Bayesian Estimation and the Kalman Filter

Allen L. Barker
Donald E. Brown
Worthy N. Martin

IPC-TR-94-002
July 15, 1994
(Revised Sept. 19, 1994)

Institute for Parallel Computation
School of Engineering and Applied Science
University of Virginia
Charlottesville, VA 22901

This research was sponsored in part by the Jet Propulsion Laboratory
under grant number 95772.

Abstract

In this tutorial article we give a Bayesian derivation of a basic state estimation result for discrete-time Markov process models with independent process and measurement noise and measurements not affecting the state. We then list some properties of Gaussian random vectors and show how the Kalman filtering algorithm follows from the general state estimation result and a linear-Gaussian model definition. We give some illustrative examples including a probabilistic Turing machine, dynamic classification, and tracking a moving object.

1 Introduction

The goal of this paper is to provide a relatively self-contained derivation of some Bayesian estimation results leading to the Kalman filter, with emphasis on conceptual simplicity. The results we present are really just a repackaging of standard results in optimal estimation theory and Bayesian analysis, following mainly from references [Med69, JH69, Sal89, Ber85]. We hope, though, that this paper will provide useful results which can be put to immediate practical use. We adopt a Bayesian approach because it lends itself to a straightforward, intuitive derivation.

The usual Bayesian derivation proceeds by first generating a posterior density from the prior density and current measurement, and then updating this posterior density to be the prior density for the next time step. This process is then repeated sequentially for all measurements. In this paper we consider the problem as a batch estimation problem, where we are given all the data at once. From this batch estimate the recursive algorithm follows from the ordering of the computations by which the mathematical expression is evaluated. We also encapsulate some of the algebraic manipulations into a theorem on multiplying Gaussian densities.

We have tried to write out enough steps in the derivations that each equation follows easily from the previous ones. Some results are stated without proof, though, and we have sacrificed some formality and generality for the sake of clarity. In Section 2 we formally define the problem for general densities. In Section 3 we derive an expression for the desired solution in terms of the known densities. In Section 4 we give some theorems on Gaussian random vectors and densities. In Section 5 we give a linear, Gaussian model and use the results in Sections 3 and 4 to derive the Kalman filtering algorithm, which efficiently solves the problem in this case. In Section 6 we give some examples. The general progression is from abstract to more concrete; some readers may wish to skim the first few sections on a first reading and concentrate on the examples, particularly Example 3. The notes at the end of each section provide additional information but are not needed to follow the main text except as indicated.

2 The Problem

First a bit of notation. We will write $x_k \equiv x(t_k)$, that is, the discrete subscript k indexes a real-valued variable t_k which is the argument to x.

These real-valued variables can take on any values and in particular need not be evenly spaced. We assume the association is ordered so that $t_i < t_j$ iff $i < j$. We refer to these variables as time instances, though in many applications these variables do not refer to time. We write the tilde symbol above random variables, and take the variable name without the tilde to refer to a member of the random variable's range. Thus $\tilde{x}$ is a vector random variable, an observation (or value, or mathematically a realization) of which may be x. We write $p(x)$ for the density function of random vector $\tilde{x}$, and likewise $p(x|y)$ for this density conditioned on $\tilde{y} = y$. We assume the density value associated with any observed event is nonzero. We generally allow random vectors to contain both continuous and discrete random variables as elements.

We consider system models of the class

$$\tilde{x}_{k+1} = f(\tilde{x}_k, \tilde{\pi}_k, t_k, t_{k+1}) \quad (1)$$

$$\tilde{z}_{k+1} = g(\tilde{x}_{k+1}, \tilde{\phi}_{k+1}), \quad (2)$$

where the density $p(x_0)$ of random vector $\tilde{x}_0$ and the densities for all members of the vector random variable families $\tilde{\pi}_k$ and $\tilde{\phi}_k$ are assumed to be known a priori. All of these random variables with known densities are assumed mutually statistically independent. We also assume that from these equations the densities $p(x_{k+1}|x_k)$ and $p(z_{k+1}|x_{k+1})$ can be computed. We are given the set $Z = \{(z_1, t_1), \ldots, (z_n, t_n)\}$ of observed values for the random variables $\tilde{z}_k$ at n known time points. The problem is to determine, for any given future time instant t_q with $q \geq n$, the posterior density $p(x_q|Z)$ for state $\tilde{x}_q$ given the observed data Z. The problem is illustrated in Figure 1.

Notes:

1. We assume density functions, possibly containing Dirac delta functions, are defined for all probability distributions we deal with.

2. The random variable families $\tilde{\pi}$ and $\tilde{\phi}$ in (1) and (2) are *discrete-time independent stochastic processes.* The $\tilde{x}_k$ form a *discrete time Markov process* because we have the property $p(x_k|x_0, \ldots, x_{k-1}) = p(x_k|x_{k-1})$. This property holds for *any* past state conditioning: we can eliminate conditioning on all but the most recent state. It can be shown that a Markov process is also reverse-time Markov: given the state conditioned on any future states we can eliminate the conditioning

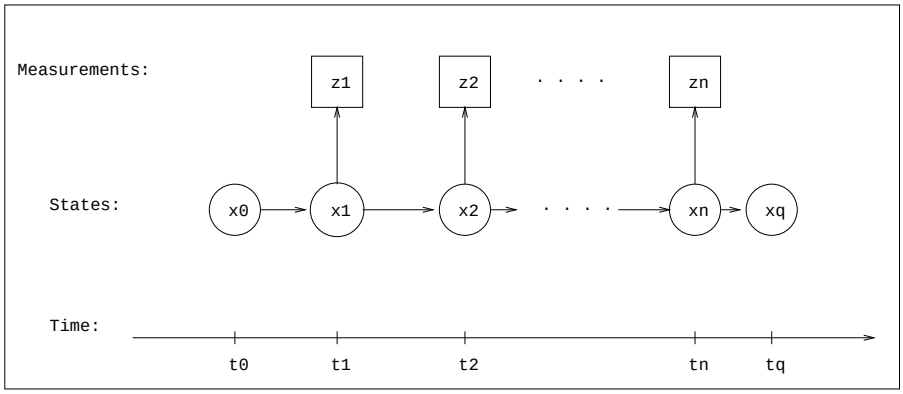

Figure 1: Given the measured data Z and the density for initial state x_0, calculate the density for final state x_q.

on all but the closest future state in time. When all elements of the state vector $\tilde{x}_k$ are discrete random variables the term *Markov chain* is usually used instead of Markov process.

3. Equation (1) is a *motion model*, or *state transition equation*, and is a description, here a stochastic (or probabilistic) one, of the motion of the state vector with time. Equation (2) is a *sensor model*, or *measurement model*, and is a description, here also stochastic, of the information sensors return about the state.

4. Note the generality of the concept of "states" x_k in (1). A dependence on any finite number of previous states can be reduced to Markov dependence using the trick of augmenting the state vector and "copying states forward" in the state update (1). This is analogous to reducing a high order differential equation to a system of first order equations.

5. The time values in the arguments to f, equation (1), are known values, and we could generally include vectors of any known values in the arguments to f and g.

6. In our derivations, we really only need to know the densities $p(x_{k+1}|x_k)$ and $p(z_{k+1}|x_{k+1})$ rather than (1) and (2) and the random variables involved. Equations (1) and (2) are useful in modeling physical situations; in some cases, though, it may be simpler to just define these conditional densities as the model.

