Modular Tracking Framework: A unified approach to registration based tracking

Abhineet Singh[a,**], Martin Jagersand[a]

[a]*Department of Computing Science, University of Alberta, Edmonton T6G2R3, Canada*

ABSTRACT

This paper presents a modular, extensible and highly efficient open source framework for registration based tracking called Modular Tracking Framework (MTF). Targeted at robotics applications, it is implemented entirely in C++ and designed from the ground up to easily integrate with systems that support any of several major vision and robotics libraries including OpenCV, ROS, ViSP and Eigen. It implements more methods, is faster, and more precise than other existing systems. Further, the theoretical basis for its design is a new way to conceptualize registration based trackers that decomposes them into three constituent sub modules - Search Method (SM), Appearance Model (AM) and State Space Model (SSM). In the process, we integrate many important advances published after Baker & Matthews' landmark work in 2004. In addition to being a practical solution for fast and high precision tracking, MTF can also serve as a useful research tool by allowing existing and new methods for any of the sub modules to be studied better. When a new method is introduced for one of these, the breakdown can help to experimentally find the combination of methods for the others that is optimum for it. By extensive use of generic programming, MTF makes it easy to plug in a new method for any of the sub modules so that it can not only be tested comprehensively with existing methods but also become immediately available for deployment in any project that uses the framework. With 16 AMs, 11 SMs and 13 SSMs implemented already, MTF provides over 2000 distinct single layer trackers. It also allows two or more of these to be combined together in several ways to create a practically unlimited variety of novel multi layer trackers.

1. Introduction

Fast and high precision visual tracking is crucial to the success of several robotics applications like visual servoing and autonomous navigation. Image patch trackers are usually classified as either registration-based (RBT) or online learning and detection based trackers (OLT). In recent years OLTs have become popular (Wu et al., 2013; Kristan et al., 2015) due to their robustness to changes in the object appearance. This makes OLT suited to long term tracking. However, OLTs are neither precise, nor fast, and commonly estimate only 2 DOF image translation or, adding scale, 3 DOF. RBTs (Sec. 2.2)

are often more suitable for robotics applications, being several times faster and capable of estimating higher degrees-of-freedom (DOF) transformations like 6 DOF affine and 8 DOF homography to pixel precision. Using visual servoing or a calibrated camera, this allows precise full 6 DOF 3D translation and rotation alignments in applications such as robot arm control and augmented reality.

Though several major advances have been made in this domain since the original Lucas Kanade (LK) tracker was introduced almost thirty five years ago (Lucas and Kanade, 1981), efficient open source implementations of recent trackers are surprisingly difficult to find. In fact, the only such tracker offered by the popular OpenCV library (Bradski, 2000), uses a pyramidal implementation of the original 1981 algorithm (Bouguet, 2000). Similarly, the more recent ROS library (Quigley et al., 2009) currently does not have any package that implements a modern RBT. The XVision system (Hager and Toyama, 1998) did introduce a full tracking software framework including a

---

Abbreviations: **MTF**: Modular Tracking Framework; **SM**: Search Method; **AM**: Appearance Model; **SSM**: State Space Model; **ILM**: ILumination Model; **RBT**: Registration-Based Tracker; **OLT**: Online Learning and detection based Tracker; **LK**: Lucas Kanade; **DOF**: Degrees Of Freedom

[**]Corresponding author: Tel.: +1-587-596-0470;

e-mail: asingh1@ualberta.ca (Abhineet Singh)

video pipeline. However, it has not been updated for a long time and mainly implements variants of Hager's 1998 algorithm (Hager and Belhumeur, 1998). Even the fairly recent MRPT library (Harris and Conrad, 2011) includes only a version of the original LK tracker apart from a low DOF particle filter based tracker that is too imprecise and slow to be considered relevant for our target applications. The Visual Servoing Platform (ViSP) library (Marchand et al., 2005) does provide somewhat similar tracking functionality as the proposed framework but is more limited, slower and not modular, hence harder to extend with new AMs, SMs and SSMs (see Sec. 6).

To address the need for a tracking library targeted at high DOF robotics and augmented reality applications, we introduce Modular Tracking Framework (**MTF**) (Singh, 2018) - a generic system for RBT that provides highly efficient implementations for a large subset of trackers introduced in literature to date and is designed to be easily extensible with additional methods, and compatible with popular pachages such as ROS, OpenCV and ViSP.

MTF conceptualizes an RBT as being composed of three semi independent sub modules - Search Method (**SM**), Appearance Model (**AM**) and State Space Model (**SSM**). SM is treated here very generally as a way to use the functionality in AM and SSM - through a well defined interface - to solve the tracking problem. The three sub modules are semi-independent in the sense that the functional specification of the interface between them is general enough to allow any method implementing one of these to be combined unchanged with many combinations of methods for the other two.

Such an approach can help to address another urgent need in this field - that of unifying the myriad of contributions made in the last three decades so they can be better understood. When a new RBT is published, it often contributes to only one or two of these sub modules while using existing methods for the rest (Singh et al., 2016, 2017). In such cases, the modular decomposition can provide a way within which the contributions of the new tracker can be clearly demarcated and thus studied better. By following this decomposition closely through extensive use of generic programming, MTF provides a convenient interface to plug in a new method for any sub module and test it against existing methods for the other two. This will not only help to compare the new method against existing ones in a more comprehensive way but also make it immediately available to any project that uses MTF. To facilitate the latter, MTF provides a simple ROS interface for seamless integration with robotics systems along with interfaces for Python and MATLAB to aid its use in research applications.

Further, MTF is designed to allow two or more trackers to be combined in several ways (Sec. 3) to create composite multi-layer trackers that perform better than any of their constituents. This approach has been shown to be promising (Zhang et al., 2015; Singh et al., 2017; Singh, 2017) in creating trackers that are robust to challenges like illumination changes, fast motion and occlusions by combining the advantages provided by their constituents. This is also a much easier way to improve tracking performance than designing new algorithms for individual sub modules to handle specific challenges. An example of such

a composite tracker that can be considered as the current state of the art in RBT is the LMES tracker (Singh, 2017) which has a two layer composite Search Method (SM) created by using LMS (Sec. 2.4) and ESM (Sec. 2.3) in cascade (Sec. 3.2) as shown in Fig. 4. This tracker benefits from the larger search radius of stochastic SMs as well as the high precision of gradient based SMs to offer superior performance over either.

To summarize, following are the main contributions of this work:

- Provide a unifying formulation for RBT that can be seen as an extension of the framework reported in Baker and Matthews (2004) with newer methods.

- Present a fast tracking library for robotics applications that is based on this formulation and is also easy to extend due to its modular design.

  - It currently has 16 AMs, 11 SMs and 13 SSMs implemented (Fig. 3) and, since each combination of these methods constitutes a distinct tracker, it can be used to run over 2000 single layer trackers.

  - Any two or more of these trackers can be combined together in a multitude of ways (Sec. 3) to provide virtually unlimited opportunities to create novel multi layer composite trackers.

This paper extends a preliminary version of this work (Singh and Jagersand, 2017) by providing detailed function specifications for the main classes in MTF (Sec. 3), examples of existing SMs recast to fit the proposed framework (Sec. 4), examples of composite SMs (Sec. 3.2) and several extensions to the formulation of Baker & Matthews (Baker and Matthews, 2004) (appendix).

The rest of this paper is organized as follows: Section 2 introduces the mathematical basis for the design of MTF while section 3 describes the class structure of MTF along with specifications for important functions. Section 4 presents several SMs as examples of using the functionality described in section 3 to implement the theory of section 2. Section 5 presents several use cases for MTF while section 6 provides performance comparison of MTF with another existing library for RBT as well as with state of the art OLTs. Finally, section 7 concludes with promising avenues for future extensions to this work. The paper includes more technical implementation details than typical in the literature. We feel this is important, as it ultimately affects performance. Someone interested in the broad strokes can read the technical sections cursorily. Someone just wanting to use the implemented trackers can focus on sections 5 and 6, while those aiming to do research in tracking systems mainly benefit from the design sections 3 and 4.

2. Theoretical Background

2.1. Notation

Let $I_t : \mathbb{R}^2 \mapsto \mathbb{R}$ refer to an image captured at time t. I_t is treated as a smooth function of real values using sub pixel interpolation (Dame and Marchand, 2010) for non integer locations. The patch corresponding to the tracked object's location
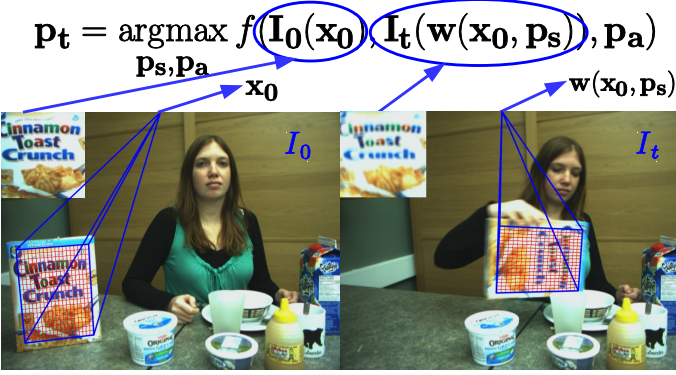
$$\mathbf{p_t} = \operatorname*{argmax}_{\mathbf{p_s},\mathbf{p_a}} f(\mathbf{I_0(x_0)}, \mathbf{I_t(w(x_0,p_s))}, \mathbf{p_a})$$

Fig. 1. Two frames from a sequence showing the different components of RBT. The grid of points x where pixel values are extracted are shown in red. The function w warps the red grid on I_t to match (align with) the template I_0 as shown in the top left corners. For better visibility, the grid is sampled at 25×25 though higher resolutions may be used in practice.

in I_t is denoted by $\mathbf{I_t(x_t)} \in \mathbb{R}^N$ where $\mathbf{x_t} = [\mathbf{x_{1t}}, ..., \mathbf{x_{Nt}}]$ with $\mathbf{x_{kt}} = [x_{kt}, y_{kt}]^T \in \mathbb{R}^2$ being the Cartesian coordinates of pixel k.

Further, $\mathbf{w(x, p_s)} : \mathbb{R}^2 \times \mathbb{R}^S \mapsto \mathbb{R}^2$ denotes a warping function of S parameters that represents the set of allowable image motions of the tracked object by specifying the deformations that can be applied to $\mathbf{x_0}$ to align $\mathbf{I_t(x_t)} = \mathbf{I_t(w(x_0, p_{st}))}$ with $\mathbf{I_0(x_0)}$. Examples of $\mathbf{w}$ include homography, affine, similitude, isometry and translation (Szeliski, 2006).

Finally $f(\mathbf{I^*}, \mathbf{I^c}, \mathbf{p_a}) : \mathbb{R}^N \times \mathbb{R}^N \times \mathbb{R}^A \mapsto \mathbb{R}$ is a function of A parameters that measures the similarity between two patches - the reference or template patch $\mathbf{I^*}$ and a candidate patch $\mathbf{I^c}$. Examples of f with $A = 0$ include sum of squared differences (SSD) (Hager and Belhumeur, 1998), sum of conditional variance (SCV) (Richa et al., 2011), normalized cross correlation (NCC) (Scandaroli et al., 2012), mutual information (MI) (Dame and Marchand, 2010) and cross cumulative residual entropy (CCRE) (Rogerio Richa, 2012). So far, the only examples with $A \neq 0$, to the best of our knowledge, are those with an illumination model (**ILM**) (Silveira and Malis, 2007; Bartoli, 2008) where f is expressed as $f(\mathbf{I^*}, g(\mathbf{I^c}, \mathbf{p_a}))$ with $g : \mathbb{R}^N \times \mathbb{R}^A \mapsto \mathbb{R}^N$ accounting for differences in lighting conditions under which I_0 and I_t were captured.

