Reference Positioning Engine & Measurements Database for Indoor Time-Delay Wi-Fi Client Positioning

Leor Banin, Ofer Bar-Shalom, Nir Dvorecki, and Yuval Amizur

Abstract

The following whitepaper describes a reference positioning engine (PE) and time-delay measurements database that can be used for the development and performance evaluation of passive, time-delay based, Wi-Fi scalable location systems. The database includes fine-time measurements (FTM) timestamps that were collected in an indoor Wi-Fi venue. As a turnkey for using this database, a reference Matlab® code of a positioning engine (PE) that can be executed by a Wi-Fi client station, is provided. The reference PE demonstrates the usage of the database and enables to analyze the performance of the PE using additional information included in the database files.

Index Terms

Geolocation, Positioning, Position Estimation, Location Estimation, Indoor navigation, Fine timing measurement, FTM, Time delay estimation, WLAN, Wi-Fi, IEEE 802.11

I. Introduction

The following paper outlines the Matlab® code of a reference positioning engine (PE) that uses time delay measurements for estimating and tracking the position of a client receiver. The receiver is assumed to be equipped with IEEE802.11 WLAN (Wi-Fi) chipset that is capable of fine-timing measurements [1]. To test the PE's performance, the Matlab code is provided

L. Banin, O. Bar-Shalom, N. Dvorecki and Y. Amizur are with Intel's Location Core Division, 94 Em Hamoshavot Rd., Petah Tikva 49527, Israel. leor.banin@intel.com, ofer.bar-shalom@intel.com, nir.dvorecki@intel.com, yuval.amizur@intel.com

along with a set of files that contain time delay measurements collected in an indoor venue. The files contain both real-life time delay timestamps of Wi-Fi units operating in the venue that were collected *passively* by a client that was roaming within the venue. Additional files, containing synthetic measurements generated using a network simulator, enable the testing of network capabilities that are currently not supported by the real Wi-Fi hardware available, but are planned to be part of the future IEEE802.11az standard. The measurement files consist of all the information required for enabling the analysis of PE performance and the accuracy of its position estimates, (for which "ground truth" (reference) position information of the client receiver is included). The Wi-Fi network deployed in the venue consists of broadcasting stations (bSTA), which periodically broadcast measurement frames at a rate of 5Hz. These frames enable the other bSTA to measure their times of arrival (ToA) and publish their measurement results in their subsequent broadcasting events. This passive location mode is called "collaborative time of arrival" (CToA). A detailed description of this concept can be found in [3]. The client station that passively listens to the network broadcast transmissions and measures their ToA is called a "CToA station" (cSTA). These terms are used within the code. It should be emphasized that even though the terms used throughout the document correspond to so-called "non-trigger-based scalable location", the positioning engine implementation is generic and supports both "trigger-based" (TB HEz-Multi User ranging) and "non-trigger-based" (VHTz-Single User/SU ranging). See [2] for details on these protocols. The remainder of this paper is organized as follows. Section II contains an annotated listing of the Matlab® code included in the files provided. Section III contains a detailed description of the measurement data included in the database. Section IV-A describes the contents of the software package, its installation and its usage. Finally, Section V provides some numerical examples of the performance measures that can be obtained by executing the PE code on the given inputs.

II. MATLAB® CODE STRUCTURE

A. *The* `MainPE.m` *Function*

The function `MainPE.m`is the main function used for executing the environment. The function receives as an input a description of the environment configuration file, which includes all the information for executing the test: the name of the `*.csv` measurement file, configuration parameters of the test venue and so on. The `MainPE` function first extracts the test configuration parameters, then extracts the measurement table out of the `*.csv` file provided, executes the positioning engine, and then plots the results.

```
1
function mainPE(configMfile)
3 if exist('configMfile','var')
    configFunc = str2func(configMfile);
5 else
    configFunc = @testConfig;
7 end
cfg = configFunc();
9 [measTable, bPos, refPos] = ReadFile(cfg); % Read measurements file
[posEst,pValid] = RunPE(cfg,measTable); % Run PE
11 % pValid - are the entries produced for every packet received by the client
posMat = posEst(:,pValid)';
13 refPosMat = refPos(pValid,:);
PlotResults(cfg,posMat,bPos,refPosMat) % plot results
15 end
```

Listing 1. The `MainPE.m` Function Listing

Following is the listing of the `testConfig.m` script. Multiple scripts are included in the code distribution, each is targeted for running a different test configuration.

```
1 function cfg = testConfig()
% % 6rsp @ 5hz, Real data set
3 % % Tx time is randomly spread.
cfg.measFile = 'RD.csv';
5 cfg.name = cfg.measFile(1:end-4);
cfg.initPos = [-1.5; 7; 1.7];
7 % file contain measurement from 6 bSTA, we remove 2 of them.
cfg.bSTA2remove = [1,6]; % for using all bSTAs: cfg.bSTA2remove = [];
```

```
% value must be the equal or bigger than the biggest bSTA id used.
cfg.maxSTAid = 6;
cfg.disableClkTracking = 0;
% Map image filename, xMinMax, yMinMax
cfg.mapFile = 'arc_map.png';;
cfg.xMinMax = [-44.5301, 6.7192];
cfg.yMinMax = [-15.3423, 35.9070];
end
```

Listing 2. The `testConfig.m` Function Listing

B. The `RunPE.m` Function

The function `RunPE.m` is the function that executes the EKF-based PE. The function loops over all the measurements. The function prepares each measurement to be fed into an EKF-based PE for processing. A measurement is defined as,

$$z = (\mathrm{ToA}_j + \nu_j) - (\mathrm{ToD}_i + \nu_i) \tag{1}$$

where ToA_j denotes the timestamp corresponding to the time of arrival measured by the jth receiver, and ToD_i denotes the timestamp corresponding to the time of departure measured by the ith transmitter. As each timestamp is reported at the local clock of the measuring unit, the scalars $\nu_i,\ \nu_j$, are introduced. These scalars correspond to the clock offsets of the jth receiver and ith transmitter that are measured w.r.t. to the client clock. By definition, if the ToA is measured by the client directly then, $\nu_j \equiv 0$. This process is implemented by lines 24-57 of the following listing. Depending on the type of the measurement, the EKF is called on line 59 or 61 with the appropriate parameters.