7. This problem, with $t_q \geq t_n$, is a *filtering* problem. The problem with $t_q < t_n$ is a *smoothing* problem, which can be handled similarly to the filtering problem.

8. Equation (1) is an *iterated function system*; such systems have received much recent study in relation to nonlinear dynamics and chaos theory. See [Ber92, CY92] for reviews of such systems from a statistical viewpoint.

3 Calculating the Posterior Density

In this section we derive an expression for the desired posterior density $p(x_q|Z)$ in terms of the known density functions. We assume the reader is familiar with multivariate joint and conditional densities, and with relations such as Bayes' rule and $p(x,y) = p(x|y)p(y)$ for random vectors x and y.

The first step we take in deriving our expression for $p(x_q|Z)$ is to write $p(x_q|Z)$ in terms of the joint density $p(x_0,\ldots,x_n,x_q|Z)$. By definition $p(x_q|Z)$ is just the marginal

$$p(x_q|Z) = \int dX\ p(x_0,\ldots,x_n,x_q|Z), \tag{3}$$

where $\int dX$ is taken to mean the multiple integral over all space of the variables $x_0,\ldots,x_n$. If any elements in mixed random vector x_i are discrete we may take the integral as a shorthand for summation with respect to discrete elements and integration with respect to continuous ones. Alternately, we could consider discrete densities as mixtures of delta functions or write the integrals as Stieltjes integrals.

Applying Bayes' rule to (3), we get

$$p(x_q|Z) = \int dX\ p(Z|x_0,\ldots,x_n,x_q)p(x_0,\ldots,x_n,x_q)/p(Z), \tag{4}$$

where

$$p(Z) = \int dx_q \int dX\ p(Z|x_0,\ldots,x_n,x_q)p(x_0,\ldots,x_n,x_q) \tag{5}$$

is the normalizing constant. Now, from equation (2) and the independence of the $\tilde{\phi}_k$ we can write

$$p(x_q|Z) = c^{-1} \int dX\ p(z_1|x_1)\ldots p(z_n|x_n)p(x_0,\ldots,x_n,x_q), \tag{6}$$

where we have set $c = p(Z)$.

Since we assume $q > n$, we can break down the joint density $p(x_0, \ldots, x_n, x_q)$ as

$$\begin{aligned} p(x_0, \ldots, x_n, x_q) &= p(x_q|x_0, \ldots, x_n)p(x_0, \ldots, x_n) \\ &= p(x_q|x_n)p(x_0, \ldots, x_{n-1}, x_n) \\ &= p(x_q|x_n)p(x_n|x_0, \ldots, x_{n-1})p(x_0, \ldots, x_{n-1}) \\ &= p(x_q|x_n)p(x_n|x_{n-1})p(x_0, \ldots, x_{n-1}) \\ &\vdots \\ &= p(x_q|x_n)\left[\prod_{i=1}^{n} p(x_i|x_{i-1})\right] p(x_0), \end{aligned}$$

where we have used the Markov property (see note 2, section 1) to eliminate conditioning and repeatedly applied Bayes' rule to "unroll" $p(x_0, \ldots, x_n, x_q)$. Plugging in we get

$$p(x_q|Z) = c^{-1} \int dX \; p(z_1|x_1) \ldots p(z_n|x_n)p(x_q|x_n) \left[\prod_{i=1}^{n} p(x_i|x_{i-1})\right] p(x_0). \tag{7}$$

Finally, on rearranging terms, we obtain the result

$$\begin{aligned} p(x_q|Z) &= c^{-1} \int dX \; p(x_0) \left[p(x_1|x_0)p(z_1|x_1)\right] \left[p(x_2|x_1)p(z_2|x_2)\right] \ldots \\ &\qquad \ldots \left[p(x_{n-1}|x_{n-2})p(z_{n-1}|x_{n-1})\right] \left[p(x_n|x_{n-1})p(z_n|x_n)\right] p(x_q|x_n) \\ p(x_q|Z) &= c^{-1} \int dX \; p(x_0) \left[\prod_{i=1}^{n} p(x_i|x_{i-1})p(z_i|x_i)\right] p(x_q|x_n). \end{aligned} \tag{8}$$

The game now is to *evaluate* this expression. Notice that the terms dependent on any x_i appear in sequence, with at most 3 terms in the sequence. These sequences are strictly increasing in time "left to right" in the equation.

Whether expression (8) can be efficiently evaluated depends strongly on the form of the densities involved. We would like to find a sequence of evaluation for the integrals such that the result after each step leaves an expression which can then be efficiently evaluated in the next step, and so on. In real-time problems an ordering which follows the time ordering of the states is also desirable.

Let us consider equation (8) in the case where we take $q = n + 1$. In this case we can rewrite (8) as

$$p(x_{n+1}|Z_n) = \tag{9}$$
$$c^{-1} \int dx_n \int dX_{(n-1,0)} \left[p(x_0) \left[\prod_{i=1}^{n-1} p(x_i|x_{i-1}) p(z_i|x_i) \right] p(x_n|x_{n-1}) \right.$$
$$\left. \cdot p(z_n|x_n) p(x_{n+1}|x_n) \right].$$

where Z_n indicates the data set up to (z_n, t_n) and $dX_{(n-1,0)}$ indicates the integral with respect to $x_0, \ldots, x_{n-1}$. When written in this form we can see that the recursive equation

$$p(x_{k+1}|Z_k) = c_k^{-1} \int dx_k \; \underline{p(x_k|Z_{k-1})} p(z_k|x_k) p(x_{k+1}|x_k). \tag{10}$$

holds by plugging in the underlined expression. The underlined density is a recursive "function call" and the other densities in the r.h.s. were assumed to be known a priori. Using recursive relations like (10) one can efficiently update a previous estimate whenever new data is received, without recomputing everything. This is especially important if the integrations are performed numerically. Note that the way (10) is written the recursion only goes down to $p(x_2|Z_1) \equiv p(x_2|z_1)$, since Z_0 is undefined. In this case it is more convenient to define the recursion in terms of a pair of mutually recursive equations for $p(x_k|Z_{k-1})$ and $p(x_k|Z_k)$ (see note 2).

Notes:

1. See [MCTW86] for a measure-theoretic treatment of mixed continuous and discrete random vectors.

2. It is often convenient to evaluate (8) using the time-ordered pair of mutually recursive equations given by

$$\begin{aligned} p(x_{k+1}|Z_k) &= \int dx_k \; p(x_{k+1}, x_k|Z_k) \\ &= \int dx_k \; p(x_{k+1}|x_k, Z_k) p(x_k|Z_k) \\ &= \int dx_k \; p(x_{k+1}|x_k) \underline{p(x_k|Z_k)} \end{aligned}$$

and

$$\begin{aligned} p(x_k|Z_k) &= p(x_k|Z_{k-1}, z_k) \\ &\propto p(z_k|x_k, Z_{k-1}) p(x_k|Z_{k-1}) \\ &= p(z_k|x_k) \underline{p(x_k|Z_{k-1})}, \end{aligned}$$

where we define $Z_0 = \emptyset$ and $p(x_k|\emptyset) = p(x_k)$. The recursive "function calls" are underlined. The first equation can be considered a *prediction* of a future state, and the second a *measurement update* or a *correction* of the prediction when given a new observation or sensor report. We could alternately have started with these equations and used them to derive equation (8).