2.2. Registration based tracking

RBT aligns an image template I_0 with the corresponding pixels in the current image I_t, see Fig 1. Mathematically, the goal is to find a warp function $\mathbf{w(p)}$ that transform the pixel coordinates to match the image patches. If perfect intensity constancy was possible then $I_0 = I_t(\mathbf{w(p)})$. In real world videos, the appearance of the template will vary, so equality is not possible, and one of several possible Appearance Models - AM, denoted with f above, are used for the similarity metric.

Using the notation in the last section, RBT can be formulated (Eq. 1) as a search problem where the goal is to find the optimal parameters $\mathbf{p_t} = [\mathbf{p_{st}}, \mathbf{p_{at}}] \in \mathbb{R}^{S+A}$ that maximize the similarity, measured by f, between the target patch $\mathbf{I^*} = \mathbf{I_0(x_0)}$ and the

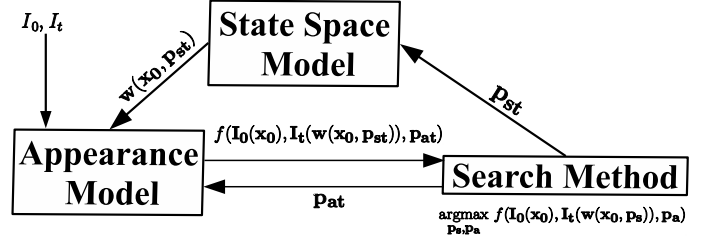warped image patch $\mathbf{I^c} = \mathbf{I_t(w(x_0, p_t))}$, that is,

$$\mathbf{p_t} = \operatorname*{argmax}_{\mathbf{p_s},\mathbf{p_a}} f(\mathbf{I_0(x_0)}, \mathbf{I_t(w(x_0,p_s))}, \mathbf{p_a}) \qquad (1)$$

As has been observed before (Szeliski, 2006; Rogerio Richa, 2012), this formulation gives rise to an intuitive way to decompose the tracking task into three modules - the similarity metric f, the warping function $\mathbf{w}$ and the optimization approach. We refer to these respectively as Appearance Model- AM, State-Space Model - SSM and Search Method - SM. These can be designed to be semi independent in the sense that any given optimizer can be applied unchanged to several combinations of methods for the other two modules which in turn interact only through a well defined and consistent interface.

A pictorial representation of the meanings of various components in this equation is shown in Fig. 1. As has been observed before (Szeliski, 2006; Rogerio Richa, 2012), this formulation gives rise to an intuitive way to decompose the tracking task into three modules - the similarity metric f, the warping function $\mathbf{w}$ and the optimization approach. These can be designed to be semi independent in the sense that any given optimizer can be applied unchanged to several combinations of methods for the other two modules which in turn interact only through a well defined and consistent interface.

Fig. 2 shows the effective flow of information between the three sub modules though in practice SM serves as the interface between AM and SSM, which do not interact directly. Following steps are performed for each frame I_t:

1. SM computes the optimum parameters for $\mathbf{w}$ and f and passes these respectively to the SSM and AM.
2. SSM then warps the initial grid points $\mathbf{x_0}$ using these parameters and passes the resultant points $\mathbf{w(x_0, p_{st})}$ to the AM
3. AM extracts pixel values at these points and computes the similarity of the resultant patch with the template using $\mathbf{p_{at}}$ which it then passes back to the SM.
4. SM uses this similarity to find the parameters $\mathbf{p_{t+1}}$ that maximize it for the next frame I_{t+1}.

2.3. Gradient Descent and the Chain Rule

Though several types of SMs have been reported in literature, gradient descent based methods (Lucas and Kanade, 1981) are most widely used due to their speed and simplicity. As mentioned in (Baker and Matthews, 2004), the LK tracker can be

Fig. 2. Decomposition of RBT showing the interaction between the resultant sub modules

formulated in four different ways depending on which image is searched for the warped patch - I_0 or I_t - and how $\mathbf{p_s}$ is updated in each iteration - additive or compositional. The four resultant formulations are thus called Forward Additive (**FALK**) (Lucas and Kanade, 1981), Inverse Additive (**IALK**) (Hager and Belhumeur, 1998), Forward Compositional (**FCLK**) (Shum and Szeliski) and Inverse Compositional (**ICLK**) (Baker and Matthews, 2001). There is also a more recent approach called Efficient Second order Minimization (**ESM**) (Benhimane and Malis, 2007) that tries to make the best of both ICLK and FCLK by using information from both I_0 and I_t.

What all these methods have in common is that they solve Eq 1 by estimating an incremental update $\Delta\mathbf{p_t}$ to the optimal parameters $\mathbf{p_{t-1}}$ at time $t-1$ using some variant of the Newton method as:

$$\Delta\mathbf{p_t} = -\hat{\mathbf{H}}^{-1}\hat{\mathbf{J}}^T \tag{2}$$

where $\hat{\mathbf{J}}$ and $\hat{\mathbf{H}}$ respectively are estimates for the Jacobian $\mathbf{J} = \partial f/\partial\mathbf{p}$ and the Hessian $\mathbf{H} = \partial^2 f/\partial\mathbf{p}^2$ of f w.r.t. $\mathbf{p}$. For any formulation that seeks to decompose this class of trackers (among others) in the aforementioned manner, the chain rule for first and second order derivatives is indispensable and the resultant decompositions for $\mathbf{J}$ and $\mathbf{H}$ are given by Eqs. 3 and 4 respectively, assuming $A = 0$ (or $\mathbf{p} = \mathbf{p_s}$) for simplicity.

$$\mathbf{J} = \frac{\partial f(\mathbf{I}(\mathbf{w}(\mathbf{p})))}{\partial\mathbf{p}} = \frac{\partial f}{\partial\mathbf{I}}\nabla\mathbf{I}\frac{\partial\mathbf{w}}{\partial\mathbf{p}} \tag{3}$$

$$\mathbf{H} = \frac{\partial\mathbf{I}}{\partial\mathbf{p}}^T\frac{\partial^2 f}{\partial\mathbf{I}^2}\frac{\partial\mathbf{I}}{\partial\mathbf{p}} + \frac{\partial f}{\partial\mathbf{I}}\frac{\partial^2\mathbf{I}}{\partial\mathbf{p}^2} \tag{4}$$

with $\frac{\partial\mathbf{I}}{\partial\mathbf{p}} = \nabla\mathbf{I}\frac{\partial\mathbf{w}}{\partial\mathbf{p}}$ and $\frac{\partial^2\mathbf{I}}{\partial\mathbf{p}^2} = \frac{\partial\mathbf{w}}{\partial\mathbf{p}}^T\nabla^2\mathbf{I}\frac{\partial\mathbf{w}}{\partial\mathbf{p}} + \nabla\mathbf{I}\frac{\partial^2\mathbf{w}}{\partial\mathbf{p}^2}$. It follows that the AM computes terms involving $\mathbf{I}$ and f ($\nabla\mathbf{I}$, $\nabla^2\mathbf{I}$, $\partial f/\partial\mathbf{I}$ and $\partial^2 f/\partial\mathbf{I}^2$) while the SSM computes those with $\mathbf{w}$ ($\partial\mathbf{w}/\partial\mathbf{p}$, $\partial^2\mathbf{w}/\partial\mathbf{p}^2$). Further, these generic expressions do not give the whole scope of the decompositions since the exact forms of $\hat{\mathbf{J}}$ and $\hat{\mathbf{H}}$ as well as the way these are split vary for different variants of LK. The reader is referred to (Baker and Matthews, 2004) for more details though formulations relevant to the functions in MTF (Tables 1 and 3), including several extensions to (Baker and Matthews, 2004), are also presented in the appendix.

2.4. Stochastic Search

A limitation of gradient descent type SMs is that they are prone to getting stuck in local maxima of f especially when the object's appearance changes due to factors like occlusions, motion blur or illumination variations. An alternative approach to avoid this problem is to use stochastic search so as to cover a larger portion of the search space of $\mathbf{p}$. There are currently four main SMs in this category in MTF - particle filter (**PF**) (Kwon et al., 2014), nearest neighbor (**NN**) (Dick et al., 2013), least median of squares (**LMS**) (Rousseeuw, 1984) and random sample consensus (**RANSAC**) (Zhang et al., 2015).

These SMs work by generating a set of random samples for $\mathbf{p}$ and evaluating the goodness of each by some measure of
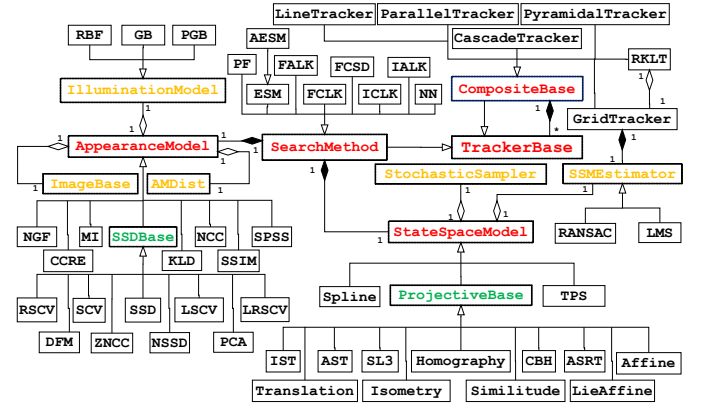

Fig. 3. MTF Class Diagram showing all models currently implemented. Pure and partially abstract classes are respectively shown in red and green while concrete classes are in black. Classes that are sub parts of AM and SSM are in yellow. The meanings of acronyms not defined in text can be found in (Singh, 2017, 2018).

similarity with the template. NN and PF generate samples directly by drawing them from an appropriate Gaussian distribution while LMS and RANSAC obtain them indirectly by finding the best fit parameters that explain the transformation between two sets of corresponding points. The performance of these SMs thus thus depends mostly on the number and quality of stochastic samples used. While the former is limited only by the available computational resources, the latter is a bit harder to guarantee for a general SSM/AM. For methods that draw samples from a Gaussian distribution, the quality thereof is determined by the covariance matrix used and, to the best of our knowledge, no widely accepted method exists to estimate it in the general case. Most works either use heuristics or perform extensive hand tuning to get acceptable results, sometimes even using different values for each tested sequence (Kwon et al., 2014).

Given this, a reasonable way to decompose these methods to fit our framework is to delegate the responsibility of generating the set of samples and estimating its mean entirely to the SSM and AM while letting the latter evaluate the suitability of each sample by providing the likelihood of the corresponding patch. Such a decomposition ensures both theoretical validity and good performance in practice since the definition of what constitutes a good sample and how the mean of a sample set is to be evaluated depends on the SSM/AM, as do any heuristics for generating these samples (like the variance for each component of $\mathbf{p}$).

