Applied Computer Vision

David Vernon
Carnegie Mellon University Africa

vernon@cmu.edu
www.vernon.eu

Lecture 23

Video Image Processing

Object Tracking II: Kalman Filter

Tracking Objects

Motion models

- $\boldsymbol{x}_t = \boldsymbol{x}_{t-1} + (\boldsymbol{x}_{t-1} - \boldsymbol{x}_{t-2}) + \varepsilon$
- $\boldsymbol{x}_t = \boldsymbol{x}_{t-1} + speed_{av} + \varepsilon$
- $\boldsymbol{x}_{t+k} = \boldsymbol{x}_t + k\ speed_{av}$

If the previous positions or velocity are known, we can predict the position of the object in the next frame(s) and restrict the search for it

But how do we deal with the inherent uncertainty (or error ε) in the measurements?

Tracking Objects

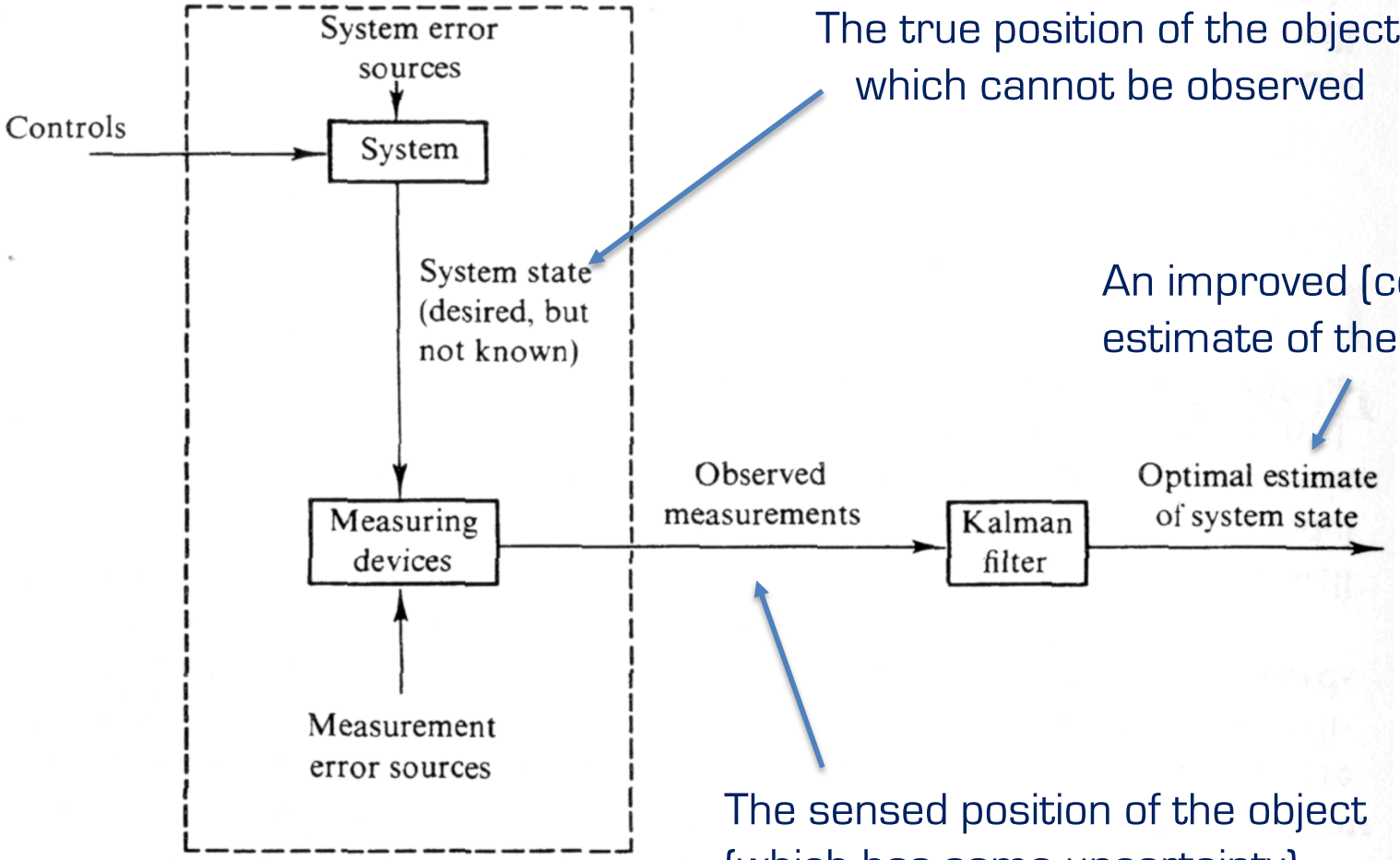
- If the dynamics of the moving object are known, we can predict the position of the object in the current image
- This information can be combined with the actual but error-prone image observation to achieve more accurate estimate
- The most common methods are
  - Kalman filter
  - Particle filter

Tracking Objects

Kalman filter - overview

- Optimal linear estimator
- Recursive data-processing algorithm
  - Does not require all previous data to be stored and re-processed
  - When a new measurement is taken
- Incorporates all information made available to it
  - Processes all available measurements (regardless of precision)
  - To estimate the current value of the variables of interest

Tracking Objects



Source: P. S. Maybeck, Stochastic Models, Estimation, and Control, Vol. 1, 1979

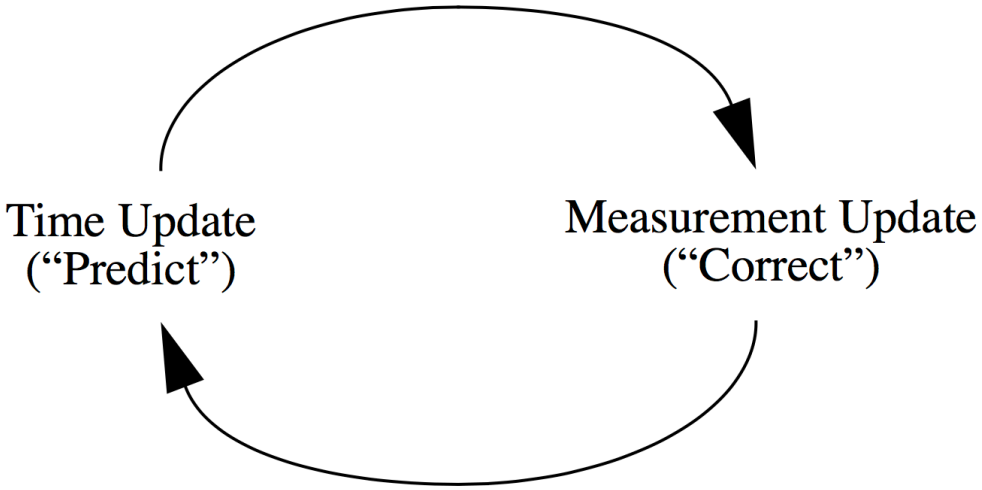
Tracking Objects

Kalman filter - overview

- Two-step process
  - Prediction
    - Produce estimates of the current state variables
    - Produce estimates of their uncertainties
  - Correction (based on new observation)
    - Observe the outcome of the next measurement (corrupted with some amount of error, including random noise)
    - Update the estimates using a weighted average of the prediction and the measurement
      - More weight being given to estimates with higher certainty

