Xinghua Lou, Bernhard Kausler, Ullrich Koethe, Fred Hamprecht
HCI, IWR, University of Heidelberg, Heidelberg 69115, Germany
May 18, 2011

BOT: A C++ Library for Biomedical Object Tracking
Design Document and User Guide

Abstract

Biomedical Object Tracking (BOT) is a C++ library for tracking massive objects in time lapse biological experiments. It features (I) object association based pairwise tracking, (II) interactive structured learning for optimal feature parametrization, (III) generic and extensible feature sets, (IV) configurable workflow for different applications (e.g. tracking cells and tracking worm) and (V) a user-friendly GUI based on the Interactive Learning and Segmentation Toolkit (ilastik). In this report, we provide the technical details on the implementation of the aforementioned algorithms, features and configurable workflows as well as the integration of this library with ilastik.

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 drew much attention due to its high efficiency in solving large-scale, complex tracking problems such as lineage-tree reconstruction [4] and cell culture study [5, 2]. 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 [3]. This approach not only boosts object association by allowing for the use of high-dimensional features for better discriminative power 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 the factory pattern;

Diversity Configurable problem setup for supporting diverse applications;

Learning Structured learning for parameter optimization;

GUI A user-friendly GUI by integration with the Interactive Learning and Segmentation Toolkit (ilastik, http://www.ilastik.org/) [7].

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. From the resulting labeling, BOT accepts value zero as background and considers all pixels with identical label (other than zero) a single object.

Regarding solver systems, BOT relies on at least an integer linear programming (ILP) solver for predicting the tracking. To perform the structured learning for parameter optimization, quadratic programming

(QP) or linear programming (LP) solver is required. By default, BOT uses IBM ILOG CPLEX[1] which provides solvers for ILP, QP and LP problems. It is nevertheless possible and easy to interface BOT with alternative solvers such as lpsolve[2] and Gurobi[3]. 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 a general 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 sophisticated biomedical phenomena before we introduce multiplet. A multiplet is a combination of several singlets (usually restricted to a neighborhood system). Multiplet captures behaviors 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 based class for class `Singlet` and class `Multiplet`. The base class `Object` employs a sparse representation (i.e. a point cloud) of the pixels within, including its 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, which bears a particular biological or technical interpretation. For example, cell division is a biological event and the involved objects are the father cell from the first frame and the two daughter cells from the second one. For another example, an over-segmentation in the second frame is an event due to a technical reason, i.e. the instability of the segmentation algorithm.

Multiple events can be defined that enables addressing 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 the segmentation flaws, i.e. split for over-segmentation and merge for under-segmentation. Pairing specifies the types of objects involved, e.g. $1 \to 1$ for singlet to singlet, $1 \to 2$ for singlet to multiplet, and $1 \to \emptyset$ for singlet to *nothing* (considered as a special type of "object"). BOT uses class `EventConfiguration` to import the definitions from an INI file[4] 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	$1 \to 1$	Change of size	Spatial displacement	...
Division	$1 \to 2$	Angle pattern	Father cell intensity	...
Appearance	$\emptyset \to 1$	Distance to the border	Overlap with the border	...
Disappearance	$1 \to \emptyset$	Distance to the border	Overlap with the border	...
Split	$1 \to 2$	Shape compactness	Mass evenness	...
Merge	$2 \to 1$	Shape compactness	Mass evenness	...

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

[1] http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/
[2] http://sourceforge.net/projects/lpsolve
[3] http://www.gurobi.com
[4] http://en.wikipedia.org/wiki/INI_file