The `RunPE` loops over all the measurements contained in the measurements table. The first action in the loop is to ensure proper initialization of the EKF states and variables. The bSTAs clock offsets states must be initialized prior to the processing of any measurements that involve these bSTAs. As seen from lines 15-21, the first measurement used for initializing the clock offset states, is a measurement of a packet that was received by the client directly. This initializes the clock offset state of the transmitting bSTA with some rather coarse accuracy (as the distance between the client and the bSTA is neglected at this stage). The initialization of the additional offsets states is implemented in lines 24-47, which implement the initialization phase of all the

offsets for all the other bSTA at multiple scenarios of measurement types and the order they arrive. Any measurement, which is used for state initialization, is discarded afterwards and is not used for further states updating.

```
function RunPE(cfg, measTable)
measN = size(measTable,1);
KF = KFclass; % Kalman Filter Constructor
INIT_KF = 1;  % KF Initialization flag
lastPacketId = -999;
lastTxid = -999;
staOffsetInit = zeros(6,1); % vector to indicate if offset was init
posEst = nan(3,measN);
pValid = false(measN,1);
initPos = cfg.initPos;
for k = 1:measN
    [packetId,type,txId,rxId,txTime,rxTime,staTx,staRx]=ParseMeasLine(measTable(k,:));
    if INIT_KF
        if type==0 % Rx by cSTA
            KF.initKF(rxTime,initPos);
            INIT_KF = 0;
            initOffset = txTime - rxTime;
            KF.SetOffsetByIndex(initOffset,txId);
            staOffsetInit(txId) = 1;
        end
        continue;
    else
        if staOffsetInit(txId)==0 || (rxId>0 && staOffsetInit(rxId)==0)
            if rxId>0 && staOffsetInit(txId)==0 && staOffsetInit(rxId)==0
                % bSTA to bSTA - both not init --> cannot do anything
                continue;
            elseif rxId < 0 && staOffsetInit(txId)==0
                % bSTA to Client - init bSTA
                initOffset = rxTime -  txTime + KF.Xoffset(txId);
                KF.SetOffsetByIndex(initOffset,txId);
                staOffsetInit(txId) = 1;
                continue;
            elseif staOffsetInit(txId)==0 && staOffsetInit(rxId)==1
                % for k = 1:measN% bSTA to bSTA - tx not init
```

```
36                  initOffset = txTime - rxTime + KF.Xoffset(rxId);
                    KF.SetOffsetByIndex(initOffset,txId);
38                  staOffsetInit(txId) = 1;
                    continue;
40              elseif staOffsetInit(txId)==1 && staOffsetInit(rxId)==0
                    % bSTA to bSTA - rx not init
42                  initOffset = rxTime - txTime + KF.Xoffset(txId);
                    KF.SetOffsetByIndex(initOffset,rxId);
44                  staOffsetInit(rxId) = 1;
                    continue;
46              end
            end
48          % Every packet is defined uniquely by its packetId and txId
            % If the new measurement differ from the last packet
50          % the time of new packet is obtained using KF.GetKFtime
            if packetId ~= lastPacketId || txId ~= lastTxid % detect new packet
52              kfTime = KF.GetKFtime(txTime,txId);  % convert txTime to cSTA clock
                lastPacketId = packetId;
54              lastTxid = txId;
            end
56          % prepare the measurement for KF
            meas = rxTime - txTime;
58          if type % bSTA to bSTA
                posEst(:,k) = KF.Run(meas,kfTime,staTx,staRx,'BSTA_TO_BSTA'); % Run KF
60          else    % bSTA to cSTA
                posEst(:,k) = KF.Run(meas,kfTime,staTx,[],'RX_BY_CSTA'); % Run KF
62              pValid(k) = true;
            end
64      end % INIT_KF
        % ----------------------------------------
66      % Outlier filtering code may be added here
        % ----------------------------------------
68  end % for k = 1:measN
    end
70
    % ---------------------------------------------------------------------
72  function [pId,type,txId,rxId,txTime,rxTime,staTx,staRx] = ParseMeasLine(measLine)
    % line Format: packetId,type,txId,rxId,txPos,rxPos,txTime,rxTime,refPos
74  pId = measLine(1);
```

```
  type = measLine(2);
76 txId = measLine(3);
  rxId = measLine(4);
78 txTime = measLine(11);
  rxTime = measLine(12);
80 %-------------------------------------------------------------
  staTx.index = txId;           % Transmitting bSTA ID
82 staTx.pos = measLine(5:7)';   % Transmitting bSTA position
  staRx.index = rxId;           % Receiver bSTA ID (not used if client)
84 staRx.pos = measLine(8:10)';  % Receiver bSTA position (not used if client)
  %-------------------------------------------------------------
86 end

88 function c = isClient(id)
  c = id < 0;
90 end
```

Listing 3. The `RunPE.m` Function Listing

C. *The `KFClass.m` Class*

The EKF code is implemented in an object-oriented fashion. Class-based object-oriented programming defines an object constructor, which is a special type of subroutine called to create an object. The EKF object constructor is initialized on line no. 4 of the `RunPE.m` function using the code line `KF = KFclass(cfg);`. This instantiates an object called `KF`, supporting the properties and methods defined by `KFClass`, which are implemented in `KFClass.m`. Such object has the following properties:

```
 properties
2        X               % States vector
         P               % States covariance matrix
4        sysNoisePos     % position system noise
         sysNoiseClk     % clock system noise
6        t               % KF time
         stateN          % number of KF states
8        clkN            % number of bSTA clocks being tracked
         statePerClk     % number of states per clock
10       Xoffset_Index   % clock offsets states index subvector
         Xdrift_Index    % clock drifts states index subvector
```

```
        Xoffset         % clock offsets states subvector
        Xdrift          % clock drifts states subvector
        NO_CLK_TRACKING % Flag to effectively disable clock tracking
end
```

