.

```
class Object {
    ...
protected:
    /* id of this object */
    int32 id_;
    ...
    /* 2D matrix of the coordinates of the pixels */
    Matrix2D pixels_;

    /* 2D matrix of the intensity values of the pixels */
    Matrix2D values_;
```

```

123
124     /* 2D matrix of the labels of the pixels */
125     Matrix2D labels_;
126 };

```

A hypothesis is represented by a pair of ids of the objects involved and a list of hypotheses is thus a vector of such pairs:

```

129 typedef std::vector<std::pair<int32, int32>> Hypotheses;

```

Features are always represented using type Matrix2D.

## 131 2.4 Configurable Workflow

### 132 2.4.1 Event Definition

BOT allows the definition of the events to be tailored to a specific problem. For example, division is a particular important event for cell culture study but not needed for worm tracking. As already shown in Table 1, an event definition consists of a name, a pairing and a list of joint features. The event's name must be unique. The feature setting will be introduced immediately in Section 2.4.2. Class EventConfiguration imports the event definitions from an INI file and manages them, such as:

```

138 [ Division ]
139 Pairing = 1 2
140 Feature1 = ObjectFeatureIntensitySum MeasureNormalizedEuclideanDistance -0.4680
141 Feature2 = ObjectFeatureCenters MeasureAnglePattern -1.3055
142 Feature3 = ObjectFeatureIntensityMean MeasureAppointmentLeft 0.9591
143 Feature4 = ObjectFeatureEccentricity MeasureAppointmentLeft -0.9441
144 Feature5 = ObjectFeatureVolumeEvenness MeasureAppointmentRight 1.6322
145 Feature6 = ObjectFeatureShapeCompactness MeasureAppointmentRight -0.7221
146 Feature7 = ObjectFeatureMassEvenness MeasureAppointmentRight -0.8504

```

### 147 2.4.2 Dynamic Feature Loading

Each event definition should be associated with a list of joint features (or measures). Each measure consists of three elements: the underlying object feature, the measure on this object feature and the weight. The object feature can be an instance of any class with a name similar to ‘‘ObjectFeature\*\*\*’’ (except class ObjectFeatureFactory and class ObjectFeatureExtractor). The same holds for the measures, that is ‘‘Measure\*\*\*’’ (again, except class MeasureFactory and class MeasureExtractor). The weight can be manually set or learned from some training examples (see Section 2.6 for more details).

BOT adopts the factory method pattern<sup>5</sup> to dynamically create extractors by their class names. Two respective factories are implemented, namely class ObjectFeatureFactory for object features and class MeasureFactory for measures. Class ObjectFeatureExtractor wraps the necessary functionalities for extracting multiple object features, just like class MeasureExtractor for measures. Only a list of class names are required as inputs. More details on the extensibility of this factory method pattern will follow in Section 2.5.

<sup>5</sup>[http://en.wikipedia.org/wiki/Factory\\_method\\_pattern](http://en.wikipedia.org/wiki/Factory_method_pattern)

## 2.5 Extensibility

### 2.5.1 Object Features

BOT has already included a rich set of generic object features but we encourage the developers to design new features that better suit their applications, and it can be efficiently accomplished. Following the factory method pattern, an virtual class `ObjectFeatureFactory` serves as the base class for object features, who has several virtual functions that need be implemented in the derived class:

```
class ObjectFeatureFactory {
public:
    /*! Default constructor
    *
    */
    ObjectFeatureFactory() {};

    /*! A virtual function that extract the object feature
    * @param feature_mat The object feature to be returned
    * @param obj The input object
    * @param context The global context
    */
    virtual void extract(Matrix2D& feature_mat, const Object& obj,
        const Context& context) = 0;

    /*! Return the shape (size) of the feature matrix
    * @param dim The input data dimension
    * @return A Matrix2D::difference_type object as the shape
    */
    virtual Matrix2D::difference_type shape(int dim = 2) = 0;
    ...
}
```