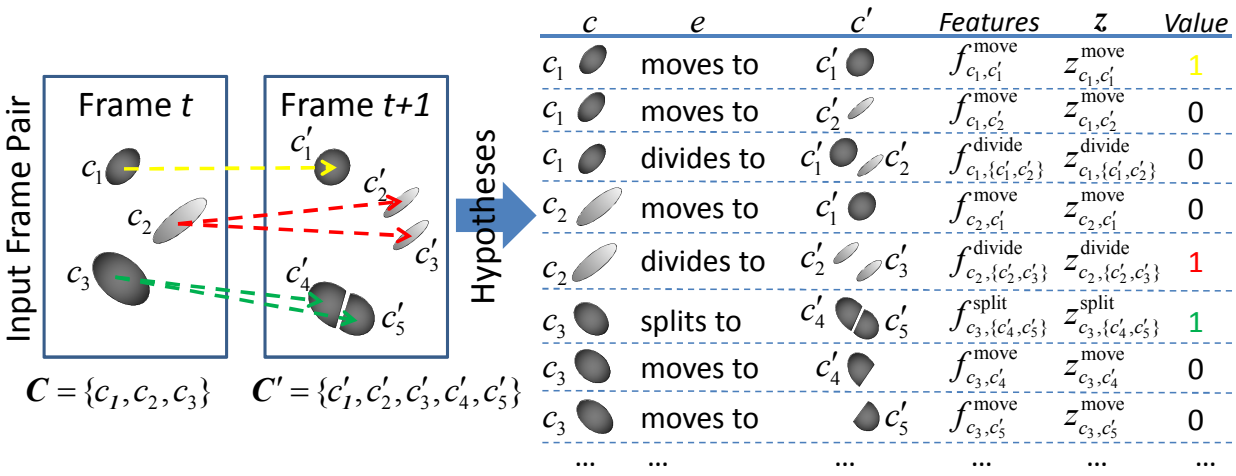

Figure 1: Toy example: two frames of objects and a list of the possible associations as hypotheses. One particular interpretation of the scene is indicated by colored arrows (left) or equivalently by a configuration of binary indicator variables z (rightmost column in table).

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 intended to respect the speed limitation of object movement (e.g. cell). Appropriate joint feature shall be selected according to the object feature, such as using earth mover's distance [6] for comparing two intensity histograms (see class `MeasureEarthMoversDistance`). Note that a joint feature does not have to be 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 it mean intensity 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, and sometime 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) is unique respectively for singlets and multiplets in the same frame.

```
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_;

    /* 2D matrix of the labels of the pixels */
    Matrix2D labels_;
};
```

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:

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

Features are always represented using type `Matrix2D`.

2.4 Configurable Workflow

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. It shall be unique for the name but not necessary for the pairing (for example, division and split). 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:

```
[Division]
Pairing = 1 2
Feature1 = ObjectFeatureIntensitySum MeasureNormalizedEuclideanDistance -0.4680
Feature2 = ObjectFeatureCenters MeasureAnglePattern -1.3055
Feature3 = ObjectFeatureIntensityMean MeasureAppointmentLeft 0.9591
Feature4 = ObjectFeatureEccentricity MeasureAppointmentLeft -0.9441
Feature5 = ObjectFeatureVolumeEvenness MeasureAppointmentRight 1.6322
```

```
Feature6 = ObjectFeatureShapeCompactness MeasureAppointmentRight −0.7221
Feature7 = ObjectFeatureMassEvenness MeasureAppointmentRight −0.8504
```

2.4.2 Dynamic Feature Loading

Each event definition shall 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 hold for the measure, 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 [1] 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.

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 shall 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 the 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 takes an std::string as the class identifier, 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) {
        fea = new ObjectFeaturePosition();
    }
    else if ...
}
```

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

2.5.2 Joint Features

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

2.5.3 Solver System

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

```
class SolverSystem {
public:
    /*! Solve a binary integer linear programminig (binary ilp) problem with
     *  given equality and inequality constraints. In particular, it solves
     *  a problem as follows:
     *          max     f'*x
     *          s.t.    Aineq*x <=  bineq
     *                  Aeq*x   =   beq
     *                  each variable in x is binary
     *  @param f The coefficient of the objective function
     *  @param Aineq The inequality constraints: Aineq * x <= bineq
     *  @param bineq The inequality constraints: Aineq * x <= bineq
     *  @param Aeq The equality constraints: Aineq * x = bineq
     *  @param beq The equality constraints: Aineq * x = bineq
     *  @param x0 The initial solution
     *  @param x The solution
     *  @param msg The return message
     */
    virtual std::string solve_bilp(
        const Matrix2D& f,
        const Matrix2D& Aineq, const Matrix2D& bineq,
        const Matrix2D& Aeq, const Matrix2D& beq,
        const Matrix2D& x0, Matrix2D& x) const = 0;