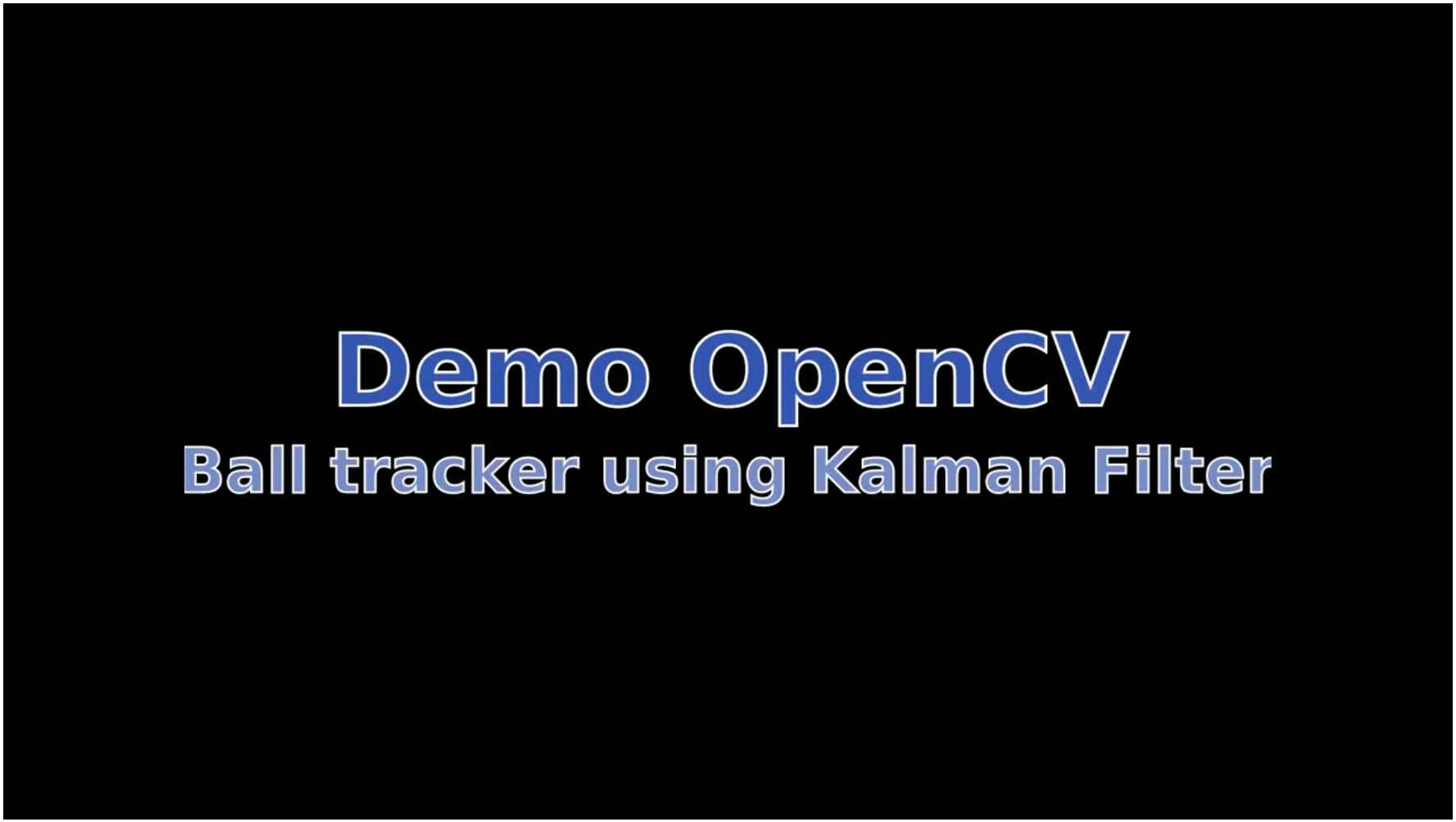
Tracking Objects

Kalman filter - overview



Credit: Welch and Bishop (2006)

Tracking Objects



Credit: https://www.youtube.com/watch?v=sG-h5ONsj9s&feature=youtu.be

Tracking Objects

Kalman filter in detail ...

"The Kalman filter is a set of mathematical equations that provides an efficient computational (recursive) means to estimate the state of a process, in a way that minimizes the mean of the squared error"

Welch and Bishop (2006)

Tracking Objects

Discrete Kalman Filter

Addresses the general problem of trying to estimate the state $x \in \Re^n$ of a discrete-time controlled process that is governed by the linear stochastic difference equation

$$x_k = Ax_{k-1} + Bu_{k-1} + w_{k-1}$$

with a measurement $z \in \Re^m$

$$z_k = Hx_k + v_k$$

Tracking Objects

Discrete Kalman Filter

n x n matrix relates
the state at the previous time step $k-1$
to the state at time step k

n x l matrix relates
the optional control input u
($u \in \Re^l$) to the state x

State at time k
$x \in \Re^n$

$$x_k = Ax_{k-1} + Bu_{k-1} + w_{k-1}$$

Process noise:
white,
i.e. temporally uncorrelated,
and Gaussian,
i.e. normal probability
distribution with zero mean
and variance Q