To add a new object feature, one simply has to

1. derive a new class from `ObjectFeatureFactory` and implement the virtual functions;
2. implement a static function `getClassName()` that returns an `std::string` as an identifier of this class;
3. in the source file `ObjectFeatureFactory.cxx`, add your newly implemented class in the following function (within the big `if-else` block) that identifies the target feature class by `std::string`, creates an instance of the corresponding class and returns a pointer to the instance.

```
ObjectFeatureFactory *ObjectFeatureFactory::make(const std::string& name) {
    ObjectFeatureFactory *fea = 0;
    if (name.compare(ObjectFeatureVolume::getClassName()) == 0) {
        fea = new ObjectFeatureVolume();
    }
    else if (name.compare(ObjectFeaturePosition::getClassName()) == 0) {
```

```

200         fea = new ObjectFeaturePosition();
201     }
202     else if ...
203 }

```

204 Furthermore, class `ObjectFeatureExtractor` creates and manages multiple object feature extractors  
205 such as deleting the extractors to avoid memory leak when necessary.

## 206 2.5.2 Joint Features

207 The procedure for extending the joint features (measures) is very much alike. The relevant classes/files  
208 are class `MeasureFactory`, class `MeasureExtractor` and file `MeasureFactory.cxx`.

## 209 2.5.3 Solver System

210 As introduced in Section 2.1, BOT allows for customized solver systems. To use one's own solver system,  
211 simply derive a new class from class `SolverSystem` and implement its two virtual functions for solving  
212 ILP and QP problems. These virtual functions are very much standardized interfaces for mathematical  
213 programming and are compatible with most state-of-the-art solver systems.

```

214 class SolverSystem {
215 public:
216     /*! Solve a binary integer linear programminig (binary ilp) problem with
217     * given equality and inequality constraints. In particular, it solves
218     * a problem as follows:
219     *          max      f'*x
220     *          s.t.     Aineq*x <= bineq
221     *                  Aeq*x  =  beq
222     *                  each variable in x is binary
223     * @param f The coefficient of the objective function
224     * @param Aineq The inequality constraints: Aineq * x <= bineq
225     * @param bineq The inequality constraints: Aineq * x <= bineq
226     * @param Aeq The equality constraints: Aineq * x = bineq
227     * @param beq The equality constraints: Aineq * x = bineq
228     * @param x0 The initial solution
229     * @param x The solution
230     * @param msg The return message
231     */
232     virtual std::string solve_bilp(
233         const Matrix2D& f,
234         const Matrix2D& Aineq, const Matrix2D& bineq,
235         const Matrix2D& Aeq, const Matrix2D& beq,
236         const Matrix2D& x0, Matrix2D& x) const = 0;
237
238     /*! Solve a quadaratic programminig (qp) problem with given equality and

```

```

239      *   inequality constraints and bounds. In particular , it solve :
240      *       min      0.5*x'*H*x+f*x
241      *       st .      Aineq*x <= bineq
242      *                  Aeq*x      = beq
243      *                  lb <= x <= ub
244      *   @param H Double matrix for objective function (quadratic term)
245      *   @param f Double matrix (vector) for objective function (linear term)
246      *   @param Aineq The inequality constraints: Aineq * x <= bineq
247      *   @param bineq The inequality constraints: Aineq * x <= bineq
248      *   @param Aeq The equality constraints: Aineq * x = bineq
249      *   @param beq The equality constraints: Aineq * x = bineq
250      *   @param lb The lower bound
251      *   @param ub The upper bound
252      *   @param x0 The initial solution
253      *   @param x The solution
254      *   @return A std::string as the message of the solution status
255      */
256      virtual std::string solve_qp(
257          const Matrix2D& H, const Matrix2D& f,
258          const Matrix2D& Aineq, const Matrix2D& bineq,
259          const Matrix2D& Aeq, const Matrix2D& beq,
260          const Matrix2D& lb, const Matrix2D& ub,
261          const Matrix2D& x0, Matrix2D& x) const = 0;
262      };

```

## 263 2.6 Tracking Prediction and Parameter Learning

### 264 2.6.1 Methodology

265 To learn more about the underlying methodology of tracking prediction and structured learning for pa-  
266 rameter optimization, please refer to our paper [1]. In BOT, they are implemented in class TrackingPredictor  
267 and class TrackingLearner, respectively. To learn more, simply follow the example in Section 3.3.