3. System Design

3.1. Overview

As shown in the class diagram in Fig. 3, MTF closely follows the decomposition described in the previous section and has three abstract base classes corresponding to the three sub modules - `SearchMethod`, `AppearanceModel` and

`StateSpaceModel`.[1] Of these, only SM is a generic/templated class that is templated on specializations of the other two classes. A concrete tracker, defined as a particular combination of the three sub modules, thus corresponds to a subclass of SM that has been instantiated with subclasses of AM and SSM.

A particular SM in this formulation is defined only by its objective - to find the **p** that maximizes the similarity measure defined by the AM. Thus, different implementations of SM can cover a potentially wide range of methods that have little in common. As a result, SM is the least specific of these classes and only provides functions to initialize, update and reset the tracker along with accessors to obtain its current state. In fact, an SM is regarded in this framework simply as one way to *use* the methods provided by the other two sub modules in order to accomplish the above objective. The idea is to abstract out as much computation from the SM to the AM/SSM as possible so as to make for a general purpose tracker. Therefore, this section describes only AM and SSM in detail while some of the SMs currently available in MTF are presented in the next section as examples of using the functionality described here to carry out the search in different ways.

Another consequence of this conceptual impreciseness of SM is that a specific SM may use only a small subset of the functionality provided by AM/SSM. For instance, gradient descent type SMs do not use the random sampling functions of SSM and conversely, stochastic SMs do not use the derivative functions required by the former. This has two further implications. Firstly, the functionality set out in AM and SSM is not fixed but can change depending on the requirements of an SM, i.e. if a new SM is to be implemented that requires some functionality not present in the current specifications, the respective class can be extended to support it - as long as such an extension makes logical sense within the definition of that class. Secondly, it is not necessary for all combinations of AMs and SSMs to support all SMs. For instance a similarity measure does not need to be differentiable to be a valid AM as long as it is understood that it cannot be used with SMs that require derivatives.

In the broadest sense, the division of functionality between AM and SSM described next can be seen as AM being responsible for everything to do with the image I, the sampled patch $\mathbf{I}(\mathbf{x})$ and the similarity f computed using it, while SSM handles the actual *points* $\mathbf{x}$ at which the patch is sampled along with the warping function $\mathbf{w}$ that defines $\mathbf{x}$ in terms of $\mathbf{x_0}$ and $\mathbf{p_s}$.

3.2. Composite Tracking

It may be noted that SM itself derives from a non generic base class called `TrackerBase` for convenient creation and interfacing of objects corresponding to heterogeneous trackers, including those external to MTF [2] , so that they can be run simultaneously and their results combined to create composite
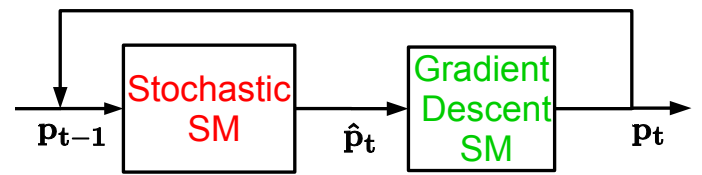

Fig. 4. A stochastic and gradient descent SM in cascade. The rough estimate produced by the former is refined by the latter and then fed back to the former to be used as the starting search point for the next frame.

trackers that are more robust than any of their components. Allowing a diverse variety of trackers to integrate seamlessly is one of the core design objectives of MTF and this is emphasized by having such composite trackers derive from a separate base class called `CompositeBase` which in turn derives from `TrackerBase` while also containing several instances of it. Individual RBTs are well known to be prone to failures and since more than three decades of research has failed to make significant improvements in this regard (Singh et al., 2017), this approach seems to be one of the more promising ones (Zhang et al., 2015; Singh et al., 2017). MTF has thus been designed to facilitate work in this direction.

Five composite trackers have currently been implemented:

- `GridTracker`: This corresponds to the stochastic SMs based on indirect sampling mentioned in Sec. 2.4. This uses a grid of (typically low DOF) trackers such that each tracks a different sub patch within the tracked object. The independent results of these trackers are then combined by a robust estimator provided by the SSM to estimate the best fit warp that gives the overall location of the patch.

- `LineTracker`: This identifies straight lines in the object of interest and uses 2 DOF trackers to track multiple points on each line. The outputs of these trackers are then used to estimate the best fit line assuming linearity of points as an invariant property under the warp that the object patch has undergone. Constraints between different lines, such as parallelism, can also be enforced to improve this estimation further. By resetting the trackers to their expected positions on these lines, any drift can be compensated for.

- `ParallelTracker`: This one runs multiple trackers in parallel to track the same patch and then combines their outputs to produce a more robust estimate of the location of the tracked object. Many methods may be used to combine the locations produced by the different trackers, with a simple example being to take the mean of the bounding box corners.

- `PyramidalTracker`: This one builds a Gaussian image pyramid (Bouguet, 2000) and then tracks each level of the pyramid through a different instance of (usually) the same tracker. The output of the tracker at level n, after appropriate scaling, is used as starting point for the one at level $n + 1$.

---

[1]For brevity, these will be referred to as SM, AM and SSM respectively with the font serving to distinguish the *classes* from the corresponding *concepts*.

[2]Several state of the art OLTs with publicly available C++ code - DSST (Danelljan et al., 2016), KCF (Henriques et al., 2015), CMT (Nebehay and Pflugfelder, 2015), Struck (Hare et al., 2016), TLD (Kalal et al., 2012), MIL (Babenko et al., 2011), RCT (Zhang et al., 2012), FragTrack (Adam et al., 2006), GOTURN (Held et al., 2016) - have already been integrated with MTF.
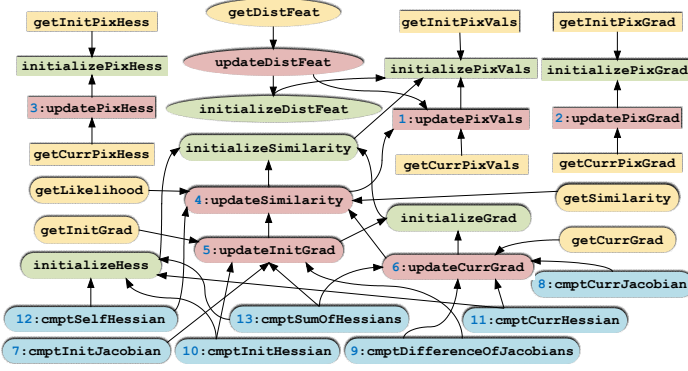
Fig. 5. Dependency relationships between various functions in **AM**: an arrow pointing from A to B means that A depends on B. Color of a function box denotes its type - **green**: initializing; **red**: updating; **blue**: interfacing and **yellow**: accessor function. Shape of a function box represents the part of **AM** it belongs to - rectangle: Image Operations; rounded rectangle: Similarity Functions; ellipse: Distance Feature. The numbers attached to some of the nodes refer Table 1.

Table 1. Specifications for important methods in AM. IDs in first column refer Fig. 5

ID	Inputs	Output/Variable updated
1	$\mathbf{x_t}$	$\mathbf{I_t(x_t)}$
2	$\mathbf{x_t}$	$\nabla \mathbf{I_t}$
3	$\mathbf{x_t}$	$\nabla^2 \mathbf{I_t}$
4	None	$f(\mathbf{I_0, I_t})$
5	None	$\dfrac{\partial f(\mathbf{I_0, I_t})}{\partial \mathbf{I_0}}$
6	None	$\dfrac{\partial f(\mathbf{I_0, I_t})}{\partial \mathbf{I_t}}$
7	$\dfrac{\partial \mathbf{I_0}}{\partial \mathbf{p_s}}$	$\dfrac{\partial f(\mathbf{I_0(p), I_t})}{\partial \mathbf{p}}$ (Eq. A.4)
8	$\dfrac{\partial \mathbf{I_t}}{\partial \mathbf{p_s}}$	$\dfrac{\partial f(\mathbf{I_0, I_t(p)})}{\partial \mathbf{p}}$ (Eq. A.1, A.2)
9	$\dfrac{\partial \mathbf{I_0}}{\partial \mathbf{p_s}}, \dfrac{\partial \mathbf{I_t}}{\partial \mathbf{p_s}}$	$\dfrac{\partial f(\mathbf{I_0, I_t(p)})}{\partial \mathbf{p}} - \dfrac{\partial f(\mathbf{I_0(p), I_t})}{\partial \mathbf{p}}$ (Eq. A.7)
10[1]	$\dfrac{\partial \mathbf{I_0}}{\partial \mathbf{p_s}}, \dfrac{\partial^2 \mathbf{I_0}}{\partial \mathbf{p_s}^2}$	$\dfrac{\partial^2 f(\mathbf{I_0(p), I_t})}{\partial \mathbf{p}^2}$
11	$\dfrac{\partial \mathbf{I_t}}{\partial \mathbf{p_s}}, \dfrac{\partial^2 \mathbf{I_t}}{\partial \mathbf{p_s}^2}$	$\dfrac{\partial^2 f(\mathbf{I_0, I_t(p)})}{\partial \mathbf{p}^2}$
12	$\dfrac{\partial \mathbf{I_t}}{\partial \mathbf{p_s}}, \dfrac{\partial^2 \mathbf{I_t}}{\partial \mathbf{p_s}^2}$	$\dfrac{\partial^2 f(\mathbf{I_t, I_t(p)})}{\partial \mathbf{p}^2}$ (Eq. A.9, A.10)
13	$\dfrac{\partial \mathbf{I_0}}{\partial \mathbf{p_s}}, \dfrac{\partial^2 \mathbf{I_0}}{\partial \mathbf{p_s}^2}, \dfrac{\partial \mathbf{I_t}}{\partial \mathbf{p_s}}, \dfrac{\partial^2 \mathbf{I_t}}{\partial \mathbf{p_s}^2}$	$\dfrac{\partial^2 f(\mathbf{I_0(p), I_t})}{\partial \mathbf{p}^2} + \dfrac{\partial^2 f(\mathbf{I_0, I_t(p)})}{\partial \mathbf{p}^2}$

[1] Functions 10-13 have overloaded variants that omit the second term in Eq. 4, as in Eq. A.8, and so do not require $\dfrac{\partial^2 \mathbf{I}}{\partial \mathbf{p_s}^2}$ as input

Table 2. Specifications for important methods in ILM.

