SOME OBSERVATIONS ON "A METRIC FOR SETS OF TRAJECTORIES THAT IS PRACTICAL AND MATHEMATICALLY CONSISTENT"

In this document we include some observations on our paper "A metric for sets of trajectories that is practical and mathematically consistent". They are not essential to understand it. *This document is not part of the main paper's submission and it was not subject to any review process. We thus apologize for it not being as well polished.* We use the same setup and notation as explained in Section III of the main paper.

Another example of OSPA producing unintuitive results

Consider the setup in Figure C where two people, A_1 and A_2, pass by each other and two trackers, B and C, produce output trajectories B_1, B_2, C_1 and C_2. Space is measure on the y-axis (bottom to top) and time on the x-axis (left to right). To aid visualization, close lines should be considered on top of each other.

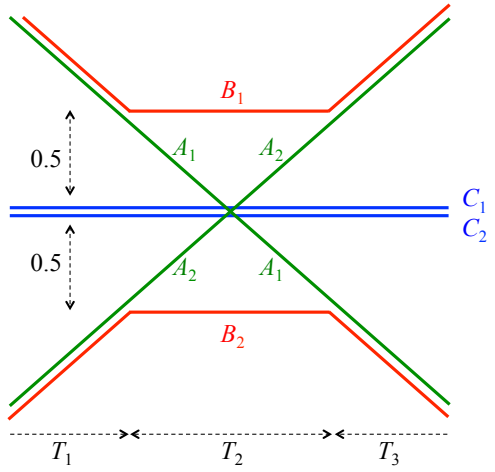

Fig. 7. Counter example that shows that $\mathcal{D}_{MOTA}$ is not a metric.

Let $T_1 = T_3$. By symmetry and the fact that OSPA does not allow associations that change with time we can assume without loss of generality that in computing $\mathcal{D}_{OSPA}(A, B)$ OSPA associates A_1 to B_1 and A_2 to B_2 and that in computing $\mathcal{D}_{OSPA}(A, C)$ OSPA associates A_1 to C_1 and A_2 to C_2.

If $T_2 << T_1 = T_3$, the average distance between A and B can be arbitrarily close to the average distance between A and C. More formally, $\mathcal{D}_{OSPA}(A, C)/\mathcal{D}_{OSPA}(A, B) \to 1$ as $T_1 = T_3 \to \infty$. In other words, OSPA says that tracker C is as good as tracker B while our intuition says that B is the better tracker. B produces good tracks for most of the time and simply makes an identity switch during internval T_2. On the other hand, C is a trivial tracker that always outputs 0 regardless of the input.

MOTA does not define a metric

First recall how to compute MOTA between two sets of trajectories A and B. First compute the CLEAR MOT association Σ_{MOT} between A and B as described in Appendix A. Second, compute a positive linear combination, μ, of three quantities ν_1, ν_2 and ν_3 where: ν_1 is the number of changes of association between trajectories; ν_2 is the number of points of trajectories in A unassociated to any point in B; and ν_3 is the number of points of trajectories in B unassociated to any point in A. MOTA is defined as $1 - \mu$ but since distances must decrease as A and B get similar we define $\mathcal{D}_{MOTA}(A, B) = \mu$.

Lema 5. *$\mathcal{D}_{MOTA}$ is not a metric for any threshold $thr_{MOT} > 0$ or positive linear combination used to define μ.*

Proof. We given an example of three sets A, B and C such $\mathcal{D}_{MOTA}(A, B) > 0$ while $\mathcal{D}_{MOTA}(A, C) = \mathcal{D}_{MOTA}(C, B) = 0$, hence the triangle inequality is violated. Our counter example works for $0.5 < thr_{MOT} < 1$ but by scaling space we can prove the lemma for any thr_{MOT}.

Consider A, B and C as in Figure 1 in the main paper, which we reproduce here for convenience. To aid visualization, close lines should be considered on top of each other and transitions in A almost instantaneous. Time is on the x-axis (left to right) and space is on the y-axis (bottom to top).