### 268 2.6.2 Two Representations of the Tracking Result

269 We use different representations of tracking result for computation and storage. For computation, the  
270 tracking result is merely a matrix of binary values (technically a vector) in which one means the correspond-  
271 ing hypothesis is accepted and zero otherwise. This representation is more of an intermediate solution of  
272 the ILP formulation or an input to the learning procedure. We therefore refer to it as Solution:

```

273 // a vector of ILP solutions (for multiple events)
274 typedef std::vector<Matrix2D> Solution;

```

275 Obviously, Solution is dependent on the hypothesis space and is therefore not suitable for storage.  
276 This is simply because one usually does not want to store the entire hypothesis space and the result should



277 be fully interpretable using the images and segmentations as its raw information. We therefore use an-  
278 other representation termed `LabelAssociation` which directly stores the labels from the segmentation or  
279 detection in a pair of matrices as the source and the target:

```
280 // associating the segmentation labels
281 struct LabelAssociation {
282     std::string name;
283     Matrix2D source;
284     Matrix2D target;
285 };
286
287 // a vector of label associations (for multiple events)
288 typedef std::vector<LabelAssociation> LabelAssociations;
```

## 289 2.7 Limitations

290 We declare two major methodological limitations of BOT:

- 291 1. BOT is currently restricted to pairwise object association. We are now investigating a global object  
292 association approach that hopefully will bring better tracking performance.
- 293 2. A multiplet is limited to only two singlets. We are investigating possible improvement such as replac-  
294 ing the singlet/multiplet generation with hierarchical segmentation.

## 295 3 User Guide

### 296 3.1 Installation and Compilation

297 BOT and several of its dependencies use CMake<sup>6</sup> as the building system. Make sure CMake is correctly  
298 install (minimum version 2.6) before processing any further.

#### 299 3.1.1 Install and Compile Dependencies

- 300 1. Download and install VIGRA from <http://hci.iwr.uni-heidelberg.de/vigra/>. The installation instruc-  
301 tions can be found at <http://hci.iwr.uni-heidelberg.de/vigra/doc/vigra/Installation.html>.
- 302 2. Download and install HDF5 from <http://www.hdfgroup.org/ftp/HDF5/current/src/>. The instal-  
303 lation instructions can be found inside the package.  
304 If you don't want to compile from source code, you may also use the pre-compiled binary package  
305 that is available here: <http://www.hdfgroup.org/HDF5/release/obtain5.html>.
- 306 3. To access IBM ILOG CPLEX, a license is required. Detailed guidelines can be found at  
307 <http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/>.  
308 You need a valid license to run CPLEX. For academic organization, you can check out this link:  
309 <http://www-01.ibm.com/software/websphere/products/optimization/academic-initiative/>.