Listing 4. The `KFclass` Properties Listing

The class properties are addressed by `KFclass` object constructor subroutine.

```
function obj = KFClass(cfg)
    obj.clkN        = cfg.maxSTAid; % should be init to the maximal bSTA index
    obj.stateN      = 3 + 2*obj.clkN; % Total number of filter states
    obj.Xoffset_Index  = 3 + 1:obj.statePerClk:obj.stateN;
    obj.Xdrift_index  = 3 + 2:obj.statePerClk:obj.stateN;
    obj.sysNoiseClk = [1e-15, 0.01e-6] .^2; % Set system noise covariance matrix
                                            % values for clock offset and drift.
    obj.sysNoisePos = [1.0, 1.0, 0.1] .^2;  % Set system noise covariance matrix
                                            % values for the client position.
    % -------------------------------------------
    % Outlier filtering code may be added here...
    % -------------------------------------------
end
```

Listing 5. The `KFClass` Method Constructor Listing

As was describe in [3], the EKF system model is defined by the following recursive equation,

$$\mathbf{x}_k = \mathbf{F}_k \mathbf{x}_{k-1} + \mathbf{w}_k, \quad k \geq 0 \tag{2}$$

where the index k denotes the discrete time-step. The vector $\mathbf{x}_k$ denotes an $N \times 1$ states vector, which describes the parameters being estimated and tracked by the filter. The states vector for the client-mode consists of the client's position coordinates and per-bSTA clock parameters (clock offset and clock offset change rate/drift). The size of the EKF state vector is thus: $N = 3 + 2M$, where M denotes the number of bSTAs being received by the cSTA (both directly and indirectly),

$$\begin{aligned}
\mathbf{p}_k &\triangleq [x_k, y_k, z_k]^T \\
\tilde{\boldsymbol{\nu}}_{m,k} &\triangleq [\nu_{m,k}, \dot{\nu}_{m,k}]^T \\
\tilde{\boldsymbol{\nu}}_k &\triangleq [\tilde{\boldsymbol{\nu}}_{1,k}^T, \ldots, \tilde{\boldsymbol{\nu}}_{M,k}^T]^T \\
\mathbf{x}_k &\triangleq \left[\mathbf{p}_k^T, \tilde{\boldsymbol{\nu}}_k^T\right]^T
\end{aligned} \tag{3}$$

The vector $\mathbf{w}_k$ denotes a random $N \times 1$ model noise vector, which described the uncertainties in the system model and has the following statistical properties:

$$\begin{aligned} E\{\mathbf{w}_k\} &= \mathbf{0} \\ E\{\mathbf{w}_k \mathbf{w}_k^T\} &= \mathbf{Q}_k \\ E\{\mathbf{w}_k \mathbf{w}_j^T\} &= \mathbf{0},\ \forall k \neq j \\ E\{\mathbf{w}_k \mathbf{x}_k^T\} &= \mathbf{0},\ \forall k \end{aligned} \tag{4}$$

The system-model noise covariance, $\mathbf{Q}_k$ is assumed to have the following structure:

$$\mathbf{Q}_k = \Delta t \cdot \begin{bmatrix} \mathbf{Q}_{\mathbf{p},k} & \mathbf{0} \\ \mathbf{0} & \mathbf{I}_M \otimes \tilde{\mathbf{Q}}_{c,0} \end{bmatrix} \tag{5}$$

where $\otimes$ denotes Kronecker multiplication and,

$$\mathbf{Q}_{\mathbf{p},k} \triangleq \begin{bmatrix} \tilde{\sigma}_x^2 & 0 & 0 \\ 0 & \tilde{\sigma}_y^2 & 0 \\ 0 & 0 & \tilde{\sigma}_z^2 \end{bmatrix} \tag{6}$$

$$\mathbf{Q}_{c,k}^m \triangleq \begin{bmatrix} \sigma_c^2 & 0 \\ 0 & \dot{\sigma}_c^2 \end{bmatrix} \tag{7}$$

1) EKF Initialization: The EKF initialization phase occurs before the first execution of the EKF (i.e., before the `Run` method is called for the first time). During this phase, the initial values of the filter states vector and its corresponding covariance matrix are set. The filter states covariance matrix is defined as follows,

$$\mathbf{P}_k = E\{(\mathbf{x}_k - \bar{\mathbf{x}}_k)(\mathbf{x}_k - \bar{\mathbf{x}}_k)^T\} \tag{8}$$

where $\bar{\mathbf{x}}_k \triangleq E\{\mathbf{x}_k\}$. When the filter is initialized the state-covariance matrix is assumed to be,

$$\mathbf{P}_0 = \begin{bmatrix} \tilde{\mathbf{P}}_{\mathbf{p},0} & \mathbf{0} \\ \mathbf{0} & \mathbf{I}_M \otimes \tilde{\mathbf{P}}_{c,0} \end{bmatrix} \tag{9}$$

where $\sigma_{\nu,0}$, $\sigma^2_{\dot{\nu},0}$ denote the initial values for the standard deviations of the clock offsets and drifts, and $\tilde{\mathbf{P}}_{\mathbf{p},0}$ denotes the initial value of states covariance matrix given by,

$$\tilde{\mathbf{P}}_{\mathbf{p},0} \triangleq \begin{bmatrix} \sigma^2_{x,0} & 0 & 0 \\ 0 & \sigma^2_{y,0} & 0 \\ 0 & 0 & \sigma^2_{z,0} \end{bmatrix} \tag{10}$$

$$\tilde{\mathbf{P}}_{c,0} \triangleq \begin{bmatrix} \sigma^2_{\nu,0} & 0 \\ 0 & \sigma^2_{\dot{\nu},0} \end{bmatrix} \tag{11}$$

```
function obj = initKF(obj,initTs,initPos)
    obj.t = initTs;         % Initialize EKF time
    Xinit = zeros(obj.stateN,1);
    Xinit(1:3) = initPos;   % Initialize EKF client position states
    ErrSigma    = 10e-3;    % Initialize clock offset STD
    PPMsigma    = 100e-6;   % Initialize clock drift STD
    latSigma    = 10;       % Initialize horizontal position STD
    Zsigma      = 0.5;      % Initialize vertical position STD
    PinitPerClk = diag([...
    ErrSigma...
    PPMsigma...
    ].^2);
    PinitPerPos = diag([...
        latSigma...
        latSigma...
        Zsigma...
     ].^2);
     Pinit = obj.BuildBigMat(PinitPerPos,PinitPerClk);
     % init KF:
     obj.X  = Xinit; % initialize the KF states vector
     obj.P  = Pinit; % initialize the KF states covariance matrix
     % ---- Update easy-access vectors ------
     obj.Xoffset = obj.X(obj.Xoffset_Index);
     obj.Xdrift  = obj.X(obj.Xdrift_index);
     % ---------------------------------------
end
```