Function	Inputs	Output
update	$\mathbf{p_a}, \Delta \mathbf{p_a}$	$\mathbf{p_a'} \quad \| \quad g(g(\mathbf{I_t, p_a}), \Delta \mathbf{p_a}) = g(\mathbf{I_t, p_a'})$
apply	$\mathbf{I_t, p_a}$	$g(\mathbf{I, p_a})$
invert	$\mathbf{p_a}$	$\mathbf{p_a'} \mid g(g(\mathbf{I_t, p_a}), \mathbf{p_a'}) = \mathbf{I_t}$
cmptParamJacobian	$\dfrac{\partial f}{\partial \mathbf{g}}, \mathbf{I_t, p_a}$	$\dfrac{\partial f}{\partial \mathbf{p_a}}$
cmptPixJacobian	$\dfrac{\partial f}{\partial \mathbf{g}}, \mathbf{I_t, p_a}$	$\dfrac{\partial f}{\partial \mathbf{I_t}}$
cmptParamHessian*	$\dfrac{\partial^2 f}{\partial \mathbf{g}^2}, \dfrac{\partial f}{\partial \mathbf{g}}, \mathbf{I_t, p_a}$	$\dfrac{\partial^2 f}{\partial \mathbf{p_a}^2}$
cmptPixHessian*	$\dfrac{\partial^2 f}{\partial \mathbf{g}^2}, \dfrac{\partial f}{\partial \mathbf{g}}, \mathbf{I_t, p_a}$	$\dfrac{\partial^2 f}{\partial \mathbf{I_t}^2}$
cmptCrossHessian*	$\dfrac{\partial^2 f}{\partial \mathbf{g}^2}, \dfrac{\partial f}{\partial \mathbf{g}}, \mathbf{I_t, p_a}$	$\dfrac{\partial^2 f}{\partial \mathbf{I_t}\partial \mathbf{p_a}}$

* All Hessian functions have overloaded variants that omit the second terms in respective expressions and so do not require $\dfrac{\partial f}{\partial \mathbf{g}}$ as input

- `CascadeTracker`: This tracks the same patch using multiple trackers too, but here the output of the tracker at each layer of the cascade is used as the starting point for the tracker at the next layer. The output of the last layer is fed back to the first one to make a closed loop system. Fig. 4 shows an example of this arrangement where two SMs are used in cascade. The top-performing LMES is an example (Sec. 6) of this.

Though not shown in Fig. 3, several of these composite trackers also have specialized variants where all the constituent trackers have the same AM and SSM and differ only in their SM. Since all SMs derive from SM, this specialization makes it possible for the constituent trackers to utilize the functionality available in SM but not in TrackerBase in the ways that they interact with and benefit from each other. TrackerBase is designed to allow third party trackers including OLTs to integrate with MTF and so does not include functions specific to RBTs. An example of such a functionality is selective pixel integration (Dellaert and Collins, 1999) which is used by the RKLT tracker (Zhang et al., 2015) to filter out parts of the patch that do not agree with the template due to partial occlusions or localized illumination changes.

It is emphasized here that any two or more of the composite trackers can themselves be combined together in arbitrary ways. For example, it is straightforward to create a cascade tracker where each layer is itself a pyramidal tracker. It would be equally easy to create a pyramidal tracker where each level is tracked by a cascade tracker. Different levels of the pyramid might even be tracked by different cascade arrangements just as different layers of the cascade in the previous example might differ in the number of pyramidal levels within the corresponding trackers.

3.3. AppearanceModel

This class can be divided into three main parts with each defined as a set of variables dependent on I_0 and I_t with a cor-responding initialize and update function for each. The division is mainly conceptual and methods in different parts are free to interact with each other in practice. Table 1 presents a brief specification of some important methods in AM.

3.3.1. Image Operations

This part, abstracted into a separate class called ImageBase, handles all pixel level operations on I_t like extracting the patch $\mathbf{I}(\mathbf{x})$ and computing its numerical gradient $\nabla \mathbf{I}$ and Hessian $\nabla^2 \mathbf{I}$. It uses sub pixel interpolation so that these quantities can be treated as continuous functions of $\mathbf{x}$.

Though AM bears a composition or "has a" relationship with ImageBase, the latter is actually implemented as a base class of the former to maintain simplicity of the interface and allow a specializing class to efficiently override functions in both classes. Moreover, having a separate class for pixel related operations means that AMs like SCV and ZNCC that differ from SSD only in using a modified version of $\mathbf{I_0}$ or $\mathbf{I_t}$ (thus deriving from SSDBase in Fig. 3), can implement the corresponding

mapping entirely within the functions defined in `ImageBase` and be combined easily with other AMs besides SSD.

3.3.2. Similarity Functions

This is the core of `AM` and handles the computation of the similarity measure $f(\mathbf{I}^*, \mathbf{I^c}, \mathbf{p_a})$ and its derivatives $\partial f/\partial \mathbf{I}$ and $\partial^2 f/\partial \mathbf{I}^2$ w.r.t. both $\mathbf{I}^*$ and $\mathbf{I^c}$. It also provides interfacing functions to use inputs from `SSM` to compute the derivatives of f w.r.t. `SSM` parameters using the chain rule. As a notational convention, all interfacing functions, including those in `SSM`, are prefixed with `cmpt`.

The functionality specific to $\mathbf{p_a}$ is abstracted into a separate class called `IlluminationModel` so it can be combined with any AM to add photometric parameters (Silveira and Malis, 2007; Bartoli, 2008) to it. This class provides functions to compute $g(\mathbf{I}, \mathbf{p_a})$ and its derivatives including $\partial g/\partial \mathbf{p_a}$, $\partial^2 g/\partial \mathbf{p_a}^2$, $\partial g/\partial \mathbf{I}$, $\partial^2 g/\partial \mathbf{I}^2$ and $\partial^2 g/\partial \mathbf{I}\partial \mathbf{p_a}$. These are called from within `AM` to compute the respective derivatives w.r.t. f so that the concept of ILM is transparent to the SM. It should be noted that `AM` is designed to support f with arbitrary $\mathbf{p_a}$ of which ILM is a special case. It also supports learning to update the object's appearance, as present, for instance, in `PCA` (Ross et al., 2008).

Since several of the functions in this part of `AM` involve common computations, there exist *transitive dependency* relationships between them (Fig. 5) to avoid repeating these computations when multiple quantities are needed by the SM. What this means is that a function lower down in the dependency hierarchy may delegate part of its computations to any function higher up in the hierarchy so that the latter must be called *before* calling the former if correct results are to be expected.

3.3.3. Distance Feature

This part, implemented within `AMDist`, is designed specifically to enable integration with the FLANN library (Muja and Lowe, 2009) that is used by the NN based SM. It provides two main functions:

- A feature transform $\mathbf{D}(\mathbf{I}^*) : \mathbb{R}^N \mapsto \mathbb{R}^K$ that maps the pixel values extracted from a patch $\mathbf{I}^*$ into a feature vector that contains the results of all computations in $f(\mathbf{I}^*, \mathbf{I^c})$ that depend only on $\mathbf{I}^*$, and likewise for $\mathbf{I^c}$. This transform is applied to all sampled patches during initialization and only the resultant feature vectors are stored in the index. At runtime, it is applied to $\mathbf{I^c}$ and the feature vector is passed to the distance functor.

- A highly optimized distance functor $f_D(\mathbf{D}(\mathbf{I}^*), \mathbf{D}(\mathbf{I^c}))$: $\mathbb{R}^K \times \mathbb{R}^K \mapsto \mathbb{R}$ that computes a measure of the distance or dissimilarity between $\mathbf{I}^*$ and $\mathbf{I^c}$ (typically the negative of $f(\mathbf{I}^*, \mathbf{I^c})$) given the distance features $\mathbf{D}(\mathbf{I}^*)$ and $\mathbf{D}(\mathbf{I^c})$ as inputs.

The main idea behind the design of these two components is to place as much computational load as possible on $\mathbf{D}$ so that the runtime speed of f_D is maximized, with the premise that the former is called mostly during initialization when the sample dataset is to be built, while the latter is called online to find the best matches for a candidate patch in the dataset. An optimal

Table 3. Specifications for important methods in SSM.

Function	Inputs	Output/Result
`compositionalUpdate`	$\Delta \mathbf{p_s}$	$\mathbf{p_{st}} = \mathbf{p_s}' \mid \mathbf{w}(\mathbf{x}, \mathbf{p_s}') = \mathbf{w}(\mathbf{w}(\mathbf{x}, \Delta\mathbf{p_s}), \mathbf{p_{st}})$
`additiveUpdate`	$\Delta \mathbf{p_s}$	$\mathbf{p_{st}} = \mathbf{p_{st}} + \Delta\mathbf{p_s}$
`invertState`	$\mathbf{p_s}$	$\mathbf{p_s}' \mid \mathbf{w}(\mathbf{w}(\mathbf{x}, \mathbf{p_s}), \mathbf{p_s}') = \mathbf{x}$
`cmptPixJacobian`	$\nabla \mathbf{I_t}$	$\left.\frac{\partial \mathbf{I_t}}{\partial \mathbf{p_s}}\right\|_{\mathbf{p_s}=\mathbf{p_{st}}}$ (Eq. A.1)
`cmptWarpedPixJacobian`	$\nabla \mathbf{I_t}$	$\left.\frac{\partial \mathbf{I_t}(\mathbf{w})}{\partial \mathbf{p_s}}\right\|_{\mathbf{p_s}=\mathbf{p_{s0}}}$ (Eq. A.2, A.3)
`cmptApproxPixJacobian`	$\nabla \mathbf{I_0}$	$\frac{\partial \mathbf{I_t}}{\partial \mathbf{p_{st}}}$ (approx) (Eq. A.5, A.6)
`cmptPixHessian`	$\nabla \mathbf{I_t}, \nabla^2 \mathbf{I_t}$	$\left.\frac{\partial^2 \mathbf{I_t}}{\partial \mathbf{p_s}^2}\right\|_{\mathbf{p_s}=\mathbf{p_{st}}}$ (Eq. A.11)
`cmptWarpedPixHessian`	$\nabla \mathbf{I_t}, \nabla^2 \mathbf{I_t}$	$\left.\frac{\partial^2 \mathbf{I_t}(\mathbf{w})}{\partial \mathbf{p_s}^2}\right\|_{\mathbf{p_s}=\mathbf{p_{s0}}}$ (Eq. A.12, A.13)
`cmptApproxPixHessian`	$\nabla \mathbf{I_0}, \nabla^2 \mathbf{I_0}$	$\frac{\partial^2 \mathbf{I_t}}{\partial \mathbf{p_{st}}^2}$ (approx) (Eq. A.14)

design may involve a trade off between the size K of the feature vector and the amount of computation performed in f_D. For non symmetrical AMs, i.e. where $f(\mathbf{I}^*, \mathbf{I^c}) \neq f(\mathbf{I^c}, \mathbf{I}^*)$ (e.g. CCRE and SCV), the feature vector may also include an indicator flag so that f_D can determine which of its arguments corresponds to the $\mathbf{D}(\mathbf{I}^*)$ and which to $\mathbf{D}(\mathbf{I^c})$. This is needed because FLANN does not specify the order in which the arguments will be passed to f_D and examination of its code showed that this order varies for each index type as well as for different calls to f_D within the same index.

3.4. StateSpaceModel

This class has a simpler internal state than `AM` and can be described by only three main variables at any time t - sampled grid points $\mathbf{x_t}$, corresponding corners $\mathbf{x_t^c}$ and state parameters $\mathbf{p_{st}}$. It may be noted (Fig. 3) that, though `SSM` is designed to support any arbitrary $\mathbf{w}$, most SSMs currently implemented are subsets of the planar projective transform and so derive from `ProjectiveBase` that abstracts out the functionality common to these.