---

<sup>6</sup><http://www.cmake.org/>

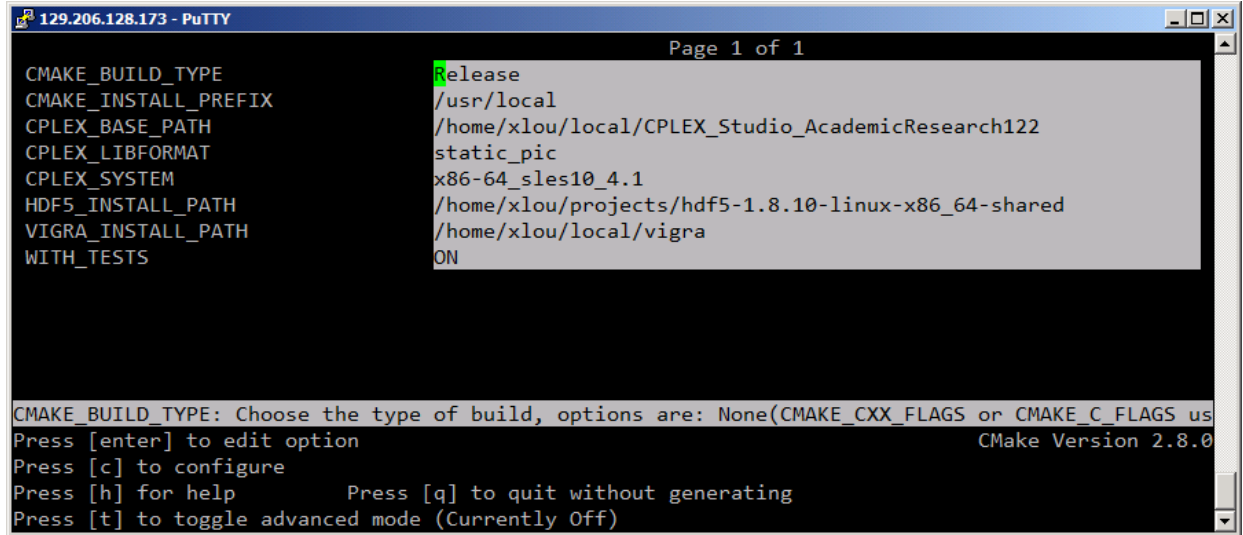


Figure 2: Configuration using ccmake GUI.

### 3.1.2 Install and Compile BOT

1. Download the BOT source code from <https://github.com/xlou/BOT/>.
  2. Open a console, go to the root directory of BOT and run command **ccmake ..**
  3. Specify variables in Table 2. The values for CPLEX\_SYSTEM and CPLEX\_LIBFORMAT can be found under [CPLEX root]/cplex/lib/. For example of the x86-64 architecture, the CPLEX library locates at [CPLEX root]/cplex/lib/x86-64\_sles10\_4.1/static\_pic/, which gives **x86-64\_sles10\_4.1** to CPLEX\_SYSTEM and static\_pic to CPLEX\_LIBFORMAT. An example is shown in Fig. 2.
  4. Download a sample dataset using this link:  
<http://hci.iwr.uni-heidelberg.de/Staff/xlou/export/dcelliq-sequence-training.h5>.  
This file is too huge to be hosted on github. Put the file in ./data.
  5. Run **make**. After the compilation, two applications will be built: ./app/TrackingTrainer for training and ./app/TrackingPredictor for prediction.
- For example, you can try the training of the structured model for cell tracking by calling:
- ```
./apps/TrackingTrainer ./data/dcelliq-sequence-training.h5 ./data/event-configuration-cell.ini
```
- Training on this sample dataset usually takes 18 iterations to converge and the resulting learned feature weights will be displayed.

| Variable                  | Description                              |
|---------------------------|------------------------------------------|
| <b>VIGRA_INSTALL_PATH</b> | The root path of VIGRA.                  |
| <b>HDF5_INSTALL_PATH</b>  | The root path of HDF5.                   |
| <b>CPLEX_BASE_PATH</b>    | The root path of CPLEX.                  |
| <b>CPLEX_LIBFORMAT</b>    | The library format of the CPLEX library. |
| <b>CPLEX_SYSTEM</b>       | The system type of the CPLEX library.    |

Table 2: Description of CMake variables for BOT.