Listing 6. The `initKF` Method Listing

2) EKF Prediction: The EKF time as well as EKF states, are predicted according to the ToA of the received packet. The predicted state-vector is defined as,

$$\hat{\mathbf{x}}_{k|k-1} = \mathbf{F}_k \hat{\mathbf{x}}_{k-1|k-1} \tag{12}$$

The predicted covariance estimate:

$$\mathbf{P}_{k|k-1} = \mathbf{F}_{k-1} \mathbf{P}_{k-1|k-1} \mathbf{F}_{k-1}^T + \mathbf{Q}_{k-1} \tag{13}$$

The method `predictKF` implements the EKF prediction step. It receives a handle to the `KFClientClass`, the states transition matrix, $\mathbf{F}_k$, and the system noise covariance matrix $\mathbf{Q}_k$. Equations (12)-(13) are realized by the method `predictKF`.

```
function predictKF(obj,F,Q)
    obj.X = F * obj.X;           % calculate X_k/k-1
    obj.P = F * obj.P * F' + Q; % calculate P_k/k-1
end
```

Listing 7. The `predictKF` Method Listing

The dynamic system-model linear transfer function denoted by $\mathbf{F}_k$, is an $N \times N$ block-diagonal matrix defined as follows.

$$\mathbf{F}_k \triangleq \begin{bmatrix} \mathbf{I}_3 & \mathbf{0} \\ \mathbf{0} & \mathbf{I}_M \otimes \tilde{\mathbf{F}}_k \end{bmatrix} \tag{14}$$

where Δt corresponds to the elapsed time between two consecutive discrete time steps and,

$$\tilde{\mathbf{F}}_k \triangleq \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \tag{15}$$

The states transition matrix, $\mathbf{F}_k$, is generated using the `CreateF` method as follows.

```
function F = CreateF(obj,dt)
    FperPos  = eye(3); % per position
    FperClk  = [1 dt ; 0 1 ]; % per tracked bSTA clock
    F        = obj.BuildBigMat(FperPos,FperClk);
end
```

Listing 8. The `CreateF` Method Listing

The system noise covariance matrix, $\mathbf{Q}_k$, is created using the method `CreateQ` as follows.

```
1
function Q = CreateQ(obj, dt)
3     QperPos = diag( dt * obj.sysNoisePos );
      if obj.NO_CLK_TRACKING && dt > 10e-3
5     % setting high system noise, making the offset and drift
      % states to have a very little on the updated outcome..
7         obj.X(obj.Xoffset_Index) =  obj.X(obj.Xoffset_Index) + 1e-6*randn(obj.clkN,1);
          QperClk = diag( [1e-6 1e-6].^2 );
9     else
          QperClk = diag( dt * obj.sysNoiseClk );
11    end
      Q       = obj.BuildBigMat(QperPos,QperClk);
13 end
```

Listing 9. The `CreateQ` Method Listing

The `CreateQ` method includes a mechanism that effectively disables the clock tracking of the EKF for measurements that are separated (in time) by 10ms or more. This mechanism is implemented in lines 4-8 of the method. This effect is achieved by adding a relatively large random noise to the clock offset with a standard deviation of 1μsec and setting the filter clock offset and drift system-model noise to relative large levels of 1μs and 1 ppm, respectively.

The `CreateQ` and `CreateF` methods use the `BuildBigMat` method as a utility for constructing the larger matrices, which correspond to all the states.

```
1 function out = BuildBigMat(obj,posMat,clkMat)
      out = zeros(obj.stateN);
3     out(1:3,1:3) = posMat;
      for k = 1:obj.clkN
5         s = 4 + 2*k;
          out(s:s+1,s:s+1) = clkMat;
7     end
  end
```

Listing 10. The `BuildBigMat` Method Listing

3) EKF Measurement Model: The EKF measurement model is defined as,

$$z_k = h(\mathbf{x}_k) + v_k \tag{16}$$

where z_k is a ToF measurement. The function $h(\mathbf{x}_k)$ denotes the nonlinear measurement model transfer function, and v_k denotes the additive measurement noise that has the following statistical properties:

$$\begin{aligned} E\{v_k\} &= 0 \\ E\{v_k v_j\} &= \sigma_m^2 \delta_{kj} \end{aligned} \tag{17}$$

where δ_{kj} denotes the Kronecker delta.

The method `CreateR` generates the respective measurement noise variance according to the measurement type, `measType`.

```
function R = CreateR(obj,measType)
    if strcmp(measType, 'BSTA_TO_BSTA') % bSTA to bSTA
        R = (3e-9)^2;
    elseif strcmp(measType,'RX_BY_CSTA') % Rx by client
        R = (6e-9)^2; 'RX_BY_CSTA') % higher meas. var. since client is mobile
    end
end
```

Listing 11. The `CreateR` Method Listing

The noise-free measurement models for bSTA $\rightarrow$ cSTA and bSTA $\rightarrow$ bSTA are given by (18) and (19), respectively.

$$h_\ell(\mathbf{x}_k) = \frac{1}{c}\|\mathbf{p}_k - \mathbf{q}_i\| - \nu_{i,k} \tag{18}$$

$$h_l(\mathbf{x}_k) = \frac{1}{c}\|\mathbf{q}_j - \mathbf{q}_i\| + \nu_{j,k} - \nu_{i,k} \tag{19}$$