    /*! Solve a quadaratic programminig (qp) problem with given equality and
```

```
     *  inequality constraints and bounds. In particular, it solve:
     *      min      0.5*x'*H*x+f*x
     *      st.      Aineq*x <= bineq
     *               Aeq*x    = beq
     *               lb <= x <= ub
     *  @param H Double matrix for objective function (quadratic term)
     *  @param f Double matrix (vector) for objective function (linear term)
     *  @param Aineq The inequality constraints: Aineq * x <= bineq
     *  @param bineq The inequality constraints: Aineq * x <= bineq
     *  @param Aeq The equality constraints: Aineq * x = bineq
     *  @param beq The equality constraints: Aineq * x = bineq
     *  @param lb The lower bound
     *  @param ub The upper bound
     *  @param x0 The initial solution
     *  @param x The solution
     *  @return A std::string as the message of the solution status
     */
    virtual std::string solve_qp(
        const Matrix2D& H, const Matrix2D& f,
        const Matrix2D& Aineq, const Matrix2D& bineq,
        const Matrix2D& Aeq, const Matrix2D& beq,
        const Matrix2D& lb, const Matrix2D& ub,
        const Matrix2D& x0, Matrix2D& x) const = 0;
};
```

2.6 Tracking Prediction and Parameter Learning

2.6.1 Methodology

To learn more about the underlying methodology of tracking prediction and structured learning for parameter optimization, please refer to our manuscript [3]. In BOT, they are implemented in class `TrackingPredictor` and class `TrackingLearner`, respectively. To learn more, simply follow the example in Section 3.2.

2.6.2 Two Representations of the Tracking Result

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

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

Obviously, `Solution` is dependent on the hypothesis space and is therefore not suitable for storage. This is simply because one usually does not want to store the entire hypothesis space and the result shall be fully interpretable using the images and segmentations as its raw information. We therefore use another representation termed `LabelAssociation` which directly stores the labels from the segmentation or detection in a pair of matrices as the source and the target:

```
// associating the segmentation labels
struct LabelAssociation {
    std::string name;
    Matrix2D source;
    Matrix2D target;
};

```

```
// a vector of label associations (for multiple events)
typedef std::vector<LabelAssociation > LabelAssociations;
```

2.7 Limitations

We declare two major methodological limitations of BOT:

1. BOT is currently restricted to pairwise object association. We are now investigating a global object association approach that hopefully will bring better tracking performance.
2. The multiplet only consists of two singlets. We are investigating possible improvement such as replacing the singlet/multiplet generation with hierarchical segmentation.

3 User Guide

3.1 Installation and Compilation

BOT and several of its dependencies use CMake[5] as the building system. Make sure CMake is correctly install (minimum version 2.6) before processing further.

3.1.1 Install and Compile Dependencies

1. Download and install VIGRA from http://hci.iwr.uni-heidelberg.de/vigra/. The installation instructions can be found at http://hci.iwr.uni-heidelberg.de/vigra/doc/vigra/Installation.html.
2. Download and install HDF5 from http://www.hdfgroup.org/ftp/HDF5/current/src/. The installation instructions can be found inside the package.
3. To access IBM ILOG CPLEX, a license is required. Detailed guidelines can be found at http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/.

3.1.2 Install and Compile BOT

1. Download the BOT source code from http://hci.iwr.uni-heidelberg.de/MIP/Software/.
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.
4. Run **make** and, after the compilation, run the demos in [BOT root]/tests/ such as **./test-TrackingTrainer** to perform the parameter learning.

Variable	Description
VIGRA_INSTALL_PATH	The root path of VIGRA.
HDF5_INSTALL_PATH	The root path of HDF5.
CPLEX_BASE_PATH	The root path of CPLEX.
CPLEX_LIBFORMAT	The library format of the CPLEX library.
CPLEX_SYSTEM	The system type of the CPLEX library.

Table 2: Description of CMake variables for BOT.

[5] http://www.cmake.org/

3.2 A Complete Cell Tracking Example

The following code gives a concrete example of learning the optimal weights. It is extracted from the demo test-TrackingTrainer.cxx.

```
#include "InputOutput.hxx"
#include "HypothesisSpace.hxx"
#include "ObjectFeatureExtractor.hxx"
#include "AverageObject.hxx"
#include "SingletsGenerator.hxx"
#include "MultipletsGenerator.hxx"
#include "CPLEXSolverSystem.hxx"
#include "TrackingPredictor.hxx"
#include "vigra/hdf5impex.hxx"
#include "SolutionCoder.hxx"
#include "TrainingData.hxx"
#include "TrackingTrainer.hxx"
#include "HDF5ReaderWriter.hxx"

using namespace bot;

int main()
{
    std::string filename("../data/dcelliq-sequence-training.h5");
    // load the image sequence
    std::vector<Matrix2D > images, segmentations;
    TrainingData training;
    HDF5ReaderWriter::load(filename, images, segmentations);
    std::cout << "****Loading the images/segmentations****" << std::endl;
    HDF5ReaderWriter::load(filename, training);
    std::cout << "****Loading the training data****" << std::endl;

    // get the context
    Context context(images);
    std::cout << "****Computing the Context****" << std::endl
        << context << std::endl << std::endl;

    // load the configuration
    HypothesisSpace space("../data/event-configuration-cell.ini");
    EventConfiguration conf = space.configuration();