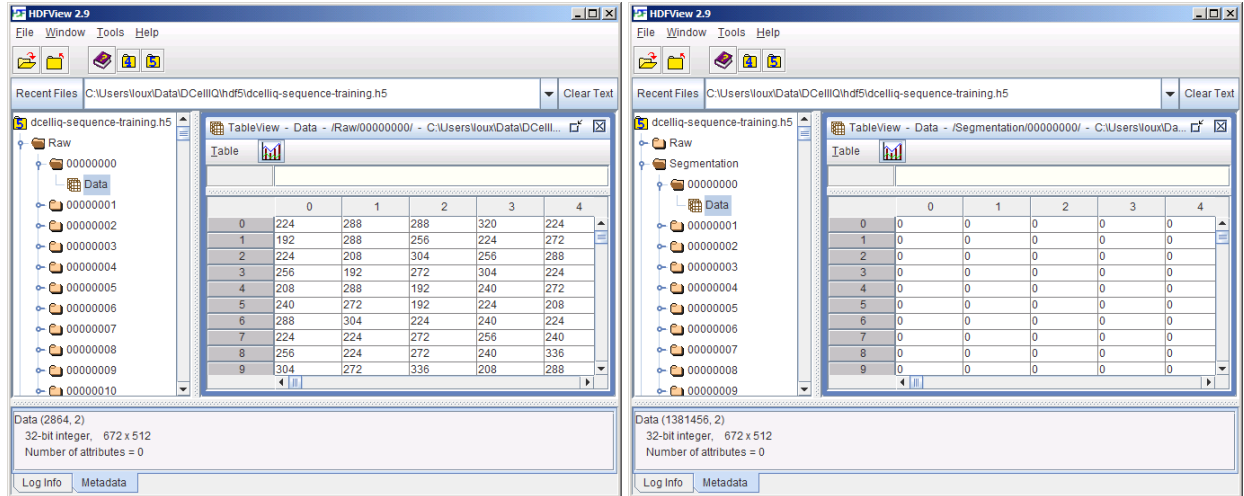


Figure 3: Save raw images and segmentation in HDF5.

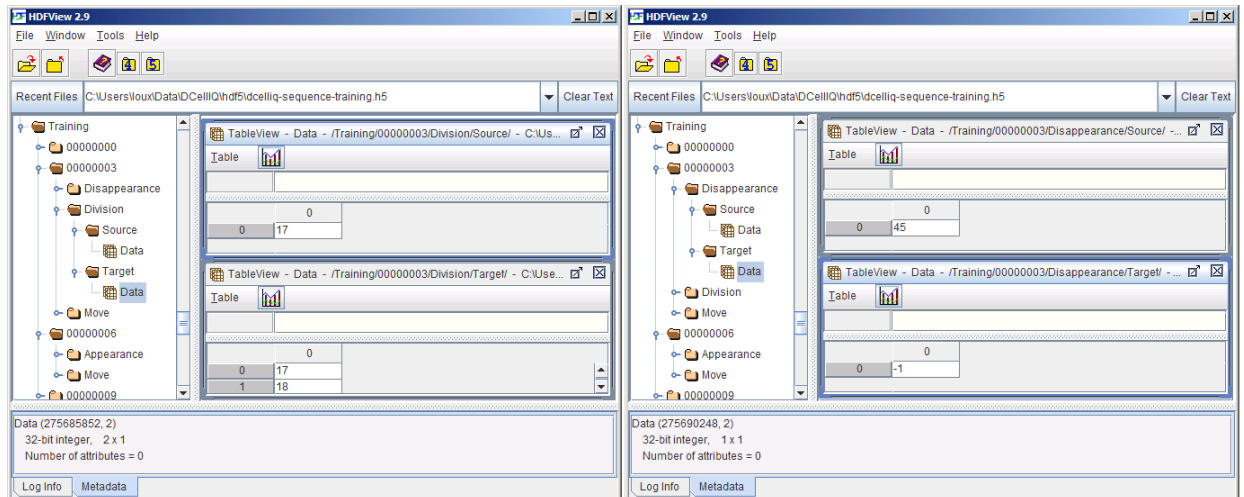


Figure 4: Save training data and tracking results in HDF5.

## 3.2 Data Preparation and Event Configuration

### 3.2.1 Data and Results in HDF5 Format

The two applications (./app/TrackingTrainer and ./app/TrackingTrainer) accept hdf5 file as input. The raw images and the segmentations are stored in a format shown in Fig. 3. For example, directory 00000000 indicates time point zero.

The tracking result and the training data are stored in the same format as in Fig. 4. The directory name (e.g. Division) must be identical to the one in the event definition ini file. Each event has a pair of subdirectories, namely Source and Target, which corresponds to associated cells from the first frame and the second frame, respectively. Note that for division one cell in the first frame is associated with two cells in the second frame (Fig. 4, left). And, for disappearance, there is no associated cell in the second frame so the id is -1 (Fig. 4, right).

The easiest way to understand this file structure is to explore the sample ./data/dcelliq-sequence-training.h5

338 using hdfview (freely available at <http://www.hdfgroup.org/hdf-java-html/hdfview/>).