Functions in `SSM` can be divided into two categories:

3.4.1. Warping Functions

This is the core of `SSM` and provides a function $\mathbf{w}$ to transform a regularly spaced grid of points $\mathbf{x_0}$ representing the target patch into a warped patch $\mathbf{x_t} = \mathbf{w}(\mathbf{x_0}, \mathbf{p_{st}})$ that captures the tracked object's motion in image space. It also allows for the compositional inverse of $\mathbf{w}$ to be computed (`invertState`) to support inverse SMs. Further, there are functions to compute the derivatives of $\mathbf{w}$ w.r.t. both $\mathbf{x}$ and $\mathbf{p_s}$ but, unlike `AM`, `SSM` does not store these as state variables, rather their computation is implicit in the interfacing functions that compute $\partial \mathbf{I}/\partial \mathbf{p_s}$ and $\partial^2 \mathbf{I}/\partial \mathbf{p_s}^2$ using the chain rule. This design decision was made for reasons of efficiency since $\partial \mathbf{w}/\partial \mathbf{p_s}$ and $\partial \mathbf{w}/\partial \mathbf{x}$ are large and often very sparse tensors. Computing these separately, thus, not only wastes a lot of memory but is also very inefficient.

Finally, there are four ways to update the internal state: incrementally using additive (`additiveUpdate`) or compositional

(compositionalUpdate) formulations, or outright by providing either the state vector (setState) or the corresponding corners (setCorners) that define the current location of the patch. There are no complex dependencies in SSM - the correct performance of interfacing functions and accessors depends only on one of the update functions being called every iteration. Table 3 lists the functionality of some important methods in this part.

3.4.2. Stochastic Sampler

This part is provided to support stochastic SMs and offers following functionality to this end:

- generate small random incremental updates to $\mathbf{p_s}$ (generatePerturbation) by drawing these from a zero mean normal distribution with either user provided or heuristically estimated (estimateStateSigma) variance.

- generate stochastic state samples using the given state transition model - currently random walk (additiveRandomWalk) and first order auto regression (additiveAutoRegression1) are supported. There are also compositional variants.

- estimate the mean of a set of samples of $\mathbf{p_s}$ (estimateMeanOfSamples)

- estimate the best fit $\mathbf{p_s}$ from a set of original and warped point pairs (estimateWarpFromPts) using a robust method implemented within SSMEstimator - currently RANSAC (Zhang et al., 2015) and LMS (Rousseeuw, 1984) are supported.

4. Search Methods

This section presents pseudo codes for several SMs currently implemented in MTF to exemplify the usage of functions described in the previous sections. Algorithms 1-5 illustrate how MTF implements the many variants of gradient based SMs developed over the past 35 years (Baker and Matthews, 2004) while Algorithms 6-8 illustrate stochastic SMs (Sec. 2.4). Following are some points and conventions to be noted:

- different algorithms make extensive references to portions of each other not only to avoid redundancy but also to emphasize the parts they have in common.

- *am* and *ssm* respectively refer to instances of AM and SSM (or rather of specializations thereof)

- *am* has direct access to the latest image in the sequence so it is not passed explicitly in function calls - this is one of the design features of AM to avoid the overhead of passing the image repeatedly.

- several special cases like the optional use of the first order Hessian (Eq. A.8), parameterization and online learning of AM and iterative form of the update function are demonstrated only in Alg. 1 but should be obvious by analogy for other SMs too.

Algorithm 1 ICLK

```
 1: function initialize(corners)
 2:     ssm.initialize(corners)
 3:     am.initializePixVals(ssm.getPts())
 4:     am.initializePixGrad(ssm.getPts())
 5:     am.initializeSimilarity()
 6:     am.initializeGrad()
 7:     am.initializeHess()
 8:     dI0_dps ← ssm.cmptWarpedPixJacobian(am.getInitPixGrad())
 9:     if use_first_order_hessian then
10:         d2f_dp2 ← am.cmptSelfHessian(dI0_dps)
11:     else
12:         am.initializePixHess(ssm.getPts())
13:         d2I0_dps2 ← ssm.cmptInitPixHessian(
                    am.getInitPixHess(), am.getInitPixGrad())
14:         d2f_dp2 ← am.cmptSelfHessian(dI0_dps, d2I0_dps2)
15:     end if
16: end function
17: function update
18:     for i ← 1, max_iters do
19:         am.updatePixVals(ssm.getPts())
20:         am.updateSimilarity()
21:         am.updateInitGrad()
22:         df_dp ← am.cmptInitJacobian(dI0_dps)
23:         delta_p ← −d2f_dp2.inverse()*df_dp
24:         delta_ps ← delta_p.head(ssm.getStateSize())
25:         delta_pa ← delta_p.tail(am.getStateSize())
26:         inv_delta_ps ← ssm.invertState(delta_ps)
27:         inv_delta_pa ← am.invertState(delta_pa)
28:         prev_corners ← ssm.getCorners()
29:         ssm.compositionalUpdate(inv_delta_ps)
30:         am.update(inv_delta_pa)
31:         if ‖prev_corners − ssm.getCorners()‖² < ε then
32:             break
33:         end if
34:     end for
35:     am.updateModel(ssm.getPts())
36:     return ssm.getCorners()
37: end function
```

- v.head(h) and v.tail(t) in Alg. 1 respectively refer to the first h and last t elements in the $h + t$ length vector v.

- *flann* in Alg. 6 is an instance of FLANN library (Muja and Lowe, 2009) that can build an index from a set of samples and search it for a new candidate.

- variables used to store the results of computations are not described explicitly but their meanings should be clear from their names and context. For instance, *sample_dataset* and *ssm_perturbations* used in Alg. 6 respectively refer to $n{\times}K$ and $n{\times}S$ matrices, each of whose rows contains the distance feature **D** (Sec. 3.3.3) and the SSM state $\mathbf{p_s}$ corresponding to one sample so that n = number of samples.

- only one state transition model is shown in Alg. 7 though several others are available too (Sec. 3.4.2).

- Alg. 8 shows only the GridTracker component of LMS and RANSAC; the actual robust estimation using one of these is performed in estimateWarpFromPts function of SSM (Sec. 3.4.2).

- sampling resolution of *ssm* in Alg. 8 is set to be same as the grid resolution and the function getRegion (c, s) in line 6 returns the corners of a rectangular region of size s with centroid c.

Algorithm 2 FCLK

```
1:  function initialize(corners)
2:      lines 2-7 of Alg. 1
3:      am.initializePixHess(ssm.getPts())
4:  end function
5:  function update
6:      lines 19-20 of Alg. 1
7:      am.updateCurrGrad()
8:      am.updatePixGrad(ssm.getPts())
9:      am.updatePixHess(ssm.getPts())
10:     dIt_dps ← ssm.cmptWarpedPixJacobian(am.getCurrPixGrad())
11:     d2It_dps2 ← ssm.cmptWarpedPixHessian(
                        am.getCurrPixHess(), am.getCurrPixGrad())
12:     df_dp ←am.cmptCurrJacobian(dIt_dps)
13:     d2f_dp2 ←am.cmptSelfHessian(dIt_dps, d2It_dps2)
14:     delta_p ← −d2f_dp2.inverse()∗df_dp
15:     ssm.compositionalUpdate(delta_p)
16:     return ssm.getCorners()
17: end function
```

Algorithm 3 ESM

```
1:  function initialize(corners)
2:      lines 2-3 of Alg. 2
3:      dI0_dps ← ssm.cmptWarpedPixJacobian(am.getInitPixGrad())
4:      d2f_dp2_0 ← am.cmptSelfHessian(dI0_dps, d2I0_dps2)
5:  end function
6:  function update
7:      lines 6-11 of Alg. 2
8:      am.updateInitGrad()
9:      df_dp←am.cmptDifferenceOfJacobians(dI0_dps, dIt_dps)
10:     d2f_dp2_t ←am.cmptSelfHessian(dIt_dps, d2It_dps2)
11:     d2f_dp2 ←d2f_dp2_0 + d2f_dp2_t
12:     lines 14-16 of Alg. 2
13: end function
```

Algorithm 4 IALK

```
1:  function initialize(corners)
2:      same as Alg. 2
3:  end function
4:  function update
5:      lines 6-7 of Alg. 2
6:      dIt_dps ← ssm.cmptApproxPixJacobian(am.getInitPixGrad())
7:      d2It_dps2 ← ssm.cmptApproxPixHessian(am.getInitPixHess(),
                        am.getInitPixGrad())
8:      lines 12-14 of Alg. 2
9:      ssm.additiveUpdate(delta_p)
10:     return ssm.getCorners()
11: end function
```

Algorithm 5 FALK

```
1:  function initialize(corners)
2:      same as Alg. 2
3:  end function
4:  function update
5:      lines 6-9 of Alg. 2
6:      dIt_dps ← ssm.cmptPixJacobian(am.getCurrPixGrad())
7:      d2It_dps2 ← ssm.cmptPixHessian(am.getCurrPixHess(),
                        am.getCurrPixGrad())
8:      lines 8-10 of Alg. 4
9:  end function
```

5. Use Cases

This section presents the following use cases for MTF in C++ style pseudo code:

- Track an object in an image sequence using a simple (Alg. 9) and a composite (Alg. 10) tracker.

- Estimate the trajectory of a UAV within a large satellite image of an area from images it took while flying over that area (Alg. 11).

- Create an image mosaic in real time from a video sequence captured by a camera moving over different parts of the planar scene to be stitched (Alg. 12).

Algorithm 6 NN

```
1:  function initialize(corners)
2:      lines 2-3 of Alg. 1
3:      state_sigma← ssm.estimateStateSigma()
4:      ssm.initializeSampler(state_sigma)
5:      am.initializeDistFeat()
6:      for sample_id ← 1, no_of_samples do
7:          ssm_updates.row(sample_id) ← ssm.generatePerturbation()
8:          inv_update ←ssm.invertState(ssm_updates.row(sample_id))
9:          ssm.compositionalUpdate(inv_update)
10:         am.updatePixVals(ssm.getPts())
11:         am.updateDistFeat()
12:         sample_dataset.row(sample_id) ← am.getDistFeat()
13:         ssm.compositionalUpdate(ssm_updates.row(sample_id))
14:     end for
15:     flann.buildIndex(sample_dataset)
16: end function
17: function update
18:     am.updatePixVals(ssm.getPts())
19:     am.updateDistFeat()
20:     nn_sample_id ← flann.searchIndex(am.getDistFeat())
21:     ssm.compositionalUpdate(ssm_updates.row(nn_sample_id))
22:     return ssm.getCorners()
23: end function
```

Algorithm 7 PF

```
1:  function initialize(corners)
2:      lines 2-4 of Alg. 6
3:      am.initializeSimilarity()
4:      for particle_id ← 1, no_of_particles do
5:          particles[particle_id].state ← ssm.getState()
6:          particles[particle_id].weight ← 1/no_of_particles
7:      end for
8:  end function
9:  function update
10:     for particle_id ← 1, no_of_particles do
11:         particles[particle_id].state ← ssm.compositionalRandomWalk(
12:                         particles[particle_id].state)
13:         ssm.setState(particles[particle_id].state)
14:         am.updatePixVals(ssm.getPts())
15:         am.updateSimilarity()
16:         particles[particle_id].weight ←am.getLikelihood()
17:     end for
18:     normalize weights and resample the particles
19:     mean_state ← ssm.estimateMeanOfSamples(particles);
20:     ssm.setState(mean_state)
21:     return ssm.getCorners()
22: end function
```