$$p(w) \sim N(0, Q)$$

$$z_k = Hx_k + v_k$$

Tracking Objects

Discrete Kalman Filter

$$x_k = Ax_{k-1} + Bu_{k-1} + w_{k-1}$$

$$z_k = Hx_k + v_k$$

Measurement at time k

$z \in \Re^m$

m x n matrix relates the state x_k to the measurement z_k

Measurement noise

white,

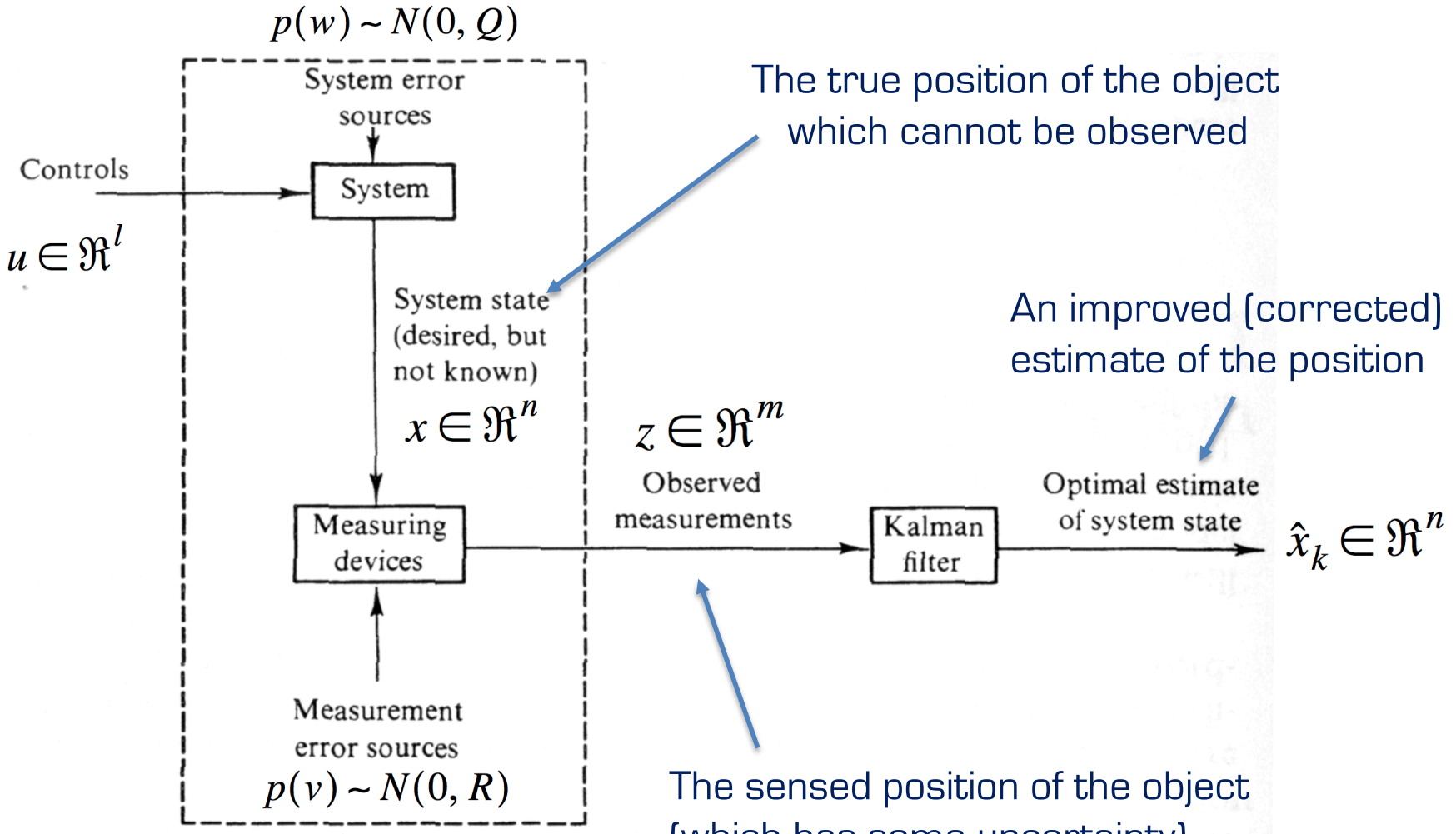
i.e. temporally uncorrelated,

and Gaussian,

i.e. normal probability distribution with zero mean and variance R

$p(v) \sim N(0, R)$

Tracking Objects



Credit: P. S. Maybeck, Stochastic Models, Estimation, and Control, Vol. 1, 1979

Tracking Objects

Discrete Kalman Filter

This is the result we want:
the *a posteriori* estimate of the state estimate at time step k

$\hat{x}_k \in \Re^n$ ← a posteriori estimate at time step k given measurement z_k

$e_k^- \equiv x_k - \hat{x}_k^-$ ← *a priori* estimate error

$e_k \equiv x_k - \hat{x}_k$ ← *a posteriori* estimate error

$P_k^- = E[e_k^- e_k^{-T}]$ ← *a priori* estimate error covariance:
a measure of the uncertainty of the value of $\hat{x}_k^- \in \Re^n$
predicted by the filter immediately before obtaining z_k

$P_k = E[e_k e_k^T]$ ← *a posteriori* estimate error covariance:
a measure of the uncertainty of the value of $\hat{x}_k \in \Re^n$
computed by the filter after integrating measurement z_k
with the prediction $\hat{x}_k^- \in \Re^n$

Tracking Objects

Discrete Kalman Filter

$$\hat{x}_k = \hat{x}_k^- + K(z_k - H\hat{x}_k^-)$$

a posteriori estimate of the state computed as a linear combination of the *a priori* estimate and a weighted difference between the actual measurement and a measurement prediction

This difference is called the measurement innovation or the residual:

It reflects the discrepancy between the predicted measurement and the actual measurement z_k

The n x m matrix K is chosen to be the gain that minimizes the *a posteriori* error covariance

Tracking Objects

Discrete Kalman Filter

$$K_k = P_k^- H^T (H P_k^- H^T + R)^{-1}$$

$$= \frac{P_k^- H^T}{H P_k^- H^T + R}$$

The n x m matrix K is chosen to be the gain that minimizes the *a posteriori* error covariance (see Welch and Bishop 2006)

This is the key to the Kalman filter … how to best combine the information we have: an *a priori* estimate of the state and the difference between the actual measurement pf the state and the predicted measurement of the state

Tracking Objects

Discrete Kalman Filter

$$K_k = P_k^- H^T (H P_k^- H^T + R)^{-1}$$

$$= \frac{P_k^- H^T}{H P_k^- H^T + R}$$

The n x m matrix K is chosen to be the gain that minimizes the *a posteriori* error covariance (see Welch and Bishop 2006)

If the uncertainty of the measurement is low (i.e. the measurement error covariance R_k approaches zero), the gain K weights the residual more heavily

$$\lim_{R_k \to 0} K_k = H^{-1}$$

and the estimated state depends more on the actual measurement

$$\hat{x}_k = \hat{x}_k^- + K(z_k - H\hat{x}_k^-)$$

Tracking Objects

Discrete Kalman Filter

$$K_k = P_k^- H^T (H P_k^- H^T + R)^{-1}$$

$$= \frac{P_k^- H^T}{H P_k^- H^T + R}$$

The n x m matrix K is chosen to be the gain that minimizes the *a posteriori* error covariance (see Welch and Bishop 2006)

On the other hand, if the uncertainty of the *a priori* estimate is low (i.e. the *a priori* estimate error covariance approaches zero), the gain K weights the residual less heavily

$$\lim_{P_k^- \to 0} K_k = 0$$

and the estimated state depends more on the *a priori* estimate

$$\hat{x}_k = \hat{x}_k^- + K(z_k - H\hat{x}_k^-)$$

Tracking Objects

Discrete Kalman Filter

Another way of thinking about the weighting by K is that as the measurement error covariance R approaches zero, the actual measurement z_k is "trusted" more and more, while the predicted measurement $H\hat{x}_k^-$ is trusted less and less. On the other hand, as the *a priori* estimate error covariance P_k^- approaches zero the actual measurement z_k is trusted less and less, while the predicted measurement $H\hat{x}_k^-$ is trusted more and more.

Welch and Bishop (2006)

Tracking Objects

Discrete Kalman Filter

- Estimates a process by using a form of feedback control:

  1. The filter estimates the process state at some time
  2. It then obtains feedback in the form of (noisy) measurements

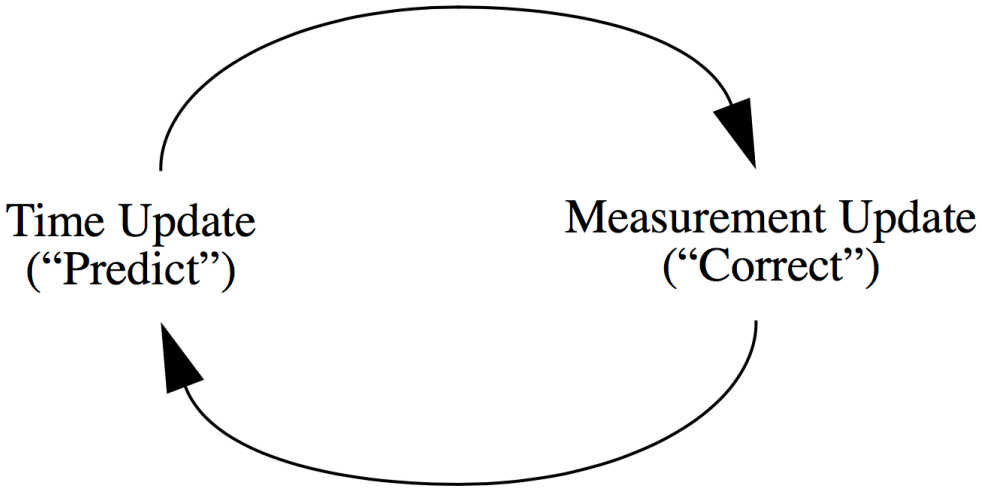
- Two groups of equations (each corresponding to these two phases)