Since the measurement transfer function, $\mathbf{h}(\cdot)$ is nonlinear, it cannot be applied to estimate the measurements covariance matrix directly. Instead we linearize $\mathbf{h}(\cdot)$ by replacing it with its first order Taylor series expansion, calculated around $\hat{\mathbf{x}}_{k|k-1}$:

$$\mathbf{h}(\mathbf{x}_k) \cong \mathbf{h}(\hat{\mathbf{x}}_{k|k-1}) + \mathbf{H}_k \cdot (\mathbf{x}_k - \hat{\mathbf{x}}_{k|k-1}) \tag{20}$$

where the notation, $\hat{\mathbf{x}}_{n|m}$ represents the estimate of $\mathbf{x}$ at time n given observations up to and including time $m \leq n$. The matrix $\mathbf{H}_k$ denotes the Jacobian of the measurement model function

vector $\mathbf{h}(\cdot)$, which is a $J \times N$ matrix defined as,

$$\mathbf{H}_k \triangleq \begin{bmatrix} \frac{\partial h_1}{\partial x_1} & \frac{\partial h_1}{\partial x_2} & \cdots & \frac{\partial h_1}{\partial x_N} \\ \vdots & & \ddots & \vdots \\ \frac{\partial h_J}{\partial x_1} & \frac{\partial h_J}{\partial x_2} & \cdots & \frac{\partial h_J}{\partial x_N} \end{bmatrix}$$
$$[\mathbf{H}_k]_{ij} \equiv \left.\frac{\partial h_i}{\partial x_j}\right|_{\mathbf{x}=\hat{\mathbf{x}}_{k|k-1}} \tag{21}$$

The method `CreateH` generates the linearized measurement model matrix lines (a line per measurement).

```
function H = CreateH(obj,type,staTx,staRx)
    H = zeros(1,obj.stateN);
    if strcmp(type,'RX_BY_CSTA')
       H(1:3)= (obj.X(1:3)' - staTx.pos')/norm (obj.X(1:3)' - staTx.pos')/3e8;
       H(4 + (staTx.index-1)*obj.statePerClk) = -1; % ToD
    elseif strcmp(type,'BSTA_TO_BSTA')
       H(4 + (staRx.index-1)*obj.statePerClk) =  1; % ToA
       H(4 + (staTx.index-1)*obj.statePerClk) = -1; % ToD
    end
end
```

Listing 12. The `CreateH` Method Listing

The Jacobian is obtained by calculating the partial derivatives of (18)-(19) w.r.t. to the states vector. Equations (22)-(23) define compact expressions for the matrix $\mathbf{H}_k$ lines, depending on the measurement type. Equation (22) defines the matrix line entry for a bSTA→cSTA measurement, while (23) defines the corresponding line for the bSTA→bSTA measurement case.

$$[\mathbf{H}_k]_\ell = \left[\frac{(\mathbf{p}_k - \mathbf{q}_n)^T}{c\|\mathbf{p}_k - \mathbf{q}_n\|}, -(\mathbf{e}_i \otimes \tilde{\mathbf{e}}_1)^T\right] \tag{22}$$
$$[\mathbf{H}_k]_l = \left[\mathbf{0}_3^T, (\mathbf{e}_j - \mathbf{e}_i) \otimes \tilde{\mathbf{e}}_1\right]^T \tag{23}$$

where $\mathbf{e}_i$ denotes the ith column of an $M \times M$ identity matrix and $\tilde{\mathbf{e}}_1 = [1, 0]^T$. Equations (22)-(23) are realized by the `CreateHx` method. Notice that since the measurements are fed into the EKF one by one (and not as a vector of measurements), the matrix $\mathbf{H}_k$ is never used as a "matrix", but only as a line-vector, according to the current measurement type.

```
function hi = CreateHx(obj,type,staTx,staRx)
    if strcmp(type,'RX_BY_CSTA')
```

```
        Xpos = obj.X(1:3)';
        hi = norm(Xpos - staTx.pos')/3e8 - obj.Xoffset(staTx.index);
    elseif strcmp(type,'BSTA_TO_BSTA')
        tof = norm(staTx.pos - staRx.pos)/3e8;
        hi = obj.Xoffset(staRx.index) - obj.Xoffset(staTx.index) + tof;
    end
end
```

Listing 13. The `CreateHx` Method Listing

The timestamps associated with a packet are translated into the client's local timebase using the method `KF.GetKFtime`. This method is called from the `RunPE.m` function (see line 52)

```
function KFtime = GetKFtime(obj,localTime,index)
    KFtime = localTime - obj.Xoffset(index);
end
```

Listing 14. The `GetKFtime` Method Listing

D. EKF Update

The measurements included in the received packet are updated according to the new EKF predicted time. The update is done per individual measurement (as opposed to joint updating a vector of measurements), such that a matrix inverse in (26) is not required. The measurement innovation (i.e., the measurement residual) is obtained using,

$$\tilde{y}_k = z_k - h(\hat{\mathbf{x}}_{k|k-1}) \tag{24}$$

For a vector of measurements, the innovation covariance matrix is calculated using,

$$\mathbf{S}_k = \mathbf{H}_k \mathbf{P}_{k|k-1} \mathbf{H}_k^T + \mathbf{R}_k \tag{25}$$

The near-optimal Kalman filter gain is obtained using,

$$\mathbf{K}_k = \mathbf{P}_{k|k-1} \mathbf{H}_k^T \mathbf{S}_k^{-1} \tag{26}$$

Using (26), the updated state estimate is calculated as,

$$\hat{\mathbf{x}}_{k|k} = \hat{\mathbf{x}}_{k|k-1} + \mathbf{K}_k \tilde{\mathbf{y}}_k \tag{27}$$

And the updated estimate covariance,

$$\mathbf{P}_{k|k} = (\mathbf{I} - \mathbf{K}_k \mathbf{H}_k)\, \mathbf{P}_{k|k-1} \tag{28}$$

Equations (26)-(28) are executed by calling the following `updateKF` method.

```
function updateKF(obj,z,H,R,hi)
    K     = obj.P * H' / (H * obj.P * H' + R); % calculate K - filter gain
    obj.X = obj.X + K*(z - hi);      % calculate X - Update states
    obj.P = obj.P - K * H * obj.P;   % calculate P - Update states covariance
end
```

Listing 15. The `updateKF` Method Listing

The `Run` method is called per measurement and executes the EKF algorithm, which is detailed in [3].

```
function posEst = Run(obj,z,t,staTx,staRx,measType)
    dt    = max(0,t - obj.t); % time difference from last update
    obj.t = obj.t + dt;       % update current KF time

    F = obj.CreateF(dt);      % generate the states transition matrix
    Q = obj.CreateQ(dt);      % generate the system noise covariance matrix
    obj.predictKF(F,Q);       % Do KF prediction

    % ---- Update easy-access vectors ------
    obj.Xoffset = obj.X(obj.Xoffset_Index); % Clock offsets vector
    obj.Xdrift  = obj.X(obj.Xdrift_index); % Clock drift vector
    % --------------------------------------

    H  = obj.CreateH(measType,staTx,staRx);
    hi = obj.CreateHx(measType,staTx,staRx); % generate the linearized meas. matrix
    R  = obj.CreateR(measType); % generate meas. noise covariance matrix

    % ------------------------------------------------
    % Measurement outlier filtering may come here...
    % ------------------------------------------------

    obj.updateKF(z,H,R,hi); % Do KF update
    % ---- Update easy-access vectors ------
    obj.Xoffset = obj.X(obj.Xoffset_Index); % Clock offsets vector
    obj.Xdrift  = obj.X(obj.Xdrift_index); % Clock drift vector
    % --------------------------------------
     posEst   = obj.X(1:3); % updated current position estimation vector
```

```
end  % function Run
```

Listing 16. The `Run` Method Listing

As seen in lines 51-55 of `RunPE`, the system updates the EKF time variable (`kfTime`, by calling the method `GetKFtime`) only once per transmitted packet, (which is uniquely identified by its packet ID, `packetId`, and its transmitter ID, `txId`). This condition effectively causes the EKF states' prediction to be done only once per transmitted packet. Only the first measurement associated with the packet that is fed into the EKF will have a $\Delta t = (\texttt{kfTime}_k - \texttt{kfTime}_{k-1}) > 0$ (see line 6 in `Run` method). For each additional measurement corresponding to that packet, `GetKFtime` is not called and thus, `kfTime` is not updated. Consequently, $\Delta t = (\texttt{kfTime}_k - \texttt{kfTime}_{k-1}) \equiv 0$. As the states' prediction relies on the values of $\mathbf{F}_k$ and $\mathbf{Q}_k$, (see line 12 in `Run` method), both of which depend on Δt, having $\Delta t = 0$ effectively disables the states' prediction, and enables only the states' update per measurement.

E. Outlier Rejection

The EKF code contains no outlier measurement rejection mechanisms, but only "hooks" to places where such mechanisms might be added into the code. Examples of EKF outlier rejection mechanisms are described in e.g., [5, Chapter 15.2].

F. Performance Visualization

The following function implements the results visualization and enables to present the positioning error and the trajectories of the cSTA on a map of the venue. The architectural blueprint map of the test venue is provided as `*.png` file named `'arc_map.png'`. Additional maps may be added similarly in future releases of this database. The positioning error statistics is presented by means of the empirical cumulative distribution function (ECDF) of the norm between the estimated position and its ground-truth reference.

```
2 function PlotResults(cfg,posMat,bPos,refPos)
  % Plot Ref. and Estimated Trajectories on Map
4 figure(cfg.MapFigNum);clf;hold on;axis equal
  ImgData = imread(cfg.mapFile);
6 FloorPlan = image(ImgData, ...
```

```
            'XData', cfg.xMinMax, ...
            'YData', fliplr(cfg.yMinMax));
axis([-12, 6.7, -5 28])
uistack(FloorPlan, 'bottom');
set(gca,'Ydir','normal');
plot(posMat(:,1),posMat(:,2),'b.','MarkerSize',6,'MarkerFaceColor','b')
plot(refPos(:,1),refPos(:,2),'r.','MarkerSize',8,'LineWidth',4)
plot(bPos(:,1),bPos(:,2),'ro','MarkerSize',8,'LineWidth',3);
axis([min(cfg.xMinMax), max(cfg.xMinMax), -5 25])
xlabel('x [m]')
ylabel('y [m]')
title(cfg.measFile)
grid on; axis equal; box on
leg1 = legend('$\hat\mathbf{p}$','$\mathbf{p}$','bSTA','Location','southwest');
set(leg1,'Interpreter','latex');

% Plot Positioning Error Empirical CDF (ECDF)
Nsamp = length(posMat(:,1));
posErr = zeros(Nsamp,1);
for kk = 1:Nsamp
    posErr(kk) = norm(posMat(kk,:)-refPos(kk,:));
end
[X,Y] = CalcEcdf(posErr);
figure; hold on;grid on; box on
plot(X,Y,'b-','LineWidth',2)
xlabel('Position Error [m]')
ylabel('CDF [%]')
title(cfg.measFile)
end

function [x,y] = CalcEcdf(x)
N = length(x);
x = sort(x);
y = (1:N)/N*100;
x=x(:);y=y(:);
end
```

Listing 17. The `PlotResults` Function Listing

III. Indoor Time-Delay Measurements Database

The reference code of the PE is distributed along with several measurements files. These files enable the code users to evaluate the performance of the reference PE and additional PEs that can accept the measurement format. The measurements files included are listed in Section IV-B3. Additional files may be added in the future. Each file is a `*.csv` file format that spreads over 14 columns (A-N), where each line represents a measurement with the following data format (the text in (`Courier`) font refers to the naming of that parameter in the reference code):