Algorithm 8 LMS/RANSAC

```
1:  function initialize(corners)
2:      sub_trackers ← vector of 2 DOF sub patch trackers
3:      ssm.initialize(corners)
4:      curr_pts ← ssm.getPts()
5:      for pt_id ← 1, no_of_pts do
6:          sub_patch_corners ← getRegion(curr_pts[pt_id], sub_patch_size)
7:          sub_trackers[pt_id].initialize(sub_patch_corners)
8:      end for
9:  end function
10: function update
11:     prev_pts ← curr_pts
12:     for pt_id ← 1, no_of_pts do
13:         sub_trackers[pt_id].update()
14:         curr_pts[pt_id] ← getCentroid(sub_trackers[pt_id].getRegion())
15:     end for
16:     opt_warp ← ssm.estimateWarpFromPts(prev_pts, curr_pts)
17:     warped_corners ← ssm.applyWarpToCorners(ssm.getCorners(), opt_warp)
18:     ssm.setCorners(warped_corners)
19:     lines 4-8 of Alg. 8
20:     return ssm.getCorners()
21: end function
```

Algorithm 9 Object Tracking - Simple

```
1:  using namespace mtf;
2:  ICLK<SSD, Homography> tracker;
3:  GaussianSmoothing pre_proc(input.getFrame(), tracker.inputType());
4:  tracker.initialize(pre_proc.getFrame(), init_location);
5:  while input.update() do
6:      pre_proc.update(input.getFrame());
7:      tracker.update(pre_proc.getFrame());
8:      new_location ← tracker.getRegion();
9:  end while
```

Algorithm 10 Object Tracking - Composite

```
1: PF<ZNCC, Affine> tracker1;
2: FCLK<SSIM, SL3> tracker2;
3: vector<TrackerBase*> trackers = {&tracker1, &tracker2};
4: CascadeTracker tracker(trackers);
5: lines 3-9 of Alg. 9.
```

Algorithm 11 UAV Trajectory Estimation in Satellite Image

```
1: ESM<MI, Similitude> tracker;
2: uav_img_corners←getFrameCorners(input.getFrame());
3: tracker.initialize(satellite_img, init_uav_location);
4: curr_uav_location←tracker.getRegion();
5: while input.update() do
6:     tracker.initialize(input.getFrame(), uav_img_corners);
7:     tracker.setRegion(curr_uav_location);
8:     tracker.update(satellite_img);
9:     curr_uav_location←tracker.getRegion();
10: end while
```

Algorithm 12 Online Image Mosaicing

```
1: FALK<MCNCC, Isometry> tracker;
2: mos_img ← writePixelsToImage(input.getFrame(), init_mos_location, mos_size);
3: mos_location ← init_mos_location;
4: while input.update() do
5:     temp_img ← writePixelsToImage(input.getFrame(), mos_location, mos_size);
6:     tracker.initialize(temp_img, mos_location);
7:     tracker.update(mos_img);
8:     mos_location ← tracker.getRegion();
9:     mos_img ← writePixelsToImage(input.getFrame(), mos_location, mos_size);
10: end while
```

Visual illustrations for all of these can be found in the demonstrations video on MTF website (Singh, 2018). Following are details regarding variables and functions used in these algorithms that assist the reader in understanding them better:

- MTF comes with an input module with wrappers for image capturing functions in OpenCV, ViSP and XVision. It is represented here by *input* and is assumed to have been initialized with the appropriate source.

- Raw images acquired by the input module can optionally be passed to the preprocessing module that provides wrappers for OpenCV image filtering and conversion functions. An example of this is GaussianSmoothing in Alg. 9.

- Though only five combinations of SM, AM and SSM are shown here, these can be replaced by virtually any combination of methods in Fig. 3.

- MTF also has a set of general utility functions for image and warping related operations. The following have been used in Alg. 11 and 12:

    - getFrameCorners(*image*) returns a 2×4 matrix containing the corners of *image* and is thus used when the entire image is to be considered as the tracked region.

    - writePixelsToImage(*patch*, *corners*, *size*) writes the pixel values in *patch* to an image with dimensions *size* within the region bounded by *corners*.

- *init_mos_location* in Alg. 12 is the user specified location of the first frame in the sequence within the mosaic image of size *mos_size*. This is typically the center of the mosaic though can be elsewhere depending on the actual sequence.
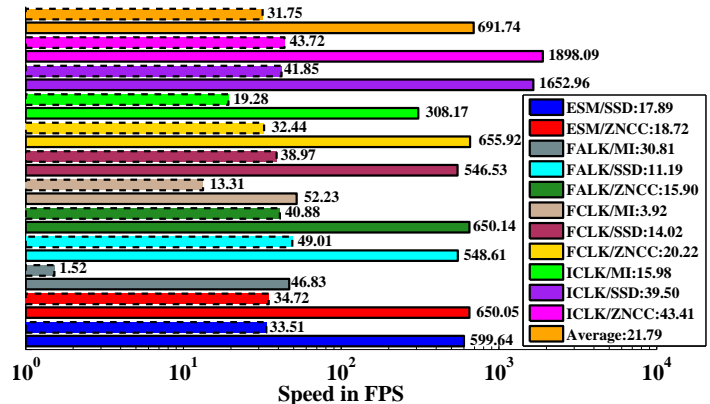

Fig. 6. ViSP vs MTF average tracker speeds for all combinations of SMs and AMs supported by ViSP. MTF and ViSP results are shown in solid and dotted lines respectively. Note that logarithmic scaling has been used on the x axis for better visibility of ViSP bars though the actual figures are also shown. Speedup provided by MTF is shown in the legends. Results were generated on a 4 GHz Intel Core i7-4790K machine with 32 GB RAM.

6. Performance

An existing system that is functionally similar to MTF is the template tracker module of the ViSP library (Marchand et al., 2005). It provides 4 SMs, 3 AMs and 6 SSMs though not all combinations work. MTF offers several advantages over ViSP. Firstly, SMs and AMs in ViSP are not implemented as independent modules; rather, each combination of methods has its own class. This makes it difficult to add a new method for either of these sub modules and combine it with existing methods for the others. Secondly, MTF has several more AMs, one more GD based SM as well as four stochastic SMs. It also allows multiple SMs to be combined effortlessly to create novel composite SMs. Thirdly, MTF is significantly faster than ViSP. As shown in fig. 6, MTF is usually more than an order of magnitude faster, with its speed being over 20 times higher than ViSP on average. This is mainly because MTF uses the Eigen library (Jacob et al., a) for all mathematical computations and this is known to be one of the fastest (Jacob et al., b). Lastly, ViSP trackers are significantly buggy and only completed about 70% of the tested sequences. This section is concluded with a comparison of MTF trackers with 8 state of the art OLTs (Kristan et al., 2016) to validate the suitability of the former for robotics applications. Results are shown in Fig. 7. These were generated using four large publicly available datasets with over 100,000 frames in all to ensure their statistical significance. The tracking performance is evaluated using success rate which is the fraction of total frames where the tracking error is less than a given threshold. Tracking error is measured using alignment error (Roy et al., 2015) that accounts for fine misalignments of pose better than conventional measures like Jaccard index or center location error. More details about the datasets and the evaluation methodology can be found in (Singh et al., 2017; Singh, 2017).

As expected, all the OLTs have far lower SR than both 2 and 8 DOF RBTs for smaller t_p since they are less precise in general (Kristan et al., 2016). They do close the gap as t_p increases
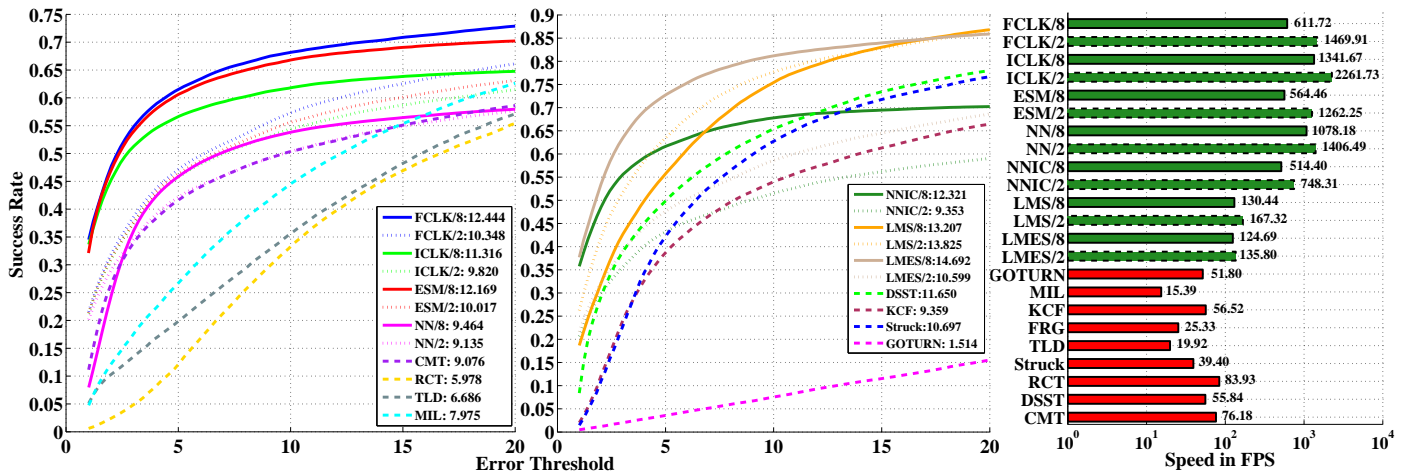
Fig. 7. Comparing OLTs with 2 and 8 DOF RBTs in terms of both accuracy and speed. OLTs are shown in dashed lines while 2 and 8 DOF RBTs are in dotted and solid lines respectively. Accuracy is measured by the tracking success rates (SR) for a range of error thresholds (t_p) over 4 large datasets with more than 100000 frames. OLTs and 2 DOF RBTs were evaluated against 2 DOF ground truth for fairness. Legends in the SR plots show the areas under the respective curves. Speed plot on the right has logarithmic scaling on the x axis for clarity though actual figures are also shown. Original C++ implementations were used for all OLTs. More details about the evaluation methodology and the meanings of several acronyms can be found in (Singh et al., 2017; Singh, 2017).

but none manage to surpass even the best 2 DOF RBTs, which themselves are notably worse than the 8 DOF ones. This difference is even larger if OLTs and 2 DOF RBTs are evaluated against the full pose ground truth which is more indicative of the motions that need to be tracked in real tasks. Though the superiority of DSST and Struck over other OLTs is consistent with VOT results (Kristan et al., 2016), the very poor performance of GOTURN (Held et al., 2016), which is one of the best trackers there, indicates a fundamental difference in the challenges involved within the two paradigms of tracking. The speed plot shows another reason why OLTs are not suitable for high speed tracking applications - they are 10 to 30 times slower than the faster RBTs.

7. Conclusions and Future Work

This paper presented MTF, a modular and extensible open source framework for RBT. It provides highly efficient C++ implementations for several well established trackers that will hopefully address practical tracking needs of the wider robotics community. To this end, it has been designed to integrate well with popular libraries like ROS, OpenCV and ViSP so it can be easily used with existing as well as future projects that require fast trackers. A novel method to decompose RBTs into sub modules was also formulated that facilitates the separat study of appearance models, state space, and search methods.