  1. Time update equations (prediction)
     - Project forward in time the current state and error covariance estimates
     - To obtain the *a priori* estimates for the next time step

  2. Measurement update equations (correction)
     - Incorporate a new measurement into the *a priori* estimate
     - To obtain an improved *a posteriori estimate*

Tracking Objects

Discrete Kalman Filter



Credit: Welch and Bishop (2006)

Tracking Objects

Discrete Kalman Filter

- Time update equations:

$$\hat{x}_k^- = A\hat{x}_{k-1} + Bu_{k-1}$$

$$P_k^- = AP_{k-1}A^T + Q$$

- Measurement update equations:

$$K_k = P_k^- H^T (HP_k^- H^T + R)^{-1}$$

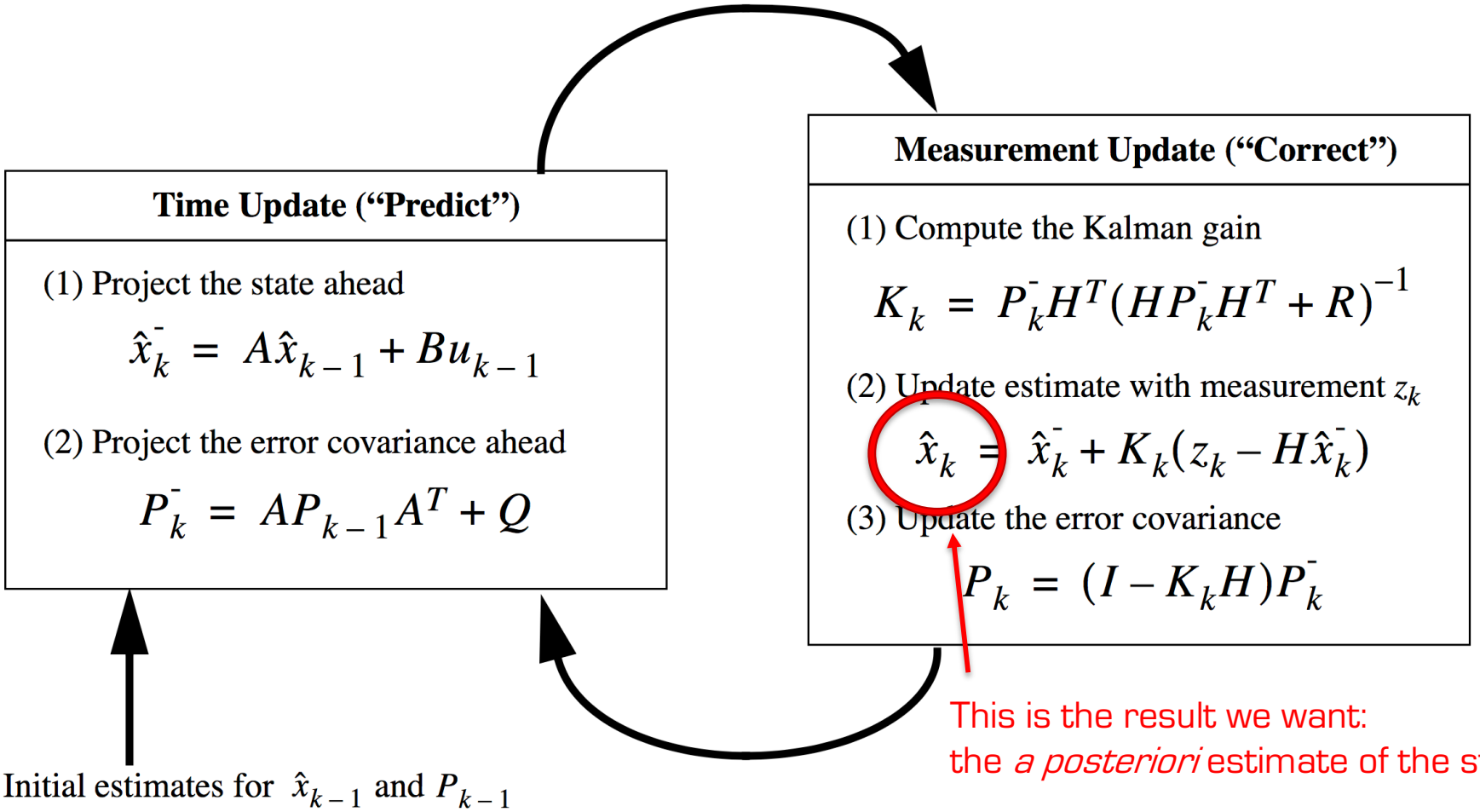
$$\hat{x}_k = \hat{x}_k^- + K_k(z_k - H\hat{x}_k^-)$$

$$P_k = (I - K_k H)P_k^-$$

Note: Trucco and Verri (1998) say that this expression can lead to serious numerical problems if both terms are small and they suggest a more complex expression that yields more stable estimates

Tracking Objects

Discrete Kalman Filter



Credit: Welch and Bishop (2006)

Tracking Objects

So how do we apply this to tracking objects?

Tracking Objects

Discrete Kalman Filter

- The state estimate provides the location of the search for the object being tracked
- The uncertainty of the state estimate (i.e. the diagonal elements of the covariance matrix) allows the size of the search region to be set automatically: higher uncertainty, larger search region
- The elements of the state covariance matrix are usually initialized to very large values (large search region) and they decrease and reach a steady state quite rapidly (in a few frames), thereby restricting the search region

Tracking Objects

The discrete Kalman filter requires the following information:

- Definition of state x_k and system model A
- System noise covariance matrix Q
- Definition of measurement z_k and measurement model H
- Measurement noise covariance matrix R
- Initial system state $\hat{x}_0$
- Initial state covariance matrix P_0
- Here we assume there is no optional control input B

Example:

State is position and velocity

$$x_k = \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \end{bmatrix}$$

$$A = \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Either measure time period between cycles or (better) embed cycle in a time thread

$$B = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Need to estimate the process noise

$$Q = \begin{bmatrix} \sigma_w^2 & 0 & 0 & 0 \\ 0 & \sigma_w^2 & 0 & 0 \\ 0 & 0 & \sigma_w^2 & 0 \\ 0 & 0 & 0 & \sigma_w^2 \end{bmatrix} = \begin{bmatrix} 0.1 & 0 & 0 & 0 \\ 0 & 0.1 & 0 & 0 \\ 0 & 0 & 0.1 & 0 \\ 0 & 0 & 0 & 0.1 \end{bmatrix}$$

$$z_k = \begin{bmatrix} x \\ y \end{bmatrix}$$

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

Need to estimate the measurement noise

$$R = \begin{bmatrix} \sigma_v^2 & 0 \\ 0 & \sigma_v^2 \end{bmatrix} = \begin{bmatrix} 0.1 & 0 \\ 0 & 0.1 \end{bmatrix}$$

Need to choose initial values of position and velocity

$$x_0 = \begin{bmatrix} x_0 \\ y_0 \\ \dot{x}_0 \\ \dot{y}_0 \end{bmatrix}$$

Use a large initial estimate of state uncertainty

$$P_0 = \begin{bmatrix} 1.0 & 0 & 0 & 0 \\ 0 & 1.0 & 0 & 0 \\ 0 & 0 & 1.0 & 0 \\ 0 & 0 & 0 & 1.0 \end{bmatrix}$$

Example:

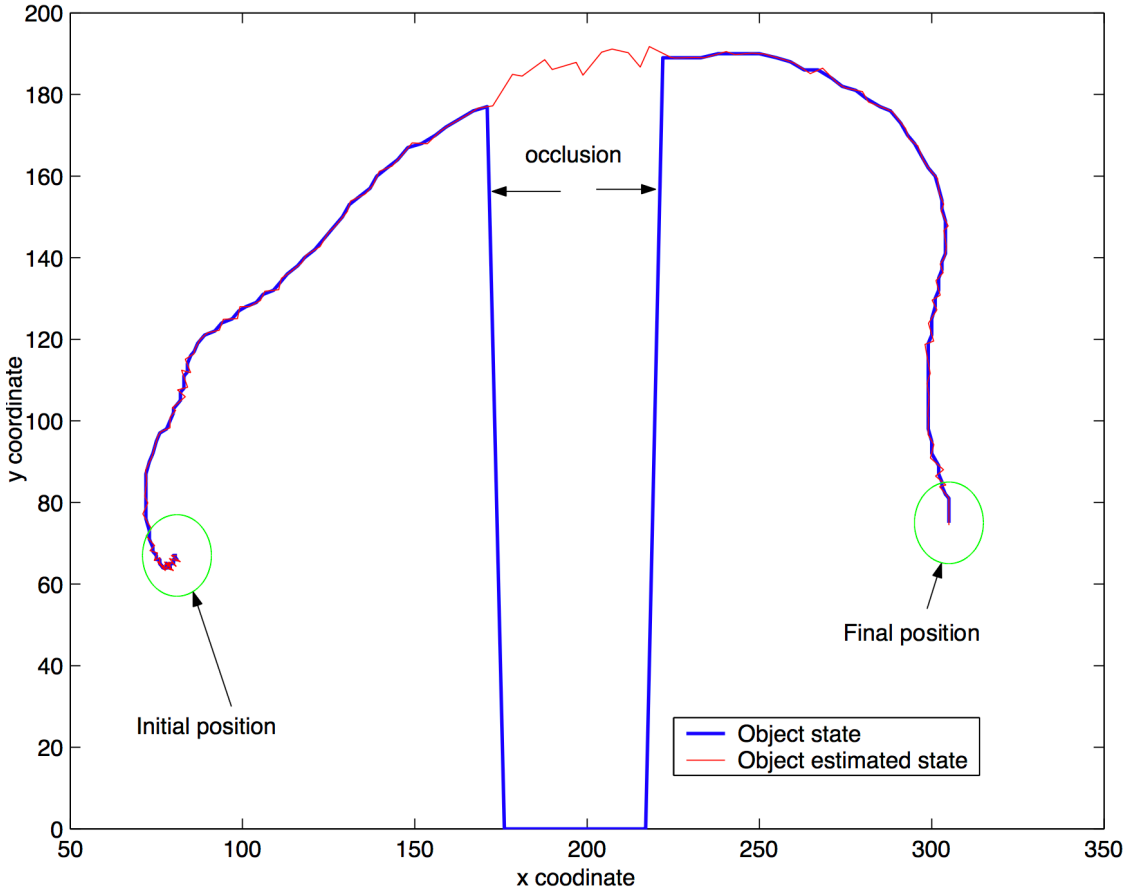
State is position and change in position

$$\mathbf{x}_{k+1} = \mathbf{F}_{k+1,k}\mathbf{x}_k + \mathbf{w}_k$$

$$\begin{bmatrix} x_{k+1} \\ y_{k+1} \\ \Delta x_{k+1} \\ \Delta y_{k+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_k \\ y_k \\ \Delta x_k \\ \Delta y_k \end{bmatrix} + \mathbf{w}_k$$

$$\mathbf{y}_k = \mathbf{H}_k\mathbf{x}_k + \mathbf{v}_k$$

$$\begin{bmatrix} xm_k \\ ym_k \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_k \\ y_k \\ \Delta x_k \\ \Delta y_k \end{bmatrix} + \mathbf{v}_k$$



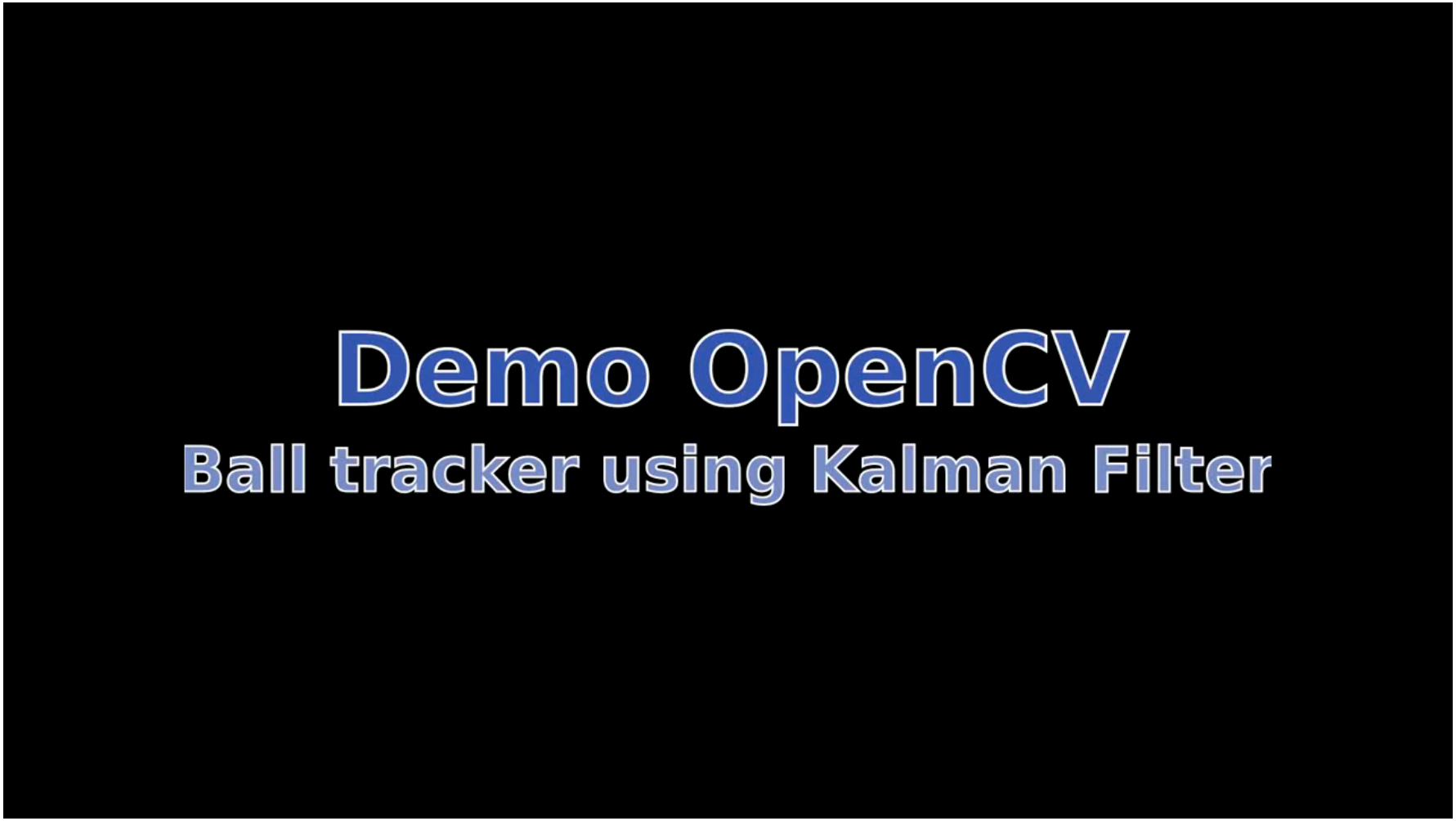
Credit: Cuevas, Zaldivar, Rojas (2005)