### 339 3.2.2 Event Configuration in INI Format

340 You can using the default event configuration file `./data/event-configuration-cell.ini`, or construct you own. The format is very simple:

```
342 [Global]
343 k_neighbors = [how many nearest neighbors to be included]
344 distance_threshold = [maximum allowed distance as hypotheses thresholding]
345
346 [Event name, e.g. Move]
347 Pairing = [one to one association (put 1 1) or one to two association (put 1 2) or ...]
348 Feature1 = [the basic object feature] [joint feature, e.g. the measure] [feature weight]
349 Feature2 = [the basic object feature] [joint feature, e.g. the measure] [feature weight]
350 ...
351
352 [Event name, e.g. Division]
353 ...
```

### 354 3.3 A Complete Cell Tracking Example

355 The following code gives a concrete example of learning the parameter of high-dimensional features. It is extracted from the demo `test-TrackingTrainer.cxx`.

```
357 #include "InputOutput.hxx"
358 #include "HypothesisSpace.hxx"
359 #include "ObjectFeatureExtractor.hxx"
360 #include "AverageObject.hxx"
361 #include "SingletsGenerator.hxx"
362 #include "MultipletsGenerator.hxx"
363 #include "CPLEXSolverSystem.hxx"
364 #include "TrackingPredictor.hxx"
365 #include "vigra/hdf5impex.hxx"
366 #include "SolutionCoder.hxx"
367 #include "TrainingData.hxx"
368 #include "TrackingTrainer.hxx"
369 #include "HDF5ReaderWriter.hxx"
370
371 using namespace bot;
372
373 int main(int argc, char* argv[])
374 {
375     std::string filename(argv[1]);
376     // load the image sequence
377     std::vector<Matrix2D> images, segmentations;
378     TrainingData training;
379     HDF5ReaderWriter::load(filename, images, segmentations);
380 }
```

```

381 HDF5ReaderWriter::load(filename, images, segmentations);
382 std::cout << "****Loading the images/segmentations****" << std::endl;
383 HDF5ReaderWriter::load(filename, training);
384 std::cout << "****Loading the training data****" << std::endl;
385
386 // get the context
387 Context context(images);
388 std::cout << "****Computing the Context****" << std::endl
389     << context << std::endl << std::endl;
390
391 // load the configuration
392 HypothesisSpace space(argv[2]);
393 EventConfiguration conf = space.configuration();
394
395 // create singlets/multiplets and extract object features
396 std::cout << "****Extracting singlets and multiplets****" << std::endl;
397 SingletsSequence singlets_vec;
398 SingletsSequence avg_singlet_vec;
399 MultipletsSequence multiplets_vec;
400 SingletsGenerator singletsGenerator;
401 MultipletsGenerator multipletsGenerator(conf.k(), conf.d_max());
402 ObjectFeatureExtractor extractor(conf.get_feature_names(), context);
403 for (int32 indT = 0; indT < images.size(); indT++) {
404     // generate singlets and multiplets
405     Singlets singlets =
406         singletsGenerator(images[indT], segmentations[indT]);
407     Multiplets multiplets =
408         multipletsGenerator(images[indT], segmentations[indT], singlets);
409
410     // extract features for them
411     extractor(singlets);
412     extractor(multiplets);
413     // save
414     singlets_vec.push_back(singlets);
415     avg_singlet_vec.push_back(AverageObject::average(singlets));
416     multiplets_vec.push_back(multiplets);
417
418     std::cout << "#T=" << indT
419         << ": #singlets=" << singlets.size()
420         << ": #multiplets=" << multiplets.size() << std::endl;
421 }
422
423 // generate hypotheses and extract joint features
424 space(singlets_vec, avg_singlet_vec, multiplets_vec);
425 const std::vector<FramePair>& framepairs = space.framepairs();
426
427