MTF is still a work in progress and offers several promising avenues of future extensions for each of the sub modules. This includes novel composite SMs especially those that, like GridTracker, run a large number of relatively simple trackers simultaneously and combine their outputs to deduce the state of the tracked patch with greater precision and robustness than any single tracker can possibly provide. A contribution in this direction is the LineTracker (Fig. 3) (Singh, 2017). More generally a variety of multiple-view constraints, e.g. pro-

jective invariants, could be used to constrain and stabilize multiple individual MTF trackers.

One of the most potentially beneficial ways to improve AMs is the incorporation of online learning to update the template. As mentioned in Sec. 3.3, MTF is designed to support this and two related AMs - DFM (Siam, 2015) and PCA (Ross et al., 2008) - are already implemented that respectively utilize offline and online learning. However, more powerful learning methods should be implemented, for example utilizing deep neural networks. ILMs can also be extended with better parameterization that can account for other sources of variations in the appearance of the object patch such as motion blur and occlusion. Another promising extension is the ability to handle depth information from 3D cameras like Kinect whose increasing ubiquity may make this the easiest way to improve tracking performance.

SSMs can be improved by using motion learning from annotated sequences to generate better stochastic samples. As mentioned in Sec. 2.4, the performance of stochastic SMs depends largely on the quality of samples so any progress in this direction should definitely be worthwhile. Further, addition of non rigid SSMs that can go beyond the basic planar projective transforms will be a useful extension albeit with somewhat limited application domain. Similarly, SSMs that support 3D motion estimation are needed to complement the aforementioned depth information processing support in AMs. Finally, improvements can be made to the implementations of existing methods to make them more practically useful. For instance, slower methods like PF, MI, CCRE, NGF and GridTracker can be efficiently parallelized.

Appendix A. $\hat{\mathbf{J}}$ and $\hat{\mathbf{H}}$ for different variants of LK

Appendix A.1. Jacobian

Denoting $\mathbf{w}(\mathbf{x}, \mathbf{p}_s)$ with $\mathbf{w}(\mathbf{p})$ for conciseness ($A = 0$ and $\mathbf{x}$ is constant in this context) and letting $\hat{\mathbf{p}}_t$ denote an estimate of $\mathbf{p}_t$

to which an incremental update is sought, the formulations for $\hat{\mathbf{J}}$ used by FALK and FCLK are:

$$\hat{\mathbf{J}}_{fa} = \frac{\partial f}{\partial \mathbf{I^c}}\bigg|_{\mathbf{I^c}=\mathbf{I}_t(\mathbf{w}(\hat{\mathbf{p}}_t))} \nabla\mathbf{I}_t|_{\mathbf{x}=\mathbf{w}(\hat{\mathbf{p}}_t)} \frac{\partial \mathbf{w}}{\partial \mathbf{p}}\bigg|_{\mathbf{p}=\hat{\mathbf{p}}_t} \quad (A.1)$$

$$\hat{\mathbf{J}}_{fc} = \frac{\partial f}{\partial \mathbf{I^c}}\bigg|_{\mathbf{I^c}=\mathbf{I}_t(\mathbf{w}(\hat{\mathbf{p}}_t))} \nabla\mathbf{I}_t(\mathbf{w})|_{\mathbf{x}=\mathbf{x}_0} \frac{\partial \mathbf{w}}{\partial \mathbf{p}}\bigg|_{\mathbf{p}=\mathbf{p}_0} \quad (A.2)$$

where $\nabla\mathbf{I}_t(\mathbf{w})$ in Eq. A.2 refers to the gradient of I_t warped using $\hat{\mathbf{p}}_t$, i.e. I_t is first warped back to the coordinate frame of I_0 using $\mathbf{w}(\hat{\mathbf{p}}_t)$ to obtain $I_t(\mathbf{w})$ whose gradient is then computed at $\mathbf{x} = \mathbf{x}_0$. It can be further expanded (Baker and Matthews, 2004) as:

$$\nabla\mathbf{I}_t(\mathbf{w})|_{\mathbf{x}=\mathbf{x}_0} = \nabla\mathbf{I}_t|_{\mathbf{x}=\mathbf{w}(\hat{\mathbf{p}}_t)} \frac{\partial \mathbf{w}}{\partial \mathbf{x}}\bigg|_{\mathbf{p}=\hat{\mathbf{p}}_t} \quad (A.3)$$

Since $\nabla\mathbf{I}_t$ is usually the most computationally intensive part of $\mathbf{J}_{fc}$ and $\mathbf{J}_{fa}$, the so called inverse methods approximate this with the gradient of $\nabla\mathbf{I}_0$ for efficiency as this only needs to be computed once. The specific expressions for these methods are:

$$\hat{\mathbf{J}}_{ic} = \frac{\partial f}{\partial \mathbf{I^*}}\bigg|_{\mathbf{I^*}=\mathbf{I}_0(\mathbf{x}_0)} \nabla\mathbf{I}_0|_{\mathbf{x}=\mathbf{x}_0} \frac{\partial \mathbf{w}}{\partial \mathbf{p}}\bigg|_{\mathbf{p}=\mathbf{p}_0} \quad (A.4)$$

$$\hat{\mathbf{J}}_{ia} = \frac{\partial f}{\partial \mathbf{I^c}}\bigg|_{\mathbf{I^c}=\mathbf{I}_t(\mathbf{w}(\hat{\mathbf{p}}_t))} \nabla\mathbf{I}_0|_{\mathbf{x}=\mathbf{x}_0} \frac{\partial \mathbf{w}^{-1}}{\partial \mathbf{x}}\bigg|_{\mathbf{p}=\hat{\mathbf{p}}_t} \frac{\partial \mathbf{w}}{\partial \mathbf{p}}\bigg|_{\mathbf{p}=\hat{\mathbf{p}}_t} \quad (A.5)$$

where the middle two terms in Eq. A.5 are derived from Eqs. A.1 and A.3 by assuming (Hager and Belhumeur, 1998) that $\mathbf{w}(\hat{\mathbf{p}}_t)$ perfectly aligns I_t with I_0, i.e. $I_t(\mathbf{w}) = I_0$ so that

$$\nabla\mathbf{I}_t(\mathbf{w}) = \nabla\mathbf{I}_0 \quad (A.6)$$

In its original paper (Benhimane and Malis, 2007), ESM was formulated as using the mean of the pixel gradients $\nabla\mathbf{I}_0$ and $\nabla\mathbf{I}_t(\mathbf{w})$ to compute $\mathbf{J}$ but, as this formulation is only applicable to SSD, we consider a generalized version (Brooks and Arbel, 2010; Scandaroli et al., 2012) that uses the *difference* between FCLK and ICLK Jacobians:

$$\hat{\mathbf{J}}_{esm} = \hat{\mathbf{J}}_{fc} - \hat{\mathbf{J}}_{ic} \quad (A.7)$$

Appendix A.2. Hessian

For clarity and brevity, evaluation points for the various terms have been omitted in the equations that follow as being obvious from analogy with the previous section.

It is generally assumed (Baker and Matthews, 2004; Benhimane and Malis, 2007) that the second term of Eq. 4 is too costly to compute and too small near convergence to matter and so is omitted to give the so called Gauss Newton Hessian

$$\mathbf{H}_{gn} = \frac{\partial \mathbf{I}}{\partial \mathbf{p}}^T \frac{\partial^2 f}{\partial \mathbf{I}^2} \frac{\partial \mathbf{I}}{\partial \mathbf{p}} \quad (A.8)$$

Though $\mathbf{H}_{gn}$ works very well for SSD (and in fact even better than $\mathbf{H}$ (Baker and Matthews, 2004; Dame and Marchand, 2010)), it is well known (Dame and Marchand, 2010; Scandaroli et al., 2012) to *not* work well with other AMs like MI,

CCRE and NCC for which an approximation to the Hessian *after convergence* has to be used by assuming perfect alignment or $\mathbf{I}_t(\mathbf{w}(\hat{\mathbf{p}}_t)) = \mathbf{I}_0(\mathbf{x}_0)$. We refer to the resultant approximation as the **Self Hessian** $\mathbf{H}_{self}$ and, as this substitution can be made by setting either $\mathbf{I^c} = \mathbf{I}_0(\mathbf{x}_0)$ or $\mathbf{I^*} = \mathbf{I}_t(\mathbf{w}(\hat{\mathbf{p}}_t))$, we get two forms which are respectively deemed to be the Hessians for ICLK and FCLK:

$$\hat{\mathbf{H}}_{ic} = \mathbf{H}_{self}^* = \frac{\partial \mathbf{I}_0}{\partial \mathbf{p}}^T \frac{\partial^2 f(\mathbf{I}_0, \mathbf{I}_0)}{\partial \mathbf{I}^2} \frac{\partial \mathbf{I}_0}{\partial \mathbf{p}} + \frac{\partial f(\mathbf{I}_0, \mathbf{I}_0)}{\partial \mathbf{I}} \frac{\partial^2 \mathbf{I}_0}{\partial \mathbf{p}^2} \quad (A.9)$$

$$\hat{\mathbf{H}}_{fc} = \mathbf{H}_{self}^c = \frac{\partial \mathbf{I}_t}{\partial \mathbf{p}}^T \frac{\partial^2 f(\mathbf{I}_t, \mathbf{I}_t)}{\partial \mathbf{I}^2} \frac{\partial \mathbf{I}_t}{\partial \mathbf{p}} + \frac{\partial f(\mathbf{I}_t, \mathbf{I}_t)}{\partial \mathbf{I}} \frac{\partial^2 \mathbf{I}_t}{\partial \mathbf{p}^2} \quad (A.10)$$

It is interesting to note that $\mathbf{H}_{gn}$ has the exact same form as $\mathbf{H}_{self}$ for SSD (since $\frac{\partial f_{ssd}(\mathbf{I}_0, \mathbf{I}_0)}{\partial \mathbf{I}} = \frac{\partial f_{ssd}(\mathbf{I}_t, \mathbf{I}_t)}{\partial \mathbf{I}} = \mathbf{0}$) so it seems that interpreting Eq. A.8 as the first order approximation of Eq. 4, as in (Baker and Matthews, 2004; Dame and Marchand, 2010), is incorrect and it should instead be seen as a special case of $\mathbf{H}_{self}$.