Tracking Objects

Discrete Kalman Filter

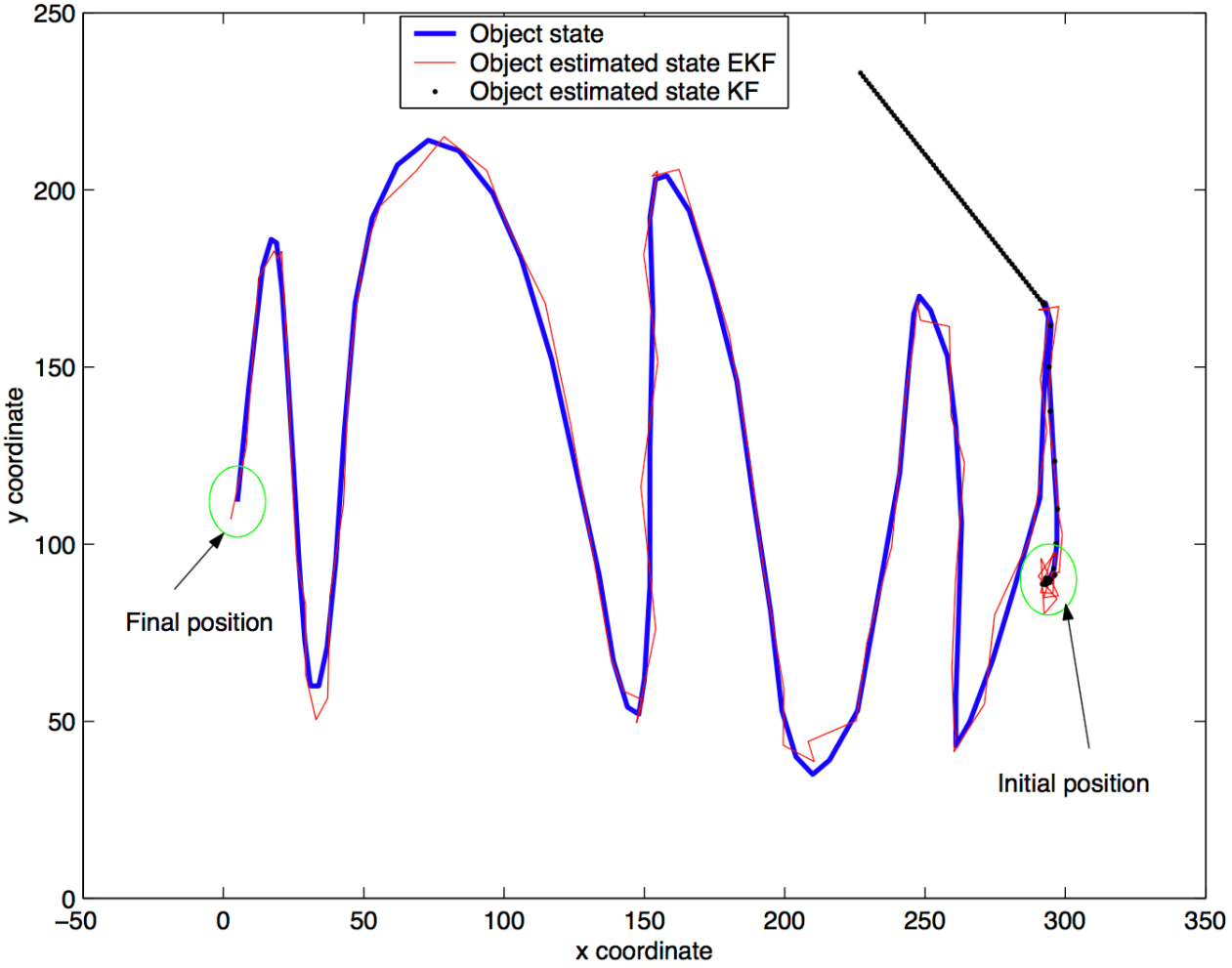
- This formulation of the state estimation problem assumes a linear model of a dynamical system
- If, however, the model is nonlinear, the Kalman filter can be adapted through a linearization procedure
- This is referred to as the extended Kalman filter (EKF)

Tracking Objects



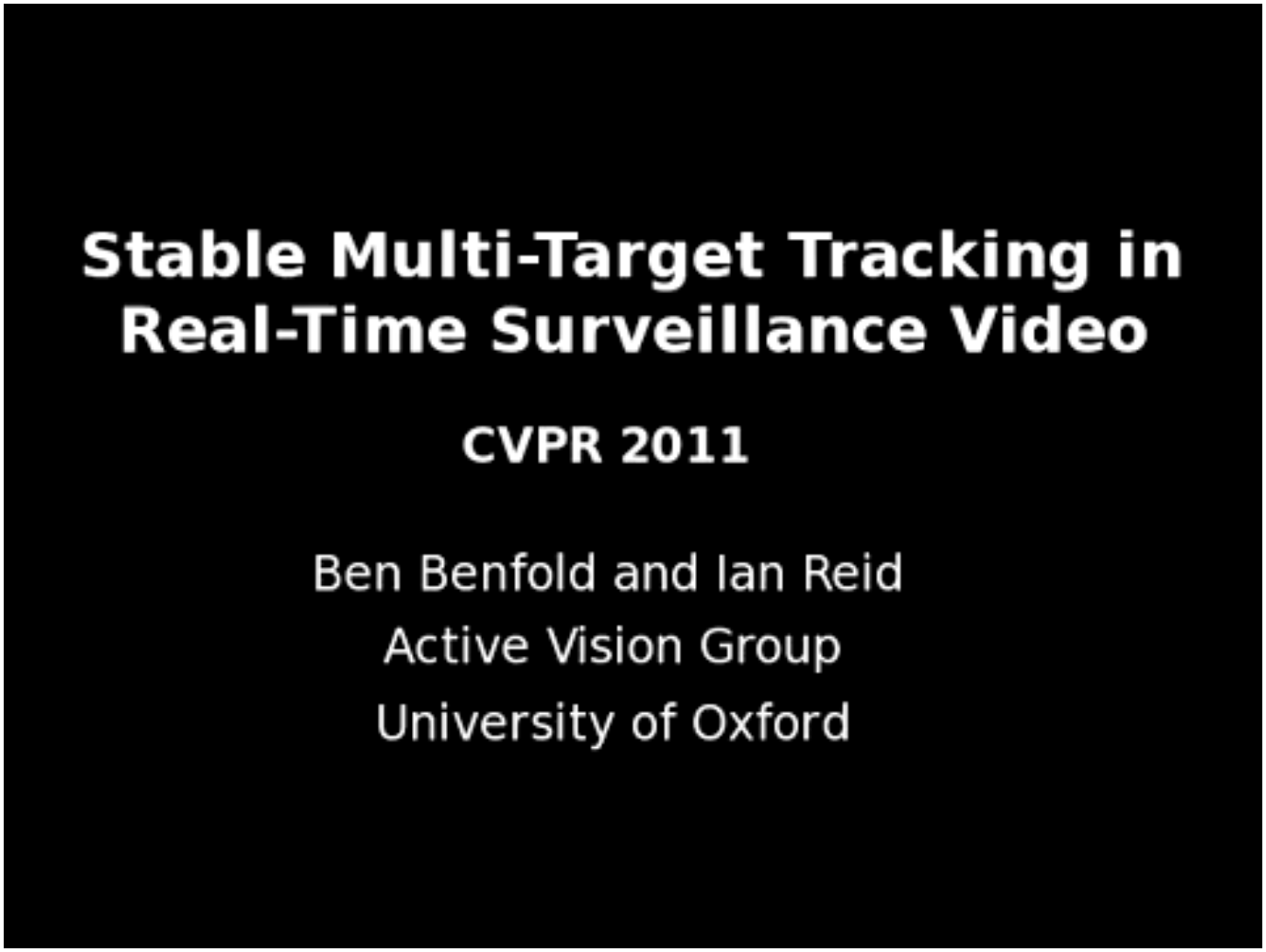
Credit: https://www.youtube.com/watch?v=sG-h5ONsj9s&feature=youtu.be