3. When we "unrolled" the joint density we pulled out the x_i in order of decreasing time so we could use the Markov property to eliminate dependencies. Using the backward transition densities $p(x_k|x_{k+1})$ we can pull out the x_i in increasing order using the reverse-time Markov property. In fact, we can pull out variables in arbitrary order and eliminate dependencies on all but the two nearest states in time, above and below. When given $p(x_0)$, though, we should pull out x_0 last.

4 Some Theorems on Gaussian Random Vectors and Densities

Before considering the linear-Gaussian model we first present some theorems related to Gaussian random vectors. Then in the following section we apply these theorems to evaluate the integrals in equation (8) for the linear-Gaussian model. From this point on, until the examples, we consider only continuous random variables. First we make some definitions. Let

a be an $r \times 1$ matrix (column vector),
A be an $r \times r$ symmetric, positive definite matrix,
b be an $s \times 1$ matrix (column vector),
B be an $s \times s$ symmetric, positive definite matrix,
Q be an $r \times s$ matrix,
x be an $s \times 1$ matrix (column vector),
$\tilde{x}$ be an $s \times 1$ random matrix (column vector).

Define

$$C \equiv C(Q, A, B) = (Q'A^{-1}Q + B^{-1})^{-1} \quad (11)$$
$$= B - BQ'(A + QBQ')^{-1}QB, \quad (12)$$

and

$$c \equiv c(Q,a,A,b,B) = C[Q'A^{-1}a + B^{-1}b] \quad (13)$$
$$= b + CQ'A^{-1}(a - Qb). \quad (14)$$

Also define the r-dimensional Mahalanobis distance M_r as the quadratic form

$$M_r(a,A,x) = M_r(x,A,a) = (1/2)(x-a)'A^{-1}(x-a), \quad (15)$$

and the Gaussian density function by

$$J_r(A) = (2\pi)^{-r/2}\det(A)^{-1/2} \quad (16)$$
$$G_r(a,A,x) = G_r(x,A,a) = J_r(A)\, e^{-M_r(a,A,x)}. \quad (17)$$

Here $G_r(a,A,x)$ is the r-dimensional Gaussian (i.e. normal) density function with mean a and covariance matrix A.

Using the above definitions we first give a theorem on adding Mahalanobis distances.

Theorem 1 *Let variables a, A, r, b, B, s, c, C, and Q be defined as above. Then*

$$M_r(a,A,Qx) + M_s(b,B,x) = M_s(c,C,x) + M_r(a,A+QBQ',Qb).$$

This theorem was taken from [Sal89], Appendix A, and a proof can be found there. The proof is straightforward, though somewhat tedious, and involves completing the square and applying the matrix inversion lemma (see note 1).

The following theorem, illustrating one of the amazing reproducing properties of the Gaussian density, can be easily proven using Theorem 1 along with the relation

$$\det(A)\det(B)\det(Q'A^{-1}Q + B^{-1}) = \det(QBQ' + A).$$

Theorem 2 *Let variables a, A, r, b, B, s, c, C, and Q be defined as above. Then*

$$G_r(a,A,Qx)G_s(b,B,x) = G_s(c,C,x)G_r(a,A+QBQ',Qb).$$

Note that Theorem 2 can be used to shift the dependence on x from a pair of Gaussians to a single Gaussian. The theorem is illustrated in Figures 2 and 3 for an $r = s = 2$ dimensional case with $Q = I$. The two Gaussians

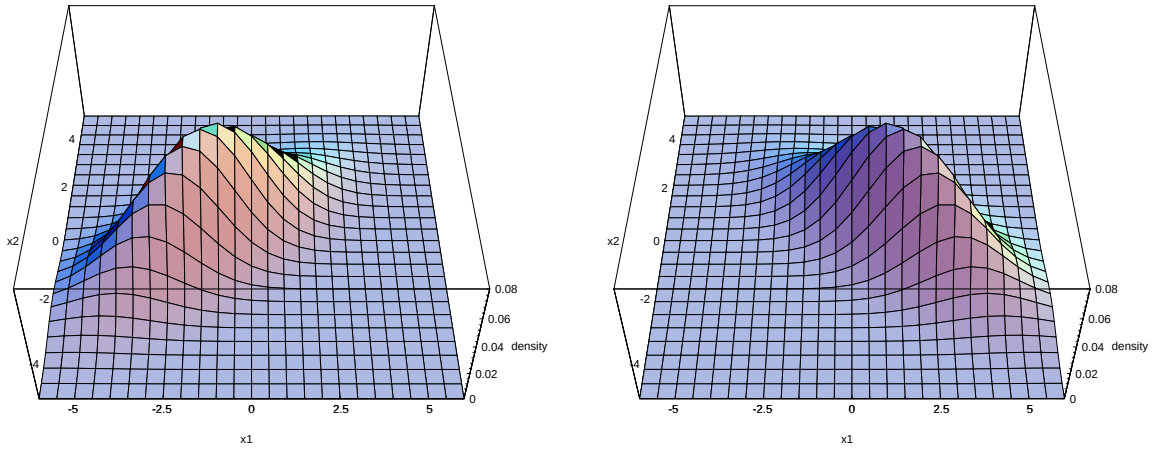

Figure 2: Two Gaussian densities in $x = (x1, x2)'$ space to be multiplied.

shown in Figure 2 are the Gaussians in the l.h.s of Theorem 2. Note that they are both Gaussians in variable $x = (x1, x2)'$. The Gaussian in Figure 3 is their product, which is again Gaussian in x by the r.h.s. of Theorem 2. The Gaussian in Figure 3 is not normalized; from Theorem 2 we know its normalizing constant is the reciprocal of another Gaussian independent of x.

We will also need the following result, that linear transformations of Gaussian random vectors are Gaussian random vectors. See e.g. [JW88] for a proof.

Theorem 3 *Let $\tilde{x}$, x, b, B and Q be defined as above. Let d be an $s \times 1$ matrix of constants. Let $\tilde{x}$ have density $p(x) = G_s(b, B, x)$. Then random vector $\tilde{w} = \tilde{x} + d$ has density*

$$p(w) = G_s(b + d, B, w) = G_s(b, B, w - d),$$

and random vector $\tilde{y} = Q\tilde{x}$ has density

$$p(y) = G_r(Qb, QBQ', y).$$

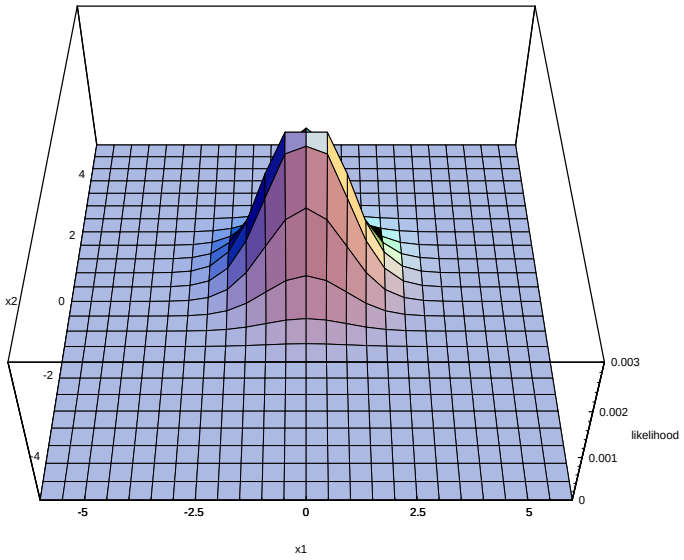

Figure 3: Product of the Gaussians is Gaussian (Theorem 2).