```

```

428 // parse the training data
429 std::cout << "****Parsing the training data****" << std::endl;
430 SolutionCoder coder;
431 int32 nTr = training.times().size();
432 for (int32 ind = 0; ind < nTr; ind++) {
433     int32 time = training.times()[ind];
434     std::cout << "****time=" << time << "****" << std::endl;
435     const LabelAssociations& association = training.associations()[ind];
436
437     const std::vector<Event>& events = framepairs[time].events();
438     const Singlets& singlets1 = singlets_vec[time];
439     const Singlets& singlets2 = singlets_vec[time+1];
440     const Multiplets& multiplets1 = multiplets_vec[time];
441     const Multiplets& multiplets2 = multiplets_vec[time+1];
442
443     Solution solution;
444     coder.decode(
445         association,
446         events,
447         singlets1, singlets2,
448         multiplets1, multiplets2,
449         solution);
450     training.solutions().push_back(solution);
451 }
452
453 // start the training
454 TrackingTrainer trainer;
455
456 // start the training
457 TrackingTrainer trainer;
458 const std::vector<Matrix2D> null_vector;
459 std::vector<Matrix2D> weights = conf.weights(0.5);
460 std::string msg = trainer(training, framepairs, weights, true);
461 std::cout << "Training returns:" << msg << std::endl;
462 conf.weights() = weights;
463
464 // print the final weights
465 std::cout << "Learned weights:" << std::endl;
466 conf.print();
467
468 // printe intermediate results: weights, epsilons, losses
469 std::cout << "Weights:" << std::endl << trainer.weights() << std::endl << std::endl;
470 std::cout << "Epsilons:" << std::endl << trainer.epsilons() << std::endl << std::endl;
471 std::cout << "Losses:" << std::endl << trainer.losses() << std::endl << std::endl;
472
473 return EXIT_SUCCESS;
474 }

```

## Acknowledgement

We especially thank Bjoern Andres (University of Heidelberg), Christoph Straehle (University of Heidelberg) and Jan Funke (ETHZ) for their constructive comments on our manuscript and the code. We also thank Julian J. McAuley (The Australian National University) and Choon Hui Teo (Yahoo! Labs) for their suggestion on the implementation of the bundle method.

We are very grateful for the following software/code packages that BOT depends on:

1. The IBM Academic Initiative<sup>7</sup> allows us to access IBM ILOG CPLEX free of charge.
2. The SimpleIni library<sup>8</sup> allows us to efficiently read and write INI-style configuration files.
3. The FastEMD library<sup>9</sup> provides fast earth mover's distance computation.

## References

- [1] X. Lou and F. A. Hamprecht. Structured Learning for Cell Tracking. In *Neural Information Processing Systems (NIPS)*, 2011.
- [2] X. Lou, F. O. Kaster, M. S. Lindner, B. X. Kausler, U. Koethe, H. Jaenicke, B. Hoeckendorf, J. Wittbrodt, and F. A. Hamprecht. DELTR: Digital Embryo Lineage Tree Reconstructor. In *IEEE International Symposium on Biomedical Imaging: From Nano to Macro (ISBI)*, 2011. (Oral).
- [3] T. Kanade, Z. Yin, R. Bise, S. Huh, S. E. Eom, M. Sandbothe, and M. Chen. Cell Image Analysis: Algorithms, System and Applications. In *IEEE Workshop on Applications of Computer Vision (WACV)*, 2011.
- [4] D. Padfield, J. Rittscher, and B. Roysam. Coupled Minimum-Cost Flow Cell Tracking. In *Information Processing in Medical Imaging*, 2009.
- [5] Y. Rubner, C. Tomasi, and L. J. Guibas. A Metric for Distributions with Applications to Image Databases. In *IEEE International Conference on Computer Vision (ICCV)*, 1998.
- [6] C. Sommer, C. Strähle, U. Köthe, and F. A. Hamprecht. ilastik: Interactive Learning and Segmentation Toolkit. In *IEEE International Symposium on Biomedical Imaging: From Nano to Macro (ISBI)*, 2011.

---

<sup>7</sup><https://www.ibm.com/developerworks/university/academicinitiative/>

<sup>8</sup><http://code.jellycan.com/simpleini/>

<sup>9</sup><http://www.cs.huji.ac.il/~ofirpele/FastEMD/code/>