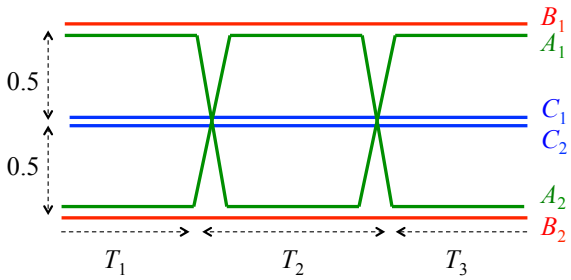

Fig. 8. Counter example that shows that $\mathcal{D}_{MOTA}$ is not a metric.

When we compute $\mathcal{D}_{MOTA}(A, C)$, Σ_{MOT} associates A_1 to C_1 and A_2 to C_2 for all times t. (we can interchange C_1 and C_2 since they are equal). There are no changes in association because the distances computed are always less than 0.5 and $0.5 < thr_{MOT}$, thus $\nu_1 = 0$. In addition, there are no unallocated tracks so $\nu_2 = \nu_3 = 0$. Therefore $\mathcal{D}_{MOTA}(A, C) = \mu = 0$ regardless of the coefficients in the linear combination μ. Similarly we get $\mathcal{D}_{MOTA}(C, B) = 0$.

When we compute $\mathcal{D}_{MOTA}(A, B)$ we find the following: before T_1, Σ_{MOT} associates A_1 to B_1; after T_1 and before $T_1 + T_2$, the distance between A_1 and B_1 becomes $1 > thr_{MOT}$ and so Σ_{MOT} changes association and matches A_1 to B_2. After $T_1 + T_2$, the distance between A_1 and B_2 becomes $1 > thr_{MOT}$ and so Σ_{MOT} changes association back to A_1 matches to B_1. We change association twice and so $\nu_1 > 0$ We just focused on who A_1 matches to. The remaining tracks are matched to each other so $\nu_2 = \nu_3 = 0$. In short $\mathcal{D}_{MOTA}(A, B) = \mu > 0$.

□

A. How to compute our metrics

Probably the simplest way to compute $\mathcal{D}_{comp}$ is using a Matlab CVX code [11] like in Figure 9. It is not coded for efficiency but for readability. The input to the code is the scalar `alpha` and the tensor `D`. The variable `D` is an `m` by `T` by

```
function [W , cvx_optval] = Dcomp(D, alpha)

m = size(D,1);
T =  size(D,2);

D =  shiftdim(D,2);

cvx_begin quiet
    variable W(m,m,T);

    swicost = 0;
    for t = [1 : T-1]
        swicost = swicost +
         sum(sum(abs(   W(:,:,t+1) - W(:,:,t)   )));
    end

    minimize    sum(sum(sum(D.*W)))
                +    alpha*swicost;

    subject to
        W >= 0;
        for t = [1:T]
            sum(W(:,:,t),1) == 1;
            sum(W(:,:,t),2) == 1;
        end
cvx_end
```

Fig. 9. Simple Matlab CVX code to compute $\mathcal{D}_{comp}$.

`m` tensor that holds the matrices of distances $\{D^{AB}(t)\}$ as specified in the definition of $\mathcal{D}_{comp}$ in equation (2). The output of the code is the tensor `W` that holds the optimal values of the association matrices $\{W(t)\}$ and the scalar `cvx_optval` that is created after the CVX code runs and that holds the value of $\mathcal{D}_{comp}(A, B)$.

In the code, the matrix norm we use in the switch cost is the element-wise 1-norm but the code can be quickly modified to consider other norms.

With a little bit more of effort it is possible to write a faster code, also in Matlab CVX, that now can exploit sparsity. In particular, we now have a new input variable, the threshold level `maxvalthre`, that imposes that if $D_{ij}^{AB}(t) >$ `maxvalthre` then $W_{ij}(t) = 0$. The other input and output variables are the same.

Finally, we can use the built-in LP solver in Matlab to produce yet another implementation of $\mathcal{D}_{comp}$. This allows us to use the mixed-integer linear program solver of Matlab to force the $W(t)$ matrices to be permutation matrices, which allows us to approximate the value of $\mathcal{D}_{nat}$. Now we have one extra input, the flag `forceint`. The other input and output variables are the same as before. To estimate $\mathcal{D}_{nat}$ we just need to set `forceint` to 1.

Table I shows the run-time between the different codes for problems of different size. Recall that m = number of (extended) trajectories and T = number of frames. Time is measured in seconds.