Notes:

1. The equivalence between (11) and (12) is known as the *matrix inversion lemma*. Another useful formula is

$$AQ'(QAQ' + B)^{-1} = (A^{-1} + Q'B^{-1}Q)^{-1}Q'B^{-1}.$$

2. The term $CQ'A^{-1}$ in (14) is known as the *Kalman gain matrix*.

5 The Linear-Gaussian Case

We now consider a linear-Gaussian case of model equations (1) and (2), and derive the Kalman filtering algorithm for efficiently computing $p(x_q|Z)$ in this case. First we define some variables. Let

$\tilde{x}$ be an $r \times 1$ random matrix (column vector),
Φ be an $r \times r$ matrix,
Γ be an $r \times u$ matrix,
$\tilde{w}$ be an $m \times 1$ random matrix (column vector),
$\tilde{z}$ be an $s \times 1$ random matrix (column vector),

H be an $s \times r$ matrix,
$\tilde{v}$ be an $s \times 1$ random matrix (column vector),
Q be an $m \times m$ symmetric, positive definite matrix,
R be an $s \times s$ symmetric, positive definite matrix,

σ be an $r \times 1$ matrix (column vector),
Σ be an $r \times r$ matrix,
λ be an $r \times 1$ matrix (column vector),
Λ be an $r \times r$ matrix,

where we have left off the subscripting. The variables in the first group occur directly in the model. Those in the second group are used in the algorithm, and of these only σ_0 and Σ_0 occur directly in the model as the known parameters for the density $p(x_0)$ of initial state $\tilde{x}_0$.

The linear-Gaussian model we consider is given by

$$\tilde{x}_{k+1} = \Phi_{k+1,k}\ \tilde{x}_k + \Gamma_{k+1,k}\ \tilde{w}_k \tag{18}$$

$$\tilde{z}_{k+1} = H_{k+1}\ \tilde{x}_{k+1} + \tilde{v}_{k+1}, \tag{19}$$

where in addition to the assumptions made for equations (1) and (2), we assume $p(w_k) = G_m(0, Q_k, w_k)$ and $p(v_{k+1}) = G_s(0, R_{k+1}, v_{k+1})$. That is, $\tilde{w}_k$ and $\tilde{v}_k$ are independent Gaussian random vectors with zero mean and covariance matrices Q_k and R_k, respectively. We are given that $p(x_0) = G_r(\sigma_0, \Sigma_0, x_0)$, with σ_0 a known $r \times 1$ matrix and Σ_0 a known $r \times r$ positive definite, symmetric matrix, i.e., $\tilde{x}_0$ is Gaussian with mean σ_0 and covariance matrix Σ_0.

Since (18) and (19) have the form of (1) and (2) an expression for the desired result $p(x_q|Z)$ is given by (8). Therefore we next compute expressions for $p(x_{k+1}|x_k)$ and $p(z_{k+1}|x_{k+1})$ to plug into (8). Using Theorem 3 on (18) and (19) we see

$$p(x_{k+1}|x_k) = G_r(\Phi_{k+1,k}\ x_k,\ \Gamma_{k+1,k} Q_k \Gamma'_{k+1,k}\ ,\ x_{k+1}) \tag{20}$$

$$p(z_{k+1}|x_{k+1}) = G_s(H_{k+1}\ x_{k+1},\ R_{k+1},\ z_{k+1}). \tag{21}$$

We now use (20) and (21), along with Theorem 2, to develop a pair of equations which we will then use to obtain an algorithm for evaluating $p(x_q|Z)$ in the integral expression (8). Using (20) followed by Theorem 2 we get

$$
\begin{aligned}
&\int dx_k\ G_r(\sigma_k, \Sigma_k, x_k) p(x_{k+1}|x_k) \\
&\quad = \int dx_k\ G_r(\sigma_k, \Sigma_k, x_k) G_r(\Phi_{k+1,k}\ x_k,\ \Gamma_{k+1,k} Q_k \Gamma'_{k+1,k}\ ,\ x_{k+1}) \\
&\quad = G_r(x_{k+1},\ \Gamma_{k+1,k} Q_k \Gamma'_{k+1,k} + \Phi_{k+1,k} \Sigma_k \Phi'_{k+1,k}\ ,\ \Phi_{k+1,k}\ \sigma_k) \qquad (22) \\
&\quad \equiv G_r(\lambda_{k+1}, \Lambda_{k+1}, x_{k+1}) \qquad (23)
\end{aligned}
$$

because the x_k term is shifted to a single, normalized Gaussian which integrates to 1. We now apply (21) followed by Theorem 2 again to get

$$
\begin{aligned}
&G_r(\lambda_{k+1}, \Lambda_{k+1}, x_{k+1}) p(z_{k+1}|x_{k+1}) \\
&\quad = G_r(\lambda_{k+1}, \Lambda_{k+1}, x_{k+1}) G_s(H_{k+1}\ x_{k+1}, R_{k+1}, z_{k+1}) \\
&\quad \propto G_r(c(H_{k+1}, z_{k+1}, R_{k+1}, \lambda_{k+1}, \Lambda_{k+1}), C(H_{k+1}, R_{k+1}, \Lambda_{k+1}), x_{k+1}) \qquad (24) \\
&\quad \equiv G_r(\sigma_{k+1}, \Sigma_{k+1}, x_{k+1}), \qquad (25)
\end{aligned}
$$

where functions c and C are defined in equations (12) and (14), and we have dropped the Gaussian term involving only constants to get proportionality.

Using these results we get an efficient algorithm, known as the Kalman filtering algorithm, for evaluating $p(x_q|Z)$ in the case of the linear-Gaussian model. We apply the above equations repeatedly in sequence, ending up with the result already normalized. In other words we evaluate (8) "left to right", where the previously evaluated part is collapsed into a single Gaussian. More specifically, start with known density $p(x_0) = G_r(\sigma_0, \Sigma_0, x_0)$ and compute $G_r(\lambda_1, \Lambda_1, x_1)$ using (22). Using this result compute $G_r(\sigma_1, \Sigma_1, x_1)$ using (24), followed by $G_r(\lambda_2, \Lambda_2, x_2)$ using (22), etc., with the final result being $p(x_q|Z) = G_r(\lambda_{n+1}, \Lambda_{n+1}, x_q)$.

Notes:

1. Using note 2 in Section 3 we see $p(x_k|(z_1, t_1), \ldots, (z_{k-1}, t_{k-1})) = G_r(\lambda_k, \Lambda_k, x_k)$ and $p(x_k|(z_1, t_1), \ldots, (z_k, t_k)) = G_r(\sigma_k, \Sigma_k, x_k)$. For this reason computing (22) is often called the *prediction step* and computing (24) the *measurement update step.*

2. The variable lettering in (18) and (19) follows that in [Med69].

3. Equation (12) is usually preferable to (11) for evaluating C since it uses fewer inverses. Similarly (14) is preferable to (13) for computing

c. The form of the equations for computing c and C starting with the Kalman gain matrix, given in [Med69] and elsewhere, is preferable computationally to the form we have presented. Instead of computing (24) using (12) and (14), the Kalman gain matrix is first computed as

$$K_{k+1} = \Lambda_{k+1} H'_{k+1} [H_{k+1} \Lambda_{k+1} H'_{k+1} + R_{k+1}]^{-1}.$$

Then it can be shown that

$$\Sigma_{k+1} = [I - K_{k+1} H_{k+1}] \Lambda_{k+1}$$

and

$$\sigma_{k+1} = \lambda_{k+1} + K_{k+1} [z_{k+1} - H_{k+1} \lambda_{k+1}].$$

Note that only one matrix inverse is required, and the matrix to be inverted has the dimensions of the *measurement* vector, not the *state* vector.