- Column A - packet identification (`pId`)
- Column B - packet type (`type`): 0 - bSTA→bSTA, 1 - bSTA→client (cSTA)
- Column C - Transmitting bSTA identification (`txId`)
- Column D - Receiving bSTA identification (`rxId`)
- Columns E-G - Transmitting bSTA position coordinates (x,y,z) (`txPos`)
- Columns H-J - Receiving bSTA position coordinates (x,y,z) (`rxPos`)
- Column K - Broadcast packet time of transmission (`txTime`) at the time-base of the transmitter
- Column L - Broadcast packet time of reception (`rxTime`) at the time-base of the receiver
- Columns M-O - Client receiver ground truth (reference) position (x,y,z) coordinates (`refPos`)

The database includes a single file with real-measurements collected from the network and 4 files containing simulated measurements. The simulated measurements were generated using skewed-clock modeling as described in [4]. It should be emphasized that this model assumes only Gaussian-distributed errors (on the clock and the measurement noise), but does not include any biases or other effects resulting from the physical propagation channel (e.g. shadowing, multipath reflections etc.). Namely, the timestamps were generated under a line-of-sight (LoS) assumption between all the receivers, and thus correspond to the geometric distance between the transmitters and the receivers. The files currently contained in the database are the following:

- `RD.csv` - real-data (RD) measurements file containing measurements of 6 bSTAs, broadcasting measurement frames at a rate of 5Hz, where the transmission time of each bSTA is randomly spread over time.
- `SD.csv` - simulated-data (SD) measurements file containing measurements of 6 bSTAs,

broadcasting measurement frames at a rate of 5Hz. The clock drifts were simulated with $\ddot{\nu}$ =0.01ppm/sec, $\tilde{\sigma}$=3ns, $\bar{\sigma}$=6ns (see [4]). The bSTA broadcasts transmission times are randomly spread.

- `SDMU.csv` - IEEE 802.11az multi-user ranging simulated-data measurements file containing measurements of 6 bSTAs, broadcasting measurement frames at a rate of 5Hz. The clock drifts were simulated with $\ddot{\nu}$ =0.01ppm/sec, $\tilde{\sigma}$=3ns, $\bar{\sigma}$=6ns. The bSTA broadcasts transmission times are all time-aligned with minimal spacing of about 50μs.
- `SDSMU.csv` - IEEE 802.11az multi-user ranging simulated-data measurements file containing measurements of 6 bSTAs, broadcasting measurement frames at a rate of 5Hz. The clock drifts were simulated with $\ddot{\nu}$ =0.01ppm/sec, $\tilde{\sigma}$=3ns, $\bar{\sigma}$=6ns. The bSTA broadcasts transmission times are aligned in two groups (Split MU): [1,2,5] are 100ms offset from [3,4,6]).

IV. Software Download, Installation & Usage

The following section describes the how to install and execute the simulation environment.

A. Download Information

The package may be downloaded from the Intel's repository on the GitHub website: www.github.com/intel/Reference-PE-and-Measurements-DB-for-WiFi-Time-based-Scalable-Location.

B. Software Installation

To install the software, simply create a folder on your PC's local drive and copy all the files into it. The folder should contain the following files:

1) Package Code Files:

1) `MainPE.m` - contains the main function to run the simulation environment.
2) `configFunc` - contains a function for extracting the test configuration.
3) `ReadFile.m` - contains a function for reading the `*.csv` measurement files.
4) `RunPE.m` - a function that implements the .
5) `KFclass.m` - contains a class of methods implementing the EKF.
6) `PlotResults.m` - contains a function that implements the results visualization (trajectory plotting and CDF).

2) Test Configuration Files:

1) `testConfig.m` - executes the default real-data (RD) measurements file (`RD.csv`).
2) `testConfigSD.m` - executes the simulated-data (SD) measurements file (`SD.csv`).
3) `testConfigSD_NT.m` - executes the simulated-data (SD) measurements file (`SD.csv`), but with no time-tracking (NT). This effect is acheived by manipulating the model-noise covariance matrix in the method `CreateQ`.
4) `testConfigSDMU.m` - executes the simulated-data of passive multi-user (MU) ranging measurements file (`SDMU.csv`).
5) `testConfigSDMU_NT.m` - executes the simulated-data of passive multi-user (MU) ranging measurements file (`SDMU.csv`), but with no time-tracking (NT).

3) Supplementary Files:

1) `RD.csv` - real-data (RD) measurements file.
2) `SD.csv` - simulated-data (SD) measurements file
3) `SDMU.csv` - IEEE 802.11az multi-user ranging simulated-data.
4) `SDSMU.csv` - IEEE 802.11az multi-user ranging simulated-data.
5) `arc_map.png` - an architectural blueprint figure of the test venue.
6) `LICENSE.md` - an text file containing the BSD-3-Clause license terms.
7) `README.md` - a text file describing the repository contents.

C. Software Execution

The code was tested on Matlab® R2016b (but may also run on its earlier versions).

To execute the code:

1) Set current Matlab® to the folder to which the `*.zip` was extracted (or add that folder to the Matlab® search path).

2) In the Matlab® Command Window type: `main(TesConfigFileName)`,
   where `TesConfigFileName` denotes the name of the Test Configuration `*.m` file.

The following options are presently supported:

- `mainPE()` - executes the PE with the `RD.csv` measurements file .
- `mainPE('testConfigSD')` - executes the PE with the `SD.csv` measurements file.
- `mainPE('testConfigSDMU')` - executes the PE with the `SDMU.csv` measurements file.
- `mainPE('testConfigSDSMU')` - executes the PE with the `SDSMU.csv` measurements file.
- `mainPE('testConfigSDMU_NT')` - executes the PE with the `SDMU.csv` measurements file, but without clock tracking.

- `mainPE('testConfigSDSMU_NT')` - executes the PE with the `SDSMU.csv` measurements file, but without clock tracking.

V. Sample Performance

The following section provides performance examples of the reference PE when operated on the input measurements files included in the database. The output is generally available as depicted in Fig. 2, which includes a plot of the estimated and reference client trajectories on the venue's blueprint map, along with the positioning errors' empirical cumulative distribution functions (ECDF). The bSTAs are located as shown in Fig. 1. For

Fig. 1. bSTA Positions

the following tests, the measurements coming from bSTA#1 and bSTA#6 were ignored by enabling the line `cfg.bSTA2remove = [1,6];` on `testConfig.m`. To enable measurements of all bSTAs this line should be modified as `cfg.bSTA2remove = [];`. Other combinations of bSTAs may be enabled accordingly. Fig. 2 provides an illustration of the expected output when using `SD.csv` as an input.