$\hat{\mathbf{H}}_{fa}$ differs from $\hat{\mathbf{H}}_{fc}$ only in the way $\frac{\partial^2 \mathbf{I}_t}{\partial \mathbf{p}^2}$ and $\frac{\partial \mathbf{I}_t}{\partial \mathbf{p}}$ are computed for the two as given in Eqs. A.11 and A.12 respectively.

$$\frac{\partial^2 \mathbf{I}_t}{\partial \mathbf{p}^2}(fa) = \frac{\partial \mathbf{w}}{\partial \mathbf{p}}^T \nabla^2\mathbf{I}_t \frac{\partial \mathbf{w}}{\partial \mathbf{p}} + \nabla\mathbf{I}_t \frac{\partial^2 \mathbf{w}}{\partial \mathbf{p}^2} \quad (A.11)$$

$$\frac{\partial^2 \mathbf{I}_t}{\partial \mathbf{p}^2}(fc) = \frac{\partial \mathbf{w}}{\partial \mathbf{p}}^T \nabla^2\mathbf{I}_t(\mathbf{w}) \frac{\partial \mathbf{w}}{\partial \mathbf{p}} + \nabla\mathbf{I}_t(\mathbf{w}) \frac{\partial^2 \mathbf{w}}{\partial \mathbf{p}^2} \quad (A.12)$$

where $\nabla^2\mathbf{I}_t(\mathbf{w})$ can be expanded by differentiating Eq. A.3 as:

$$\nabla^2\mathbf{I}_t(\mathbf{w}) = \frac{\partial \mathbf{w}}{\partial \mathbf{x}}^T \nabla^2\mathbf{I}_t \frac{\partial \mathbf{w}}{\partial \mathbf{x}} + \nabla\mathbf{I}_t \frac{\partial^2 \mathbf{w}}{\partial \mathbf{x}^2} \quad (A.13)$$

$\hat{\mathbf{H}}_{ia}$ is identical to $\hat{\mathbf{H}}_{fa}$ except that $\nabla\mathbf{I}_0$ and $\nabla^2\mathbf{I}_0$ are used to approximate $\nabla\mathbf{I}_t$ and $\nabla^2\mathbf{I}_t$. The expression for the former is in Eq. A.5 while that for the latter can be derived by differentiating both sides of Eq. A.3 after substituting Eq. A.6:

$$\nabla^2\mathbf{I}_0 = \frac{\partial \mathbf{w}}{\partial \mathbf{x}}^T \nabla^2\mathbf{I}_t \frac{\partial \mathbf{w}}{\partial \mathbf{x}} + \nabla\mathbf{I}_t \frac{\partial^2 \mathbf{w}}{\partial \mathbf{x}^2}$$

which gives:

$$\nabla^2\mathbf{I}_t(ia) = \left(\frac{\partial \mathbf{w}^{-1}}{\partial \mathbf{x}}\right)^T \left[\nabla^2\mathbf{I}_0 - \nabla\mathbf{I}_t \frac{\partial^2 \mathbf{w}}{\partial \mathbf{x}^2}\right] \frac{\partial \mathbf{w}^{-1}}{\partial \mathbf{x}}$$
$$= \left(\frac{\partial \mathbf{w}^{-1}}{\partial \mathbf{x}}\right)^T \left[\nabla^2\mathbf{I}_0 - \left(\nabla\mathbf{I}_0 \frac{\partial \mathbf{w}^{-1}}{\partial \mathbf{x}}\right) \frac{\partial^2 \mathbf{w}}{\partial \mathbf{x}^2}\right] \frac{\partial \mathbf{w}^{-1}}{\partial \mathbf{x}} \quad (A.14)$$

where the second equality again follows from Eq. A.3 and A.6. Finally, the ESM Hessian corresponding to the Jacobian in Eq. A.7 is the *sum* of FCLK and ICLK Hessians:

$$\hat{\mathbf{H}}_{esm} = \hat{\mathbf{H}}_{fc} + \hat{\mathbf{H}}_{ic} \quad (A.15)$$

References

Adam, A., Rivlin, E., Shimshoni, I., 2006. Robust fragments-based tracking using the integral histogram, in: 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06), IEEE. pp. 798–805.

Babenko, B., Yang, M.H., Belongie, S., 2011. Robust object tracking with online multiple instance learning. IEEE Transactions on Pattern Analysis and Machine Intelligence 33, 1619–1632.

Baker, S., Matthews, I., 2001. Equivalence and efficiency of image alignment algorithms, in: CVPR, pp. 1090–1097. doi:10.1109/CVPR.2001.990652.

Baker, S., Matthews, I., 2004. Lucas-Kanade 20 Years On: A Unifying Framework. IJCV 56, 221–255.

Bartoli, A., 2008. Groupwise geometric and photometric direct image registration. TPAMI 30, 2098–2108.

Benhimane, S., Malis, E., 2007. Homography-based 2D Visual Tracking and Servoing. IJRR 26, 661–676.

Bouguet, J.Y., 2000. Pyramidal Implementation of the Lucas Kanade Feature Tracker: Description of the Algorithm. Technical Report. Intel Corporation Microprocessor Research Labs.

Bradski, G., 2000. OpenCV. Dr. Dobb's Journal of Software Tools .

Brooks, R., Arbel, T., 2010. Generalizing Inverse Compositional and ESM Image Alignment. IJCV 87, 191–212.

Dame, A., Marchand, E., 2010. Accurate real-time tracking using mutual information, in: ISMAR, pp. 47–56.

Danelljan, M., Hager, G., Khan, F.S., Felsberg, M., 2016. Discriminative Scale Space Tracking. IEEE Transactions on Pattern Analysis and Machine Intelligence .

Dellaert, F., Collins, R., 1999. Fast image-based tracking by selective pixel integration, in: Proceedings of the ICCV Workshop on Frame-Rate Vision, pp. 1–22.

Dick, T., Perez, C., Jagersand, M., Shademan, A., 2013. Realtime Registration-Based Tracking via Approximate Nearest Neighbour Search, in: RSS, Berlin, Germany.

Hager, G.D., Belhumeur, P.N., 1998. Efficient Region Tracking With Parametric Models of Geometry and Illumination. TPAMI 20, 1025–1039.

Hager, G.D., Toyama, K., 1998. Xvision: A portable substrate for real-time vision applications. CVIU 69, 23–37.

Hare, S., Golodetz, S., Saffari, A., Vineet, V., Cheng, M.M., Hicks, S.L., Torr, P.H.S., 2016. Struck: Structured Output Tracking with Kernels. IEEE Transactions on Pattern Analysis and Machine Intelligence 38, 2096–2109. doi:10.1109/TPAMI.2015.2509974.

Harris, A., Conrad, J., 2011. Survey of popular robotics simulators, frameworks, and toolkits, in: IEEE Southeastcon,, pp. 243–249. doi:10.1109/SECON.2011.5752942.

Held, D., Thrun, S., Savarese, S., 2016. Learning to Track at 100 FPS with Deep Regression Networks, in: ECCV, pp. 749–765.

Henriques, J.F., Caseiro, R., Martins, P., Batista, J., 2015. High-speed tracking with kernelized correlation filters. IEEE Transactions on Pattern Analysis and Machine Intelligence 37, 583–596.

Jacob, B., Guennebaud, G., et al., a. Eigen: A C++ template library for linear algebra. http://eigen.tuxfamily.org. Accessed: 2018-05-14.

Jacob, B., Guennebaud, G., et al., b. Eigen Benchmark. http://eigen.tuxfamily.org/index.php?title=Benchmark. Accessed: 2018-05-14.

Kalal, Z., Mikolajczyk, K., Matas, J., 2012. Tracking-Learning-Detection. IEEE Transactions on Pattern Analysis and Machine Intelligence 34, 1409–1422. URL: http://dx.doi.org/10.1109/TPAMI.2011.239, doi:10.1109/TPAMI.2011.239.

Kristan, M., Leonardis, A., Matas, J., Felsberg, M., et al., 2016. The Visual Object Tracking VOT2016 Challenge Results, in: ECCV Workshops, pp. 777–823.

Kristan, M., Matas, J., Leonardis, A., et al., 2015. The Visual Object Tracking VOT2015 Challenge Results, in: Proceedings of the IEEE ICCV Workshops, pp. 1–23.

Kwon, J., Lee, H.S., Park, F.C., Lee, K.M., 2014. A geometric particle filter for template-based visual tracking. TPAMI 36, 625–643.

Lucas, B.D., Kanade, T., 1981. An Iterative Image Registration Technique with an Application to Stereo Vision, in: 7th International Joint Conference on Artificial intelligence, pp. 674–679.

Marchand, E., Spindler, F., Chaumette, F., 2005. ViSP for visual servoing: a generic software platform with a wide class of robot control skills. IEEE RAM 12, 40–52. doi:10.1109/MRA.2005.1577023.

Muja, M., Lowe, D.G., 2009. Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration. VISAPP (1) 2, 331–340.

Nebehay, G., Pflugfelder, R., 2015. Clustering of Static-Adaptive correspondences for deformable object tracking, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 2784–2791.

Quigley, M., Conley, K., Gerkey, B.P., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y., 2009. ROS: an open-source Robot Operating System, in: ICRA Workshop on Open Source Software.

Richa, R., Sznitman, R., Taylor, R., Hager, G., 2011. Visual tracking using the sum of conditional variance, in: IROS, pp. 2953–2958. doi:10.1109/IROS.2011.6094650.

Rogerio Richa, Raphael Sznitman, G.H., 2012. Robust Similarity Measures for Gradient-based Direct Visual Tracking. Technical Report. CIRL.

Ross, D.A., Lim, J., Lin, R.S., Yang, M.H., 2008. Incremental Learning for Robust Visual Tracking. IJCV 77, 125–141.

Rousseeuw, P.J., 1984. Least Median of Squares Regression. J. Am. Stat. Assoc. 79, 871–880.

Roy, A., Zhang, X., Wolleb, N., Perez Quintero, C., Jagersand, M., 2015. Tracking benchmark and evaluation for manipulation tasks, in: ICRA, pp. 2448–2453.

Scandaroli, G.G., Meilland, M., Richa, R., 2012. Improving NCC-based Direct Visual Tracking, in: ECCV. Springer, pp. 442–455.

Shum, H.Y., Szeliski, R., . Construction of Panoramic Image Mosaics with Global and Local Alignment. IJCV 36, 101–130.

Siam, M., 2015. CNN Based Appearance Model with Approximate Nearest Neigbour Search. Technical Report. University of Alberta. URL: http://webdocs.cs.ualberta.ca/~vis/mtf/dfm_report.pdf.

Silveira, G., Malis, E., 2007. Real-time visual tracking under arbitrary illumination changes, in: CVPR, pp. 1–6.

Singh, A., 2017. Modular Tracking Framework: A Unified Approach to Registration based Tracking. Master's thesis. University of Alberta. Available at: http://webdocs.cs.ualberta.ca/~vis/mtf/mtf_thesis.pdf.

Singh, A., 2018. Modular Tracking Framework: A Highly Efficient and Extensible Library for Registration based Visual Tracking. http://webdocs.cs.ualberta.ca/~vis/mtf/. Accessed: 2018-05-14.

Singh, A., Jagersand, M., 2017. Modular Tracking Framework: A Fast Library for High Precision Tracking, in: IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Vancouver.

Singh, A., Roy, A., Zhang, X., Jagersand, M., 2016. Modular Decomposition and Analysis of Registration based Trackers, in: CRV.

Singh, A., Siam, M., Jagersand, M., 2017. Unifying Registration based Tracking: A Case Study with Structural Similarity, in: WACV.

Szeliski, R., 2006. Image alignment and stitching: A tutorial. Foundations and Trends in Computer Graphics and Vision 2, 1–104.

Wu, Y., Lim, J., Yang, M.H., 2013. Online Object Tracking: A Benchmark, in: CVPR, pp. 2411–2418. doi:10.1109/CVPR.2013.312.

Zhang, K., Zhang, L., Yang, M.H., 2012. Real-time compressive tracking, in: European Conference on Computer Vision, Springer. pp. 864–877.

Zhang, X., Singh, A., Jagersand, M., 2015. RKLT: 8 DOF real-time robust video tracking combing coarse RANSAC features and accurate fast template registration, in: CRV, pp. 70–77.