4. Note that matrix Φ in (18) need not be invertible. Also, setting a column of H in (19) to the zero vector corresponds to an element of the state vector which cannot be directly observed.

5. Since the time increments $t_{k+1} - t_k$ are known values the matrices $\Phi_{k+1,k}$ and $\Gamma_{k+1,k}$ can contain any functions of these time increments. If the state's motion is assumed to obey an arbitrary finite order, constant coefficient, linear, homogeneous differential equation in continuous time then matrix $\Phi_{k+1,k}$ becomes the matrix exponential which solves the equivalent system of first order equations for x_{k+1} with initial condition x_k.

6. While we have used the time-ordered "left to right" evaluation of the integrals in (8), Theorem 2 can be used to evaluate the integrals in essentially any order. For example if the prior $p(x_0)$ were not Gaussian and its product with a Gaussian required numerical integration then we could perform the integrals analytically with respect to the Gaussians and then perform a single numerical integration at the end with respect to x_0.

7. If a known $r \times 1$ vector u_k is added to the r.h.s. of (18), sometimes called a *control term*, the only effect is to shift the mean of the Gaussian in (20). The algorithm changes in that $\lambda_{k+1} \equiv \Phi_{k+1,k}\, \sigma_k + u_k$ in (22) and (23).

8. There are a number of variations and extensions of the basic Kalman filter algorithm to address, among other things, nonlinearities in the motion model and numerical stability. See [CT84] for a survey. See also [MS83] for another Bayesian derivation of the Kalman filter, and [BSF88] for a least squares approach and many additional references.

6 Examples

In this section we give some examples applying the preceeding results. A general approach for a practical (as opposed to theoretical) application follows. While we present the approach as a sequence of steps, the steps are really interdependent.

step 1: Determine an appropriate state representation of the problem and define the state transition function, or motion model, as a discrete-time, deterministic function f. That is, assume all quantities including errors, etc., are known exactly. Similarly, describe the measurement model as a deterministic function g of the known state.

step 2: Replace x_0 and all unknown quantities (corresponding to π in f and ϕ in g) by random variables. Write all variables which are functions of random variables, e.g. the x_k and z_k for $k > 0$, as random variables. In our notation this step involves simply placing the tilde symbol above the random quantities.

step 3: Define densities for the random variables $\tilde{x}_0$, $\tilde{\pi}$, and $\tilde{\phi}$ introduced in step 2.

step 4: Make sure the independence assumptions of Section 1 hold. If not, either revise the models definitions in step 1, try an augmented state vector approach, or use more general results than those we have presented.

step 5: Compute the densities $p(x_{k+1}|x_k)$ and $p(z_{k+1}|x_{k+1})$. This is possible in theory, but may be difficult in practice. Alternately, we could start with this as the first step and *define* these densities as the model.

step 6: Gather observed data Z from the system being modeled.

step 7: Use equation (8), or an equivalent form, to predict the density for the state $\tilde{x}_q$ at future time t_q. If the model has linear-Gaussian

form and is of low enough dimension to compute the necessary matrix inverses then use the Kalman filter algorithm. Otherwise, develop an acceptably efficient algorithm for evaluating (8). This can be extremely difficult or impossible if the model is not chosen with care, and numerical or approximation techniques may be necessary.

step 8: Test the model's predictions and, if necessary, refine the model.

We emphasize that equation (8) is an extremely general mathematical result about estimation for discrete-time Markov systems with independent random disturbances and measurements not affecting the state. As such it has applications in finance, economics, engineering, the sciences, etc. The generality of systems having the form of (1) and (2) is illustrated in Example 1, concerning a Turing machine. This is a somewhat theoretical example. The random variables involved are purely discrete, thus all integrals are interpreted as sums. Example 2 is a dynamic classification problem. The state vector in this example is a combination of a discrete classification variable and continuous variables corresponding to signal values. Example 3 concerns tracking a moving object, and is worked out in some detail. In this example the model is linear-Gaussian, thus all random variables are continuous and the Kalman filtering algorithm can be applied.

6.1 Example 1: An imperfectly observed, probabilistic Turing machine

A *Turing machine* [Men64, HU79] is a model of effective computation; no known deterministic computations have been shown to be non-computable in principle by a Turing machine. Informally, a Turing machine can be thought of as a semi-infinite tape of symbols scanned by a tape head. At time t_0 the tape contains the initial tape input in the leftmost cells of the tape, and a special blank symbol in the remaining cells. A move of the Turing machine takes it from time k to time $k+1$ by writing a new character at the position of the tape head and then moving the head left or right.

More formally, let Q be a finite set of states, and let $A = \{0, 1, B\}$ be an alphabet of characters. For our purposes, Turing machine M at time k is characterized by a state q_k from a finite set of possible states Q, an n_k dimensional row vector (or string, or array) of characters $T_k \in A^{n_k}$ called the tape, and an integer p_k indexing the place of the "tape head" on the tape. For this example we will use array notation and write $T_k[i]$ for ith element of the tape, with $i \geq 0$. We also define $c_k = T_k[p_k]$ as the character currently

being "scanned" by the tape head. We thus write the Turing machine at time k as the vector $M_k = (q_k, n_k, T_k, p_k)'$.

The initial state vector of the Turing machine is given as $M_0 = (q_0, n_0, T_0, 0)'$. A function $Move$ takes the machine from time k to time $k+1$ as $M_{k+1} = Move(M_k)$. We assume we have the functions

$$Move_q : Q \times A \rightarrow Q, \tag{26}$$

$$Move_T : Q \times A \rightarrow A, \tag{27}$$

$$Move_p : Q \times A \rightarrow \{-1, 1\}. \tag{28}$$

Then we define the total $Move()$ function as

$$Move(M_k) = (Move_q(q_k, c_k), n_k + 1, T_{k+1}, p_k + Move_p(p_k, c_k))', \tag{29}$$

where $T_{k+1}[n_{k+1}] = B$, $T_{k+1}[p_k] = Move_T(q_k, c_k)$, and $T_{k+1}[i] = T_k[i]$ otherwise. Function $Move$ is not defined for all machine states M_k, and the machine is said to halt at time k if $Move(M_k)$ is undefined or if $p_{k+1} = -1$. Note that the size of the tape increases with each time step to give a constructively infinite tape.

A *probabilistic Turing machine* [San69] can be characterized as a Turing machine where the function $Move_q$ takes an additional, discrete, independent random variable as an argument [Gil77]. This random variable is restricted to have finite range, i.e., it can have only finitely many possible values. Thus, for example, in (29) $Move_q$ is replaced with $Move_q(q_k, c_k, \tilde{\pi})$, so the next state is also a random variable, etc. Note that our formulation is slightly different from those in [San69] and [Gil77]. A *nondeterministic* Turing machine, in standard computer science terminology, can be characterized as a Turing machine where $Move_q$ is multiple-valued. The nondeterministic machine essentially branches and computes the results for all possible q values at each time step. An equivalent probabilistic Turing machine is a machine which computes all outputs having nonzero probability, and where each possible $Move_q$ value in the nondeterministic machine is assumed equally likely.