Fig. 3 compares the positioning error ECDF of all the simulated-data cases.

Fig. 3 compares the positioning error ECDF of the real measured data - with and without outlier rejection. As explained, the simulated data (SD) is generated using Gaussian-distributed additive errors. Due the nature of these errors, outliers and range-bias errors are not present in this data-set. Consequently, the EKF, which is optimized for the Gaussian-case, performs well (as indicated by the "SD" and "SDMU" cases). For the case of split-MU, there are two pairs of MU AP + bSTA. The inferior performance exhibited for the case where the EKF does not track the clock offsets and drifts ("NT"), can be mainly attributed to the poor resulting geometry of the two pairs. Enabling the data from the two additional bSTAs improves the results to the level of "SDMU".

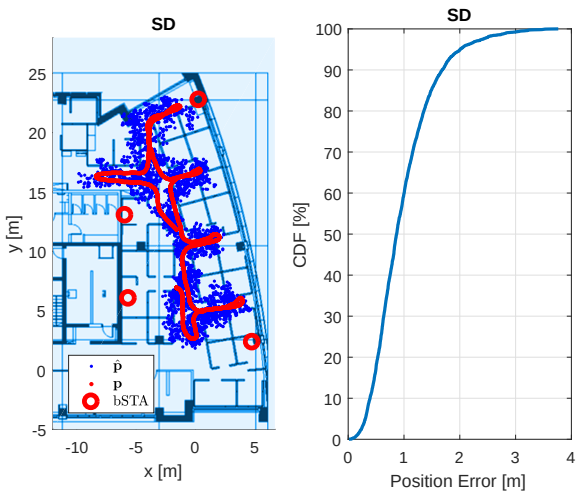

Fig. 2. Reference and Estimated Trajectories for `SD.csv`

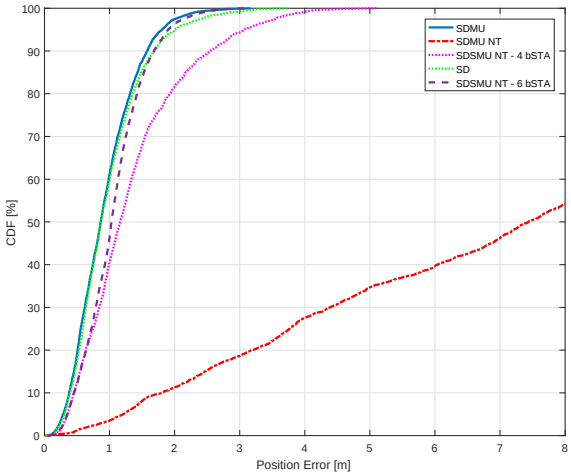

Fig. 3. Comparison of Positioning Error CDFs for Simulated Data

As can be seen from Fig. 4, the positioning accuracy using the real-measured data is somewhat inferior compared to the simulated data. As aforementioned, the performance degradation is mainly attributed to the outliers and range-biases, which are introduced, by nature, in real-network measured data, but do not exist in the model generating the simulated data. The outliers may be filtered-out using outlier-rejection mechanisms that can be applied to the EKF code. An example of the resulting performance using some mild, heuristic outlier-rejection is depicted in Fig. 4 for reference.

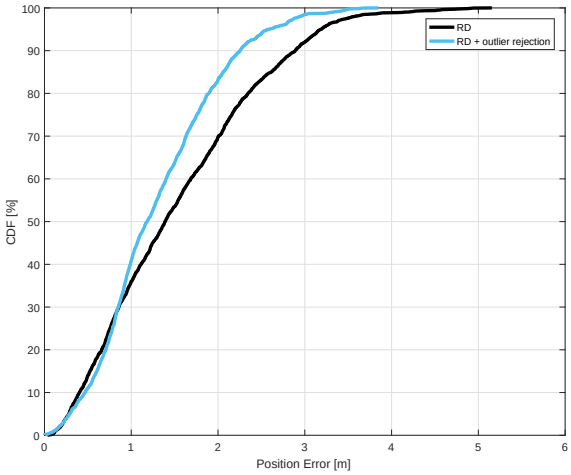

Fig. 4. Comparison of Positioning Error CDFs for Simulated Data

VI. Summary & Conclusions

We presented a reference Matlab® implementation of a positioning engine for indoor time-delay Wi-Fi client positioning. The positioning engine is based on a linearized version of the well-known Kalman filter algorithm. The source code is provided along with a measurements database for assessing and developing time-delay based Wi-Fi client positioning systems.

REFERENCES

[1] *IEEE Std 802.11™-2016 (Revision of IEEE Std 802.11-2012) - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications,* IEEE 802.11 Working Group, December 7th, 2016.

[2] *P802.11az™/D0.1 Draft Standard for Information Technology - Telecommunications and Information Exchange Between Systems Local and Metropolitan Area Networks - Specific Requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications - Amendment 8: Enhancements for Positioning*, IEEE 802.11 Working Group, November, 2017.

[3] L. Banin, O. Bar-Shalom, N. Dvorecki and Y. Amizur, "High-Accuracy Indoor Geolocation using Collaborative Time of Arrival - Whitepaper," doc.: IEEE 802.11-17/1387R0, Sept. 2017, [available online: https://mentor.ieee.org/802.11/dcn/17/11-17-1387-00-00az-high-accuracy-indoor-geolocation-using-collaborative-time-of-arrival-ctoa-whitepaper.pdf]

[4] L. Banin, O. Bar-Shalom, N. Dvorecki and Y. Amizur, "Wi-Fi Positioning using Collaborative Time of Arrival (CToA)," *IEEE Trans. on Wireless Communications*, under review, Jan. 2018.

[5] P. D. Groves, *Principles of GNSS, Inertial, and Multisensor Integrated Navigation Systems*, Artech House Boston-London, 2008.