Navigation with the SCAAT method

1. The SCAAT method

The SCAAT (single constraint at a time) method is described by Greg Welch in his paper http://www.cs.unc.edu/~welch/media/pdf/scaat.pdf.
It uses a Extended Kalman Filter (EKF) to provide an estimate of the position and velocity of a moving object based on sequential single distance measurements to stationary beacons with known positions.

2. Process model

We will use a simple linear position-velocity model:

$$x_{(t+\delta t)} = A_{(\delta t)}\, x_{(t)} + w_{(\delta t)} \tag{1}$$

where the *state* of the system is represented by the 6-element vector:

$$x_t = [\, x, y, z, x', y', z' \,]^T \tag{2}$$

The *n x n state transition matrix* A in (1) projects the state forward from time *t* to time *t* + *δt*. In our model, the matrix implements the relationship:

$$\begin{aligned} x_{(t+\delta t)} &= x_{(t)} + x'_{(t)}\, \delta t \\ x'_{(t+\delta t)} &= x'_{(t)} \end{aligned} \tag{3}$$

and likewise for the remaining elements of (2).

The *n x 1 process noise vector* $w_{(\delta t)}$ in (1) is a normally distributed zero-mean sequence that represents the uncertainty in the target state over the time interval *δt*. The *n x n process noise covariance matrix* Q is given by:

$$Q_{(\delta t)[i,j]} = \begin{array}{l} Q_{(\delta t)[i,i]} = \eta\, (\delta t)^3 / 3 \\ Q_{(\delta t)[j,i]} = \eta\, (\delta t)^2 / 2 \\ Q_{(\delta t)[j,j]} = \eta\, (\delta t) \end{array} \tag{4}$$

for each pair (*i*, *j*) ε { (x, x'), (y, y'), (z, z') }.

The η in (4) are the *correlation kernels* of the (assumed constant) noise sources presumed to be driving the dynamic model.

3. Measurement model

The 3-element *measurement vector* is composed of the measured distance to a beacon B with known coordinates:

$$z_t = [\, d, B_x, B_y, B_z \,]^T \tag{5}$$

We also define a *measurement function h* such that

$$z_t = h(x_{(t)}, B_x, B_y, B_z) + v_{(t)} \tag{6}$$

In our navigation system, the measurement function *h* calculates de distance to a beacon at (B_x, B_y, B_z):

$$h(x_{(t)}, B_x, B_y, B_z) = \sqrt{((x_1 - B_x)^2 + (x_2 - B_y)^2 + (x_3 - B_z)^2)} \quad (7)$$

The *measurement noise* $v_{(t)}$ in (6) is a normally distributed zero-mean scalar which represents any random error in the measurement.
The corresponding *measurement noise variance* $R_{(t)}$ is assumed constant.

4. Measurement Jacobian

Because we have a non-linear measurement model, we are using the *Extended Kalman Filter*. For the EKF we need the *Jacobian* of the measurement function (7), which is the derivative of the measurement function with respect to the state:

$$H = [\ \partial h/\partial x, \partial h/\partial y, \partial h/\partial z, 0, 0, 0\ ] \quad (8)$$

where:

$$\partial h/\partial x = (x_1 - B_x) / \sqrt{((x_1 - B_x)^2 + (x_2 - B_y)^2 + (x_3 - B_z)^2)} \quad (9)$$
$$\partial h/\partial y = (x_2 - B_y) / \sqrt{((x_1 - B_x)^2 + (x_2 - B_y)^2 + (x_3 - B_z)^2)} \quad (10)$$
$$\partial h/\partial z = (x_3 - B_z) / \sqrt{((x_1 - B_x)^2 + (x_2 - B_y)^2 + (x_3 - B_z)^2)} \quad (11)$$

5. Tracking algorithm

Given an initial estimate $x_{(0)}$ and error covariance estimate $P_{(0)}$, the EKF cycles through the following steps whenever a beacon distance measurement becomes available:

a. Compute the time interval δt since the previous estimate.

b. Predict the state and error covariance matrix:
$$x^- = A_{(\delta t)}\ x_{(t - \delta t)} \quad (12)$$
$$P^- = A_{(\delta t)}\ P_{(t - \delta t)}\ A^T_{(\delta t)} + Q_{(\delta t)} \quad (13)$$

c. Predict the measurement and compute the Jacobian:
$$z^- = h(x^-, B_x, B_y, B_z) \quad (14)$$
$$H = H(x^-, B_x, B_y, B_z) \quad (15)$$

d. Compute the *residual* between de actual distance measurement and the predicted measurement, and the measurement covariance:
$$e = d_{(t)} - z^- \quad (16)$$
$$R_e = H\ P^-\ H^T + R_{(t)} \quad (17)$$

e. Compute the Kalman gain:
$$K = P^-\ H^T\ R_e^{-1} \quad (18)$$

f. Correct the predicted state estimate and error covariance:
$$x_{(t)} = x^- + Ke \quad (19)$$
$$P_{(t)} = (I - KH)\ P^- \quad (20)$$

6. Implementation

Equations (12) to (20) are implemented in Processing which uses a simplifed Java syntax. The JAMA library is used for matrix operations.

```
// Extended Kalman Filter class for SCAAT navigation

import Jama.Matrix;

class EKF
{
  Matrix x;  // n x 1 state vector
  Matrix A;  // n x n state transition matrix
  Matrix P;  // n x n error covariance matrix
  Matrix Q;  // n x n process noise covariance matrix
  double eta = 0.1;// process noise variance
  double R = 0.1;  // measurement noise variance
  double e;  // residual

  public EKF()
  {
    // init the state vector
    x = new Matrix(6, 1);
    // init the covariance matrix
    P = Matrix.identity(6, 6);
    // init the state transition matrix
    A = Matrix.identity(6, 6);
    // init the process noise covariance matrix
    Q = new Matrix(6, 6);
  }

  public void predict(float dt)
  {
    // predict the [n x 1] state
    A.set(0, 3, dt);
    A.set(1, 4, dt);
    A.set(2, 5, dt);
    x = A.times(x);

    // predict the [n x n] covariance matrix
    double f1 = eta * dt;
    double f2 = f1 * dt;
    double f3 = f2 * dt;
    f2 /= 2;
    f3 /= 3;
    Q.set(0, 0, f3);
    Q.set(1, 1, f3);
    Q.set(2, 2, f3);
    Q.set(3, 3, f1);
    Q.set(4, 4, f1);
    Q.set(5, 5, f1);
    Q.set(0, 3, f2);
    Q.set(1, 4, f2);
    Q.set(2, 5, f2);
    Q.set(3, 0, f2);
    Q.set(4, 1, f2);
    Q.set(5, 2, f2);
    P = A.times(P).times(A.transpose()).plus(Q);
  }
```

```
  public void innovate(double d, double bx, double by, double bz)
  {
    // predict the measurement
    double x1 = x.get(0, 0) - bx;
    double x2 = x.get(1, 0) - by;
    double x3 = x.get(2, 0) - bz;
    double z = Math.sqrt(x1*x1 + x2*x2 + x3*x3);

    // compute the [1 x n] jacobian
    Matrix H = new Matrix(1, 6);
    H.set(0, 0, x1 / z);
    H.set(0, 1, x2 / z);
    H.set(0, 2, x3 / z);

    // compute the (scalar) innovation residual
    e = d - z;

    // compute the (scalar) innovation variance
    double Re = H.times(P).times(H.transpose()).get(0, 0) + R;

    // compute the [n x 1] kalman gain
    Matrix K = P.times(H.transpose()).times(1.0 / Re);

    // correct the predicted state and covariance matrix
    x = x.plus(K.times(e));
    P = Matrix.identity(6, 6).minus(K.times(H)).times(P);
  }
};
```