    // create singlets/muliplets and extract object features
    std::cout << "****Extracting singlets and multiplets****" << std::endl;
    SingletsSequence singlets_vec;
    SingletsSequence avg_singlet_vec;
    MultipletsSequence multiplets_vec;
    SingletsGenerator singletsGenerator;
    MultipletsGenerator multipletsGenerator(conf.k(), conf.d_max());
    ObjectFeatureExtractor extractor(conf.get_feature_names(), context);
    for (int32 indT = 0; indT < images.size(); indT ++) {
        // generate singlets and multiplets
        Singlets singlets = singletsGenerator(
            images[indT], segmentations[indT]);
        Multiplets multiplets = multipletsGenerator(
            images[indT], segmentations[indT], singlets);
```

```
371
372         // extract features for them
373         extractor(singlets);
374         extractor(multiplets);
375         // save
376         singlets_vec.push_back(singlets);
377         avg_singlet_vec.push_back(AverageObject::average(singlets));
378         multiplets_vec.push_back(multiplets);
379
380         std::cout << "#T = " << indT
381             << ": #singlets = " << singlets.size()
382             << ": #multiplets = " << multiplets.size() << std::endl;
383
384     }
385
386     // generate hypotheses and extract joint features
387     space(singlets_vec, avg_singlet_vec, multiplets_vec);
388     const std::vector<FramePair>& framepairs = space.framepairs();
389
390     // parse the training data
391     std::cout << "****Parsing the training data****" << std::endl;
392     SolutionCoder coder;
393     int32 nTr = training.times().size();
394     for (int32 ind = 0; ind < nTr; ind++) {
395         int32 time = training.times()[ind];
396         std::cout << "****time = " << time << "****" << std::endl;
397         const LabelAssociations& association = training.associations()[ind];
398
399         const std::vector<Event>& events = framepairs[time].events();
400         const Singlets& singlets1 = singlets_vec[time];
401         const Singlets& singlets2 = singlets_vec[time+1];
402         const Multiplets& multiplets1 = multiplets_vec[time];
403         const Multiplets& multiplets2 = multiplets_vec[time+1];
404
405         Solution solution;
406         coder.decode(
407             association,
408             events,
409             singlets1, singlets2,
410             multiplets1, multiplets2,
411             solution);
412         training.solutions().push_back(solution);
413     }
414
415     // start the training
416     TrackingTrainer trainer;
417     const std::vector<Matrix2D> null_vector;
418     std::vector<Matrix2D> weights = conf.weights(0.5);
419     std::string msg = trainer(training, framepairs, weights, true);
420     std::cout << "Training returns: " << msg << std::endl;
421     conf.weights() = weights;
422
423     // print the final weights
424     std::cout << "Learned weights: " << std::endl;
```

```
    conf.print();

    // printe intermediate results: weights, epsilons, losses
    std::cout << "Weights: " << std::endl << trainer.weights()
        << std::endl << std::endl;
    std::cout << "Epsilons: " << std::endl << trainer.epsilons()
        << std::endl << std::endl;
    std::cout << "Losses: " << std::endl << trainer.losses()
        << std::endl << std::endl;

    return 0;
}
```

Acknowledgement

We especially thank Bjoern Andres (University of Heidelberg) for his constructive comments on our manuscript.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[6] allows us to access IBM ILOG CPLEX free of charge.
2. The SimpleIni library[7] allows us to efficiently read and write INI-style configuration files.
3. The FastEMD library[8] provides fast earth mover's distance computation.

References

[1] R. Johnson, E. Gamma, R. Helm, and J. Vlissides. Design patterns: Elements of reusable object-oriented software. *Addison-Wesley*, 1995.

[2] T. Kanade, Z. Yin, R. Bise, S. Huh, S. E. Eom, M. Sandbothe, and M. Chen. Cell Image Analysis: Algorithms, System and Applications. In *WACV*, 2011.

[3] X. Lou and F. A. Hamprecht. Structured Learning for Cell Tracking. 2011. (Preprint).

[4] 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 *ISBI*, 2011.

[5] D. Padfield, J. Rittscher, and B. Roysam. Coupled Minimum-Cost Flow Cell Tracking for High-Throughput Quantitative Analysis. *Med Image Anal*, 2010.

[6] Y. Rubner, C. Tomasi, and L. J. Guibas. A Metric for Distributions with Applications to Image Databases. In *ICCV*, 1998.

[7] C. Sommer, C. Straehle, U. Koethe, and F. A. Hamprecht. "ilastik: Interactive learning and segmentation toolkit". In *ISBI*, 2011.

[6] https://www.ibm.com/developerworks/university/academicinitiative/
[7] http://code.jellycan.com/simpleini/
[8] http://www.cs.huji.ac.il/ ofirpele/FastEMD/code/