To put the Turing machine into the form of (1) we take $x_k \equiv M_k$, and $x_{k+1} = Move(x_k)$. The state is essentially the *instantaneous description*, or ID, of the machine at any given time. With this information stored in the state vector no information about previous states is needed to run the machine forward in time.

We have mapped the Turing machine into states x_k of the form (1), but we have not yet defined the sensor model g of (2). We take the function g

to model an external agent's observation of a probabilistic Turing machine as it evolves in time. For example we might have $\tilde{z}_{k+1} = \tilde{x}_{k+1} + \tilde{\phi}_{k+1}$, with $\tilde{\phi}$ an appropriately dimensioned vector of independent random variables, so the observer gets data corrupted by additive noise. As another example, the observer may not be able to read some tape cells at all, but can read all other state information perfectly. This type of model relates to another characterization of nondeterministic machines, where the machine is allowed to "guess an input structure" [GJ79].

6.2 Example 2: Dynamic classification of an input signal

Consider a discrete-time signal $s(t_k) \in \Re^d$, $k = 0, 1, 2, \ldots$. For example, $s(t_k)$ might be amplitude values or time-localized frequency values taken at discrete times from a person's continuous speech stream. For simplicity, assume the signal is sampled at constant time intervals. Assume that at each time point t_k the signal also has associated with it a discrete class value ω_k, from a known, finite set of classes Ω. In the speech processing example we will take the class ω_k to represent the current word (or phoneme) being spoken. The state representation we define for this problem is as follows. The state x_k is a vector of the previous L signal attribute vectors s along with a discrete classification variable, i.e.,

$$x_k \equiv (s(t_k)', s(t_{k-1})', \ldots, s(t_{k-L})', \omega_k)'. \tag{30}$$

Now, we cannot observe the state directly. At each time point t_k we measure a vector of real-valued signal attributes z_k. For example, the vector z_k might be noisy measurements taken from a microphone input. The objective is to predict the class ω_k at each time $k \geq L$. If s is continuous then these measurement times determine the discrete time instants t_k at which we consider the continuous signal $s(t)$. Of course we cannot measure the class value ω_k directly, even with noise. Including it in the state representation is a convenient fiction introduced to facilitate modeling.

Notice that we took the state to include the previous L attribute vectors to allow more realistic motion models. In the speech context this means that the current word being spoken is modeled as being a function of the speech attributes at the previous L measurement times. Thus the model could take into account a "sentence-like" block of previous speech with large enough L. The classification variable ω_k could alternately be defined as the word spoken at a *previous* time instant, to take information from the succeeding

speech into account, or the problem could be formulated as a smoothing problem.

The function f in (1), or equivalently the density $p(x_{k+1}|x_k)$, is chosen to model the flow of the state, with random elements taking uncertainties into account. Thus in the speech data example it is a stochastic speech model. We need to model, or estimate, the joint probability of $s(t_{k+1})$ and ω_{k+1} given their values at the previous L time instants. Creating this model is one of the most difficult and important tasks in an effective application. We will not be more specific in describing f.

The function g in (2), or equivalently the density $p(z_{k+1}|x_{k+1})$, is chosen to model the errors introduced in the measurement process. In the speech example we model the signal transformation and noise introduced by the microphone and any other sources, as well as the fact that we cannot directly observe the class value ω_k. For concreteness, assume g is of linear-Gaussian form (19), where H is a matrix having partitioned form $(I^{(d\times d)}|0^{(d\times(L-1)d)}|0^{(d\times 1)})$ with superscripts denoting the matrix dimensions. Thus at time $k+1$ the d-dimensional measurement z_{k+1} is the true signal vector $s(t_{k+1})$ corrupted by additive Gaussian noise. That is, $p(z_{k+1}|x_{k+1}) = p(z_{k+1}|s(t_{k+1})) = G_d(s(t_{k+1}), I, z_{k+1})$, where we have taken the correlation matrix to be the identity matrix.

If we are given a set of data, equation (8) "solves" the problem of finding the posterior density for the current state given the observed data. Since the class ω is a component of the state vector we can in principle integrate out the other components to obtain a probability for each classification at each time step. Of course (8) is really just a starting point. The models and estimation algorithms must be chosen so that an acceptably efficient implementation, in terms of running time and accuracy, can be found. Even given a model with an acceptably efficient implementation, the problem of estimating the parameters of the model remains. In this type of problem one typically has a set of *training data* with known classifications, and the goal is to set the parameters of the model to maximize the posterior density of the parameter vector given the training data. Since finding the optimal parameter vector is typically intractable, approximate algorithms like hill-climbing or annealing are used. See [Rab89] for a tutorial introduction to Markov models in speech recognition and a discussion of the many practical problems that arise.

6.3 Example 3: Tracking a moving object

For this example we modify our notation slightly. We drop the convention that subscripts are implicit time arguments, and instead we write the time arguments explicitly. Thus we write $x(t_k)$ where before we simply wrote x_k. We use subscripts instead to indicate the elements of vectors. For example $x_i(t_k)$ is the ith component of vector x at time t_k.

Consider a ball of known mass m thrown at a robot. The goal of the robot is to predict the position of the ball at time t_q, perhaps as a subproblem in an attempt to catch the ball. Over time the robot receives noisy sensor measurements about the position of the ball. We assume the raw sensor measurements have been preprocessed into position estimates contaminated with additive Gaussian noise. This noise is assumed to have a known covariance matrix at each measurement time.

In what follows we first define the state-vector representation of the thrown-ball system. Then we define the *true* differential equations describing the motion of the ball. Next, we define the robot's *model* of the ball's motion equations – these equations are not identical to the true equations of motion. We then solve for the discrete-time form of the robot's model and include a noise term to help compensate for the fact that the model is incorrect. After defining the true motion model and the robot's motion model we define the true sensor model, and assume the robot uses the true sensor model as well. The robot's model corresponds to equation (1) in the general system, and to equation (18) in the linear-Gaussian system. The sensor model corresponds to (2) in the general system and (19) in the linear-Gaussian system. Finally, we give a specific example.

The coordinate system is a fixed, rectangular system with the third coordinate as the vertical direction. We take the state $x(t_k)$ of the system to be the position of the center of mass of the ball, $y(t_k) = (y_1(t_k), y_2(t_k), y_3(t_k))'$, along with its velocity vector $v(t_k) = (v_1(t_k), v_2(t_k), v_3(t_k))'$. Thus

$$\begin{aligned} x(t_k) &\equiv (y(t_k)', v(t_k)')' \\ &\equiv (y_1(t_k), y_2(t_k), y_3(t_k), v_1(t_k), v_2(t_k), v_3(t_k))' \\ &\equiv (x_1(t_k), x_2(t_k), x_3(t_k), x_4(t_k), x_5(t_k), x_6(t_k))'. \end{aligned} \tag{31}$$

For the purposes of this example we assume the *true* motion of the ball is described by Newton's law as

$$m\ a(t) = Force(t) = m(0, 0, -g)' - \alpha(v_1(t), v_2(t), v_3(t))', \tag{32}$$

where the ball experiences a retarding force proportional to its velocity in addition to the force of gravity. The retarding force might, for example, be due to air resistance. Thus

$$m(\ddot{y}_1, \ddot{y}_2, \ddot{y}_3)' = m(0, 0, -g)' - \alpha(\dot{y}_1, \dot{y}_2, \dot{y}_3)' \tag{33}$$

or

$$\begin{pmatrix} \ddot{y}_1 + \alpha\dot{y}_1/m \\ \ddot{y}_2 + \alpha\dot{y}_2/m \\ \ddot{y}_3 + \alpha\dot{y}_3/m + g \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}. \tag{34}$$