When looking at these results, it is important to recall that our work is about a metric to compare sets of trajectories and not about preforming tracking in any way. Consider the ground truth A and output of a tracker B operating at 10 frames per second. Using the second code we can compute $\mathcal{D}(A, B)$ for

TABLE I
RUN-TIME OF THE DIFFERENT CODES

m	T	Code 1	Code 2	Code 3
10	100	27	4	4
20	100	-	62	45
40	100	-	534	-
10	500	-	35	51
10	1000	-	85	248

$T = 1000$ frames in about 85 seconds. This does not mean that we can compute $\mathcal{D}_{comp}$ in real-time or in an online fashion at 11 frames per second. We do not receive the points from the trajectories of A and B sequentially in time. To compare two sets of trajectories A and B we need to operate offline and in a full-batch setting.

```
function [W , cvx_optval] = Dcomp(D, alpha, maxvalthre)

m = size(D,1);
T =  size(D,2);

Dtild      =       reshape(D,m*T,m);
[ix1, ix2] =       find( Dtild < maxvalthre  );
ix3        =       find( Dtild < maxvalthre  );
numele     =       length(ix3);
Dlin       =       Dtild(ix3);

modifeye = eye(T); modifeye(T,T) = 0;
diffmat = sparse(  kron( modifeye -
            diag(ones(T-1,1),1) , eye(m) )    );
onesmat = sparse(  kron(eye(T) , ones(1,m))  );

cvx_begin quiet
    variable Wsparse(numele);
    W = sparse( ix1 , ix2 , Wsparse , m*T , m);

    minimize   sum(sum(  Dlin.*Wsparse  ))   +
                       alpha*sum(sum(
                       abs(  diffmat*W  )));
    subject to
        sum(W,2) == 1;
        onesmat*W == 1;
        Wsparse >= 0;
cvx_end
```

Fig. 10. Matlab CVX that exploits sparsity to compute $\mathcal{D}_{comp}$.

```
function [W , cvx_optval] = Dcomp(D, alpha,
                maxvalthre, forceint)

m = size(D,1);
T =  size(D,2);

Dtild =  shiftdim(D,2);

Esparseix =  (Dtild(:,:,2:T) < maxvalthre)
           |   (Dtild(:,:,1:T-1) < maxvalthre);
Esparseix = Esparseix(:);

Dtild = Dtild(:);
Dsparseix = Dtild < sparsemaxval;
ixintvars = find(Dsparseix);

bothsparseix = sparse([Dsparseix ; Esparseix]);
numvars = sum(bothsparseix);

objcoeff = [   Dtild' , alpha*ones(1,m*m*(T-1))   ];
objcoeff = objcoeff(bothsparseix);

modifeye = eye(T); modifeye(T,T) = 0;
diffmat = kron(   modifeye  - diag(ones(T-1,1),1)
, eye(m*m));
diffmat = sparse(diffmat);
diffmat = sparse(   [  diffmat(1:end-m*m,:) ,
             -kron( eye(T-1) , eye(m*m) ) ;
                -diffmat(1:end-m*m,:),
                -  kron( eye(T-1) , eye(m*m) ) ]);
diffmat = diffmat(:,bothsparseix);

constsumone = sparse([      kron( eye(m*T) ,
           ones(1,m)   ),zeros(m*T,m*m*(T-1));
           kron(eye(T),kron(ones(1,m),eye(m)) )  ,
           zeros(m*T,m*m*(T-1))    ]);
constsumone = constsumone(:,bothsparseix);

if (forceint == 1)
    [WE , cvx_optval] = intlinprog(objcoeff,ixintvars,
    diffmat, zeros(size(diffmat,1),1),constsumone,
    ones(size(constsumone,1)
    1),zeros(numvars,1),ones(numvars,1));
else
    [WE , cvx_optval] = linprog(objcoeff,
    diffmat, zeros(size(diffmat,1),1),constsumone,
    ones(size(constsumone,1),
    1),zeros(numvars,1),ones(numvars,1));
end

W = WE(1:n*n*T);
```

Fig. 11. Using Matlab's LP solver and sparsity to compute $\mathcal{D}_{comp}$ and estimate $\mathcal{D}_{nat}$.