Tracking Objects



Credit: Cuevas, Zaldivar, Rojas (2005)

Tracking Objects



Credit: http://www.robots.ox.ac.uk/~lav/Papers/benfold_reid_cvpr2011/benfold_reid_cvpr2011.html
Benfold, B. and Reid, I., Stable Multi-Target Tracking in Real-Time Surveillance Video, Proc. IEEE CVPR, pp. 3457-3464, 2011.

Demos

The following code is taken from the kalmanFilterTracking project in the lectures directory of the ACV repository

See:

```
kalmanFilterTrackingTracking.h
kalmanFilterTrackingImplementation.cpp
kalmanFilterTrackingApplication.cpp
```

```
/*
  Example use of openCV to track an object using colour segmentation and the Kalman filter
  ----------------------------------------------------------------------------------------
  Implementation file

  David Vernon
  16 November 2017
*/

#include "trackingKalmanFilter.h"

/* Adapted from the example provided here http://www.robot-home.it/blog/en/software/ball-tracker-con-filtro-di-kalman/                    */
/* and code here https://github.com/Myzhar/simple-opencv-kalman-tracker/blob/8df8fb04bd06ef0be06c9b68a79fda6edf3efa1a/source/opencv-kalman.cpp */

int trackingKalmanFilter(int min_hue, int max_hue) {

    // David Vernon ... trace of detected locations
    vector<cv::Point>           detectedLocations;  // store contour of detected locations here
    vector<cv::Point>           predictedLocations; // store contour of predicted locations here
    int start;

    // Camera frame
    cv::Mat frame;
    cv::Mat res;

    // >>>> Kalman Filter
    int stateSize = 6;
    int measSize  = 4;
    int contrSize = 0;

    unsigned int type = CV_32F;
    cv::KalmanFilter kf(stateSize, measSize, contrSize, type);

    cv::Mat state(stateSize, 1, type);  // [  x,   y, v_x, v_y,  w,   h]
                                        // [  x,   y, d_x, d_y,  w,   h]  David Vernon alternative state: shift in position intead of velocity
    cv::Mat meas(measSize, 1, type);    // [z_x, z_y,          z_w, z_h]
```

```
/*
  Example use of openCV to track an object using colour segmentation and the Kalman filter
  The images are input from the camera

  Read the minimum and maximum hue values of the object to be tracked from the input file
  ----------------------------------------------------------------------------------------
  Application file

  David Vernon
  16 November 2017
*/

#include "trackingKalmanFilter.h"

int main() {

   int end_of_file;
   bool debug = true;
   int min_hue;
   int max_hue;

   FILE *fp_in;

   printf("Example use of openCV to track an object using the Kalman filter\n\n");

   if ((fp_in = fopen("../data/trackingKalmanFilterInput.txt","r")) == 0) {
     printf("Error can't open input file trackingKalmanFilterInput.txt\n");
     prompt_and_exit(1);
   }

   end_of_file = fscanf(fp_in, "%d %d", &min_hue, &max_hue);

   if (end_of_file != EOF) {
      trackingKalmanFilter(min_hue, max_hue);
   }

   fclose(fp_in);
   return 0;
}
```

```
// Transition State Matrix A
// Note: set dT at each processing step! // David Vernon - two alternatives: use dT or use shift in state instead of velocity
// [ 1 0 dT 0  0 0 ]                     //                if using dT, better to implement this function in a timed thread with constant dT
// [ 0 1 0  dT 0 0 ]
// [ 0 0 1  0  0 0 ]
// [ 0 0 0  1  0 0 ]
// [ 0 0 0  0  1 0 ]
// [ 0 0 0  0  0 1 ]
cv::setIdentity(kf.transitionMatrix);

// Measure Matrix H
// [ 1 0 0 0 0 0 ]
// [ 0 1 0 0 0 0 ]
// [ 0 0 0 0 1 0 ]
// [ 0 0 0 0 0 1 ]
kf.measurementMatrix = cv::Mat::zeros(measSize, stateSize, type);
kf.measurementMatrix.at<float>(0) = 1.0f;
kf.measurementMatrix.at<float>(7) = 1.0f;
kf.measurementMatrix.at<float>(16) = 1.0f;
kf.measurementMatrix.at<float>(23) = 1.0f;

// Process Noise Covariance Matrix Q
// [ Ex   0   0     0     0    0  ]
// [ 0    Ey  0     0     0    0  ]
// [ 0    0   Ev_x  0     0    0  ]
// [ 0    0   0     Ev_y  0    0  ]
// [ 0    0   0     0     Ew   0  ]
// [ 0    0   0     0     0    Eh ]
//cv::setIdentity(kf.processNoiseCov, cv::Scalar(1e-2));
kf.processNoiseCov.at<float>(0) = (float) 1e-2;
kf.processNoiseCov.at<float>(7) = (float) 1e-2;
kf.processNoiseCov.at<float>(14) = (float) 1e-2; //5.0f;
kf.processNoiseCov.at<float>(21) = (float) 1e-2; //5.0f;
kf.processNoiseCov.at<float>(28) = (float) 1e-2;
kf.processNoiseCov.at<float>(35) = (float) 1e-2;

// Measures Noise Covariance Matrix R
cv::setIdentity(kf.measurementNoiseCov, cv::Scalar(1e-1));
// <<<< Kalman Filter
```

```
// Camera Index
int idx = 0;

// Camera Capture
cv::VideoCapture cap;

// >>>>> Camera Settings
if (!cap.open(idx)) {
   cout << "Webcam not connected.\n" << "Please verify\n";
   //return EXIT_FAILURE;
   prompt_and_exit(1);
}

cap.set(CV_CAP_PROP_FRAME_WIDTH, 1024);
cap.set(CV_CAP_PROP_FRAME_HEIGHT, 768);
// <<<<< Camera Settings

cout << "\nHit 'q' to exit...\n";

char ch = 0;
double ticks = 0;
bool found = false;
int notFoundCount = 0;

// >>>>> Main loop
while (ch != 'q' && ch != 'Q') {

    // determine dT ... time since last cycle ... in case we need it
    double precTick = ticks;
    ticks = (double) cv::getTickCount();
    double dT = (ticks - precTick) / cv::getTickFrequency(); //seconds

    // Frame acquisition
    cap >> frame;
    frame.copyTo( res );
```

```
if (found) {
    // >>>> Matrix A
    //kf.transitionMatrix.at<float>(2) = (float) dT;
    //kf.transitionMatrix.at<float>(9) = (float) dT;
    kf.transitionMatrix.at<float>(2) = (float) 1;  // David Vernon ... use delta_x (i.e. shift) for the state instead of v_x (i.e. velocity
    kf.transitionMatrix.at<float>(9) = (float) 1;  // David Vernon ... use delta_y (i.e. shift) for the state instead of v_y (i.e. velocity
    // <<<< Matrix A

    cout << "dT:" << endl << dT << endl;

    state = kf.predict();
    cout << "State post:" << endl << state << endl;

    cv::Rect predRect;
    predRect.width = (int) state.at<float>(4);
    predRect.height = (int) state.at<float>(5);
    predRect.x = (int) state.at<float>(0) - predRect.width / 2;
    predRect.y = (int) state.at<float>(1) - predRect.height / 2;

    cv::Point center;
    center.x = (int) state.at<float>(0);
    center.y = (int) state.at<float>(1);
    cv::circle(res, center, 2, CV_RGB(255,0,0), -1);

    cv::rectangle(res, predRect, CV_RGB(255,0,0), 2);

    /* David Vernon ... draw trace of predicted points */
    predictedLocations.push_back(center);
    start = predictedLocations.size()- NUMBER_OF_POINTS_TO_DRAW; // draw the last n points
    if (start < 0)
       start = 0;
    for (int i=start; i<predictedLocations.size()-1; i++) {
       cv::line(res,predictedLocations[i], predictedLocations[i+1],CV_RGB(200,0,0));
    }
    /* ... draw trace of predicted points */
}
```

```
// >>>>> Noise smoothing
cv::Mat blur;
cv::GaussianBlur(frame, blur, cv::Size(5, 5), 3.0, 3.0);
// <<<<< Noise smoothing

// >>>>> HSV conversion
cv::Mat frmHsv;
cv::cvtColor(blur, frmHsv, CV_BGR2HSV);
// <<<<< HSV conversion

// >>>>> Color Thresholding
// Note: change parameters for different colors
cv::Mat rangeRes = cv::Mat::zeros(frame.size(), CV_8UC1);

//cv::inRange(frmHsv, cv::Scalar(MIN_H_BLUE / 2, 100, 80),
//            cv::Scalar(MAX_H_BLUE / 2, 255, 255), rangeRes);
cv::inRange(frmHsv, cv::Scalar(min_hue / 2, 100, 80),          // David Vernon: use parameter values for hue instead of hard-coded values
            cv::Scalar(max_hue / 2, 255, 255), rangeRes);
// <<<<< Color Thresholding

// >>>>> Improving the result
cv::erode(rangeRes, rangeRes, cv::Mat(), cv::Point(-1, -1), 2);
cv::dilate(rangeRes, rangeRes, cv::Mat(), cv::Point(-1, -1), 2);
// <<<<< Improving the result

// Thresholding viewing
cv::imshow("Threshold", rangeRes);

// >>>>> Contours detection
vector<vector<cv::Point> > contours;
cv::findContours(rangeRes, contours, CV_RETR_EXTERNAL,
                 CV_CHAIN_APPROX_NONE);
// <<<<< Contours detection
```

```
// >>>>> Filtering
vector<vector<cv::Point> > balls;
vector<cv::Rect> ballsBox;
for (size_t i = 0; i < contours.size(); i++) {
    cv::Rect bBox;
    bBox = cv::boundingRect(contours[i]);

    /* track balls ... with appox. square bounding box ... this is the original application */
    /*
    float ratio = (float) bBox.width / (float) bBox.height;
    if (ratio > 1.0f)
        ratio = 1.0f / ratio;

    // Searching for a bBox almost square
    if (ratio > 0.75 && bBox.area() >= 400)
    {
        balls.push_back(contours[i]);
        ballsBox.push_back(bBox);
    }
    */

    /* David Vernon: track any shape */
    balls.push_back(contours[i]);
    ballsBox.push_back(bBox);
}
// <<<<< Filtering

cout << "Balls found:" << ballsBox.size() << endl;
```

```
// >>>>> Detection result
for (size_t i = 0; i < balls.size(); i++)  {
    cv::drawContours(res, balls, i, CV_RGB(20,150,20), 1);
    cv::rectangle(res, ballsBox[i], CV_RGB(0,255,0), 2);

    cv::Point center;
    center.x = ballsBox[i].x + ballsBox[i].width / 2;
    center.y = ballsBox[i].y + ballsBox[i].height / 2;
    cv::circle(res, center, 2, CV_RGB(20,150,20), -1);

    stringstream sstr;
    sstr << "(" << center.x << "," << center.y << ")";
    cv::putText(res, sstr.str(),
                cv::Point(center.x + 3, center.y - 3),
                cv::FONT_HERSHEY_SIMPLEX, 0.5, CV_RGB(20,150,20), 2);

    /* David Vernon ... draw trace of detected points */
    detectedLocations.push_back(center);
    start = detectedLocations.size()- NUMBER_OF_POINTS_TO_DRAW; // draw the last n points
    if (start < 0)
       start = 0;
    for (int i=start; i<detectedLocations.size()-1; i++) {
       cv::line(res,detectedLocations[i], detectedLocations[i+1],CV_RGB(0,200,0));
    }
    /* ... draw trace of detected points */
}
// <<<<< Detection result
```

```
// >>>>> Kalman Update
if (balls.size() == 0) {
    notFoundCount++;
    cout << "notFoundCount:" << notFoundCount << endl;
    if( notFoundCount >= 100 ) {
        found = false;
    }
    detectedLocations.clear(); // David Vernon - lost ball so reset the contour of detected locations
}
else {
    notFoundCount = 0;
    meas.at<float>(0) = (float) (ballsBox[0].x + ballsBox[0].width / 2);
    meas.at<float>(1) = (float) (ballsBox[0].y + ballsBox[0].height / 2);
    meas.at<float>(2) = (float)ballsBox[0].width;
    meas.at<float>(3) = (float)ballsBox[0].height;

    if (!found) { // First detection!
        // >>>> Initialization
        kf.errorCovPre.at<float>(0)  = (float) 1; // px
        kf.errorCovPre.at<float>(7)  = (float) 1; // px
        kf.errorCovPre.at<float>(14) = (float) 1;
        kf.errorCovPre.at<float>(21) = (float) 1;
        kf.errorCovPre.at<float>(28) = (float) 1; // px
        kf.errorCovPre.at<float>(35) = (float) 1; // px
        state.at<float>(0)           = meas.at<float>(0);
        state.at<float>(1)           = meas.at<float>(1);
        state.at<float>(2)           = (float) 0;
        state.at<float>(3)           = (float) 0;
        state.at<float>(4)           = meas.at<float>(2);
        state.at<float>(5)           = meas.at<float>(3);
        // <<<< Initialization
        kf.statePost = state;   // Kalman filter state
        found = true;
    }
    else {
        kf.correct(meas);       // Kalman Correction
    }
    cout << "Measure matrix:" << endl << meas << endl;
}
```

```
        // <<<<< Kalman Update

        // Final result
        cv::imshow("Tracking", res);

        // User key
        ch = cv::waitKey(1);
    }
    // <<<<< Main loop
    return EXIT_SUCCESS;
}
```

Reading

G. Welch and G. Bishop, An Introduction to the Kalman Filter, TR 95-042, Department of Computer Science, University of North Carolina at Chapel Hill, 2006.

E. Trucco and A. Verri, Introductory Techniques for 3-D Computer Vision, Prentice Hall, 1998.

Section 8.4.2 Feature-based Techniques

E. Cuevas, D. Zaldivar, and P. Rojas, Kalman filter for vision tracking, TR B 05-12, Freie Universität Berlin, 2005