Notice that the system can be easily decoupled into three independent systems and solved separately. For the purposes of this example, though, we consider all the equations simultaneously. Rewriting in terms of the state vector x, defined in (31), we have

$$\begin{pmatrix} \dot{x}_1(t) \\ \dot{x}_2(t) \\ \dot{x}_3(t) \\ \dot{x}_4(t) \\ \dot{x}_5(t) \\ \dot{x}_6(t) \end{pmatrix} = \begin{pmatrix} x_4(t) \\ x_5(t) \\ x_6(t) \\ -\alpha x_4(t)/m \\ -\alpha x_5(t)/m \\ -\alpha x_6(t)/m - g \end{pmatrix}. \tag{35}$$

Now, the robot is assumed to *model* the thrown-ball system as (35) but with $\alpha = 0$. That is, the retarding force is not accounted for in the model. In Figure 4 we show the x_2 and x_3 components of a ball's trajectory with $\alpha = .5$, the true model, as a dashed line. A ball's trajectory with $\alpha = 0$, the robot's assumed model, is the solid line. The initial condition for both cases is $x(t_0) = (0, 0, 0, 5, 5, 5)'$ and we take mass $m = 1$.

Solving the robot's model system for discrete-time and including additive noise in the velocity transitions to compensate for modeling errors we obtain

$$x(t_{k+1}) = \Phi(t_{k+1}, t_k)x(t_k) + \Gamma(t_{k+1}, t_k)w(t_k) + u(t_k), \tag{36}$$

where $w(t_k)$ is a 3-dimensional error or noise vector. Matrices Φ and Γ and vector u will be defined next. Defining the time increment $\Delta t_k = t_{k+1} - t_k$ and writing out the matrices equation (36) becomes

$$\begin{pmatrix} x_1(t_{k+1}) \\ x_2(t_{k+1}) \\ x_3(t_{k+1}) \\ x_4(t_{k+1}) \\ x_5(t_{k+1}) \\ x_6(t_{k+1}) \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & \Delta t_k & 0 & 0 \\ 0 & 1 & 0 & 0 & \Delta t_k & 0 \\ 0 & 0 & 1 & 0 & 0 & \Delta t_k \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1(t_k) \\ x_2(t_k) \\ x_3(t_k) \\ x_4(t_k) \\ x_5(t_k) \\ x_6(t_k) \end{pmatrix} \tag{37}$$

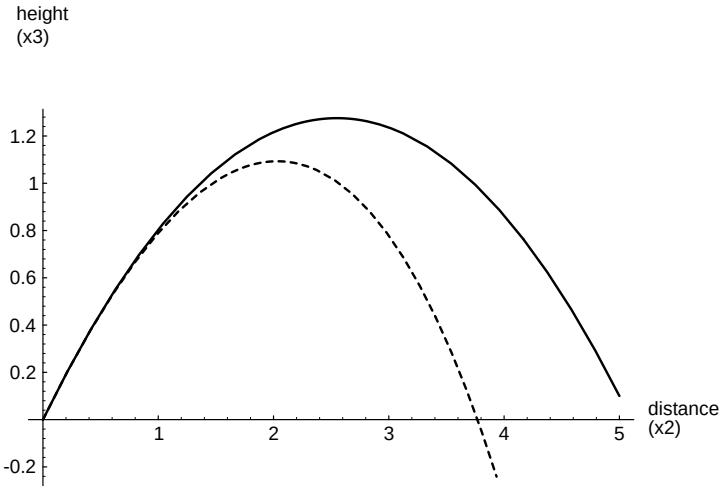

Figure 4: Trajectory of the ball under two different models.

$$+ \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ \Delta t_k & 0 & 0 \\ 0 & \Delta t_k & 0 \\ 0 & 0 & \Delta t_k \end{pmatrix} \begin{pmatrix} w_1(t_k) \\ w_2(t_k) \\ w_3(t_k) \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ -(\Delta t_k)^2 g/2 \\ 0 \\ 0 \\ -(\Delta t_k) g \end{pmatrix}.$$

We can verify that (36) with $w(t_k) = 0$ solves (35) with $\alpha = 0$ by differentiating. Notice that the additive noise in the velocity transitions grows linearly larger for longer time periods because of the way Γ is defined. We have used a control term $u(t_k)$ (see note 7, Section 5) to incorporate terms due to the constant accelleration. We could alternately have expanded to a 9-dimensional state representation by including three accelleration components in the state.

We now consider the sensor model. The robot receives sensor measurements only about the position of the ball, with Gaussian errors. Thus we have

$$z(t_{k+1}) = H(t_{k+1})x(t_{k+1}) + r(t_{k+1}), \tag{38}$$

or, writing out the matrices,

$$\begin{pmatrix} z_1(t_{k+1}) \\ z_2(t_{k+1}) \\ z_3(t_{k+1}) \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} x_1(t_{k+1}) \\ x_2(t_{k+1}) \\ x_3(t_{k+1}) \\ x_4(t_{k+1}) \\ x_5(t_{k+1}) \\ x_6(t_{k+1}) \end{pmatrix} + \begin{pmatrix} r_1(t_{k+1}) \\ r_2(t_{k+1}) \\ r_3(t_{k+1}) \end{pmatrix}. \tag{39}$$

Here r is a sensor error term (we have used the letter r instead of v for the error term to avoid conflict with the velocity vector v). Unlike the error terms w_k in the motion model (36), the r_k error terms are not assumed to affect the ball's trajectory; each r_k affects only a single measurement z_k. Notice that because of the way H is defined the robot can only directly sense the position of the ball, not its velocity, and these measurements are corrupted with noise.

Now that we have defined both the robot's model of the ball's motion and its sensor model as deterministic processes, we consider the unknown quantities to be random variables. Thus we define densities for the quantities $w(t_k)$, $r(t_k)$, and $x(t_0)$. In terms of the previous notation we would now write the tilde symbol above these quantities and all functions of these quantities, but we will not do this explicitly. We define $p(w(t_k)) = G_3(0, I, w(t_k))$, $p(r(t_k)) = G_3(0, I/4, r(t_k))$, and $p(x(t_0)) = G_6(\sigma(t_0), I, x(t_0))$. We take $\sigma(t_0) = (0, 0, 0, 5, 5, 5)'$ to be the true initial condition as well as the mean of the modeled prior $p(x(t_0))$. Notice that with these Gaussian density assumptions the robot's model has linear-Gaussian form. That is, equation (36) corresponds to (18), and (38) corresponds to (19). Thus, given a data set Z, we can use the Kalman filtering algorithm to compute the density for the state vector at any future time.

The Kalman filter algorithm allows us to compute a density estimating the state at any future time, and this estimate is Gaussian. If the robot's model were identical to the true model then these estimates would be the true densities for the future states. Since the model is incorrect, though, they are only approximations. We may want the density for a subset of the state elements, rather than for the entire state. For example, we may want the density for only the position coordinates. In this case, because the densities are Gaussian, we do not need to explicitly integrate out the other components. The estimate for a subset of coordinate positions is again Gaussian and is obtained by simply eliminating the unwanted coordinate po-

sitions and their corresponding rows and columns in the covariance matrix. This can be seen by applying Theorem 3 with the matrix Q (of Theorem 3) chosen to eliminate the unwanted coordinate positions, leaving the others unchanged.

Assume that we observe the system at times $t_1 = 1/8, t_2 = 1/3$, and $t_3 = 1$, and take $t_0 = 0$. We are given the data set $Z = \{(z(t_1), t_1), (z(t_2), t_2), (z(t_3), t_3)\}$ sensed from the true trajectory with $\alpha = 0.5$. Figure 5 is a phase plot of the x_5 and x_6 components of the true trajectory. Time is not shown and could be considered to be "coming out of the page". The small circles along the path indicate the ball's true position at measurement times $t_1, \ldots, t_3$. According to the sensor model, a measurement is a sample from a Gaussian with mean at the true position. The larger gray circles are the 39.3 percent probability ellipses (circles in this case) for the measurements. The small squares show the actual sampled values, which are the observed measurements (or the realizations of the $\tilde{z}_i$).

In Figure 6 we show the x_2 and x_3 components of the mean for the Kalman filter estimate of the state. At the end of the first three of these segments we show the 39.3 percent probability ellipse around the mean in gray. Actually, though, each point on the curves is a mean value and has a Gaussian density associated with it. Note that since the probability ellipses do not depend on the measured data they remain circular (rather than becoming elliptical) through time. The true trajectory is again shown as the dashed curve. Time increases left to right along each piece of the estimate. The breaks occur whenever a new measurement is received. The leftmost solid segment is the mean of $p(x(t))$ for $t_0 \leq t < t_1$. The next solid segment is the mean of $p(x(t)|(z(t_1), t_1))$ for $t_1 \leq t < t_2$, followed by the mean of $p(x(t)|(z(t_1), t_1), (z(t_2), t_2))$ for $t_2 \leq t < t_3$, etc. Only the final solid segment is conditioned on all the data Z. After receiving all the data the other estimates could be improved, if desired, by formulating the problem as a smoothing problem (see note 7, Section 2). Notice that the estimates tend to overshoot the true trajectory because the model does not take the velocity-dependent retarding force into account. Immediately after measurements are received the estimates tend to improve.

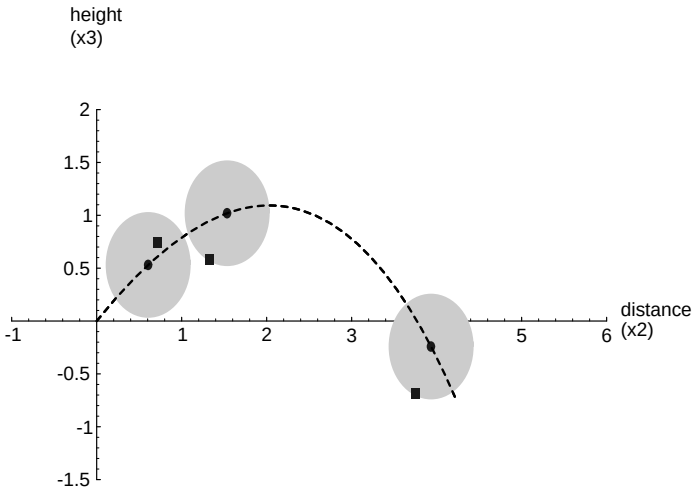

Figure 5: The ball's true path and sensor data measured from it.

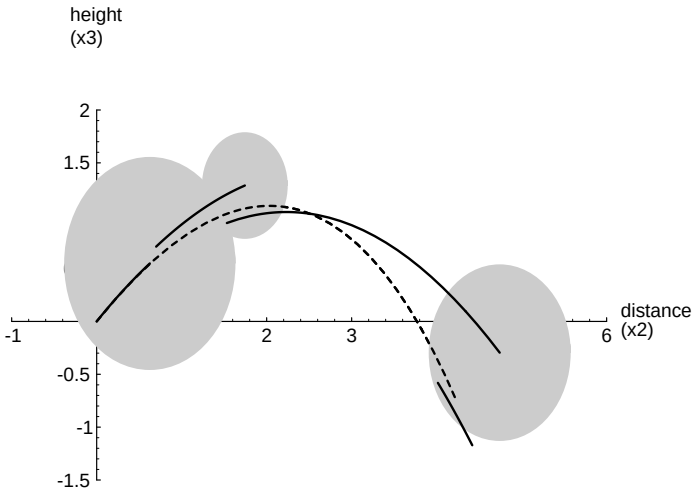

Figure 6: Estimates of the ball's path and the true path.

References

[Ber85] James O. Berger. *Statistical Decision Theory and Bayesian Analysis*. Springer-Verlag, 1985.

[Ber92] L. Mark Berliner. Statistics, probability and chaos. *Statistical Science*, 7(1):69–122, 1992.

[BSF88] Yaakov Bar-Shalom and Thomas E. Fortmann. *Tracking and Data Association*. Academic Press, 1988.

[CT84] Chaw-Bing Chang and John A. Tabaczynski. Application of state estimation to target tracking. *IEEE Transactions on Automatic Control*, AC-29(2), 1984.

[CY92] Sangit Chattergee and Mustafa R. Yilmaz. Chaos, fractals and statistics. *Statistical Science*, 7(1):49–121, 1992.

[Dil94] Dan Dill. *Interactive TeX/Mathematica documents*, Feb. 1994.

[Gil77] John Gill. Computational complexity of probabilistic turing machines. *SIAM Journal on Computing*, 6(4):675–695, 1977.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability*. Freeman, 1979.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[JH69] Arthur E. Bryson Jr. and Yu-Chi Ho. *Applied Optimal Control*. Ginn and Company, 1969.

[JW88] Richard A. Johnson and Dean W. Wichern. *Applied Multivariate Statistical Analysis*. Prentice Hall, 1988.

[MCTW86] Shozo Mori, Chee-Yee Chong, Edison Tse, and Richard P. Wishner. Tracking and classifying multiple targets without a priori identification. *IEEE Transactions on Automatic Control*, AC-31(5), 1986.

[Med69] J. S. Meditch. *Stochastic Optimal Linear Estimation and Control*. McGraw Hill, 1969.

[Men64] Elliot Mendelson. *Introduction to Mathematical Logic*. Van Nostrand, 1964.

[MS83] Richard J. Meinhold and Nozer D. Singpurwalla. Understanding the kalman filter. *The American Statistician*, 37(2):123–127, 1983.

[Rab89] Lawrence R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–285, 1989.

[Sal89] D. J. Salmond. Tracking in uncertain environments. Technical report, Royal Aerospace Establishment, Farnborough, Hants, UK, September 1989. From a Ph.D. thesis, University of Sussex.

[San69] Eugene S. Santos. Probabilistic turing machines and computability. *Proceedings of the American Mathematical Society*, 22:704–710, 1969.

[Wol91] Stephen Wolfram. *Mathematica: A System for Doing Mathematics by Computer*. Addison Wesley, 1991.

A The Kalman Filter in Mathematica

This appendix contains Mathematica [Wol91] code for generating some of the graphs, as well as Example 3. The code was written and typeset using the TeX/Mathematica system [Dil94].

First we perform some setup.

```
(
<<Statistics`ContinuousDistributions`
SetOptions[Plot, Frame->True];
$DefaultFont = {"Helvetica",4};
SeedRandom[31454623];
dash1 = { Dashing[{0.005, 0.005}] };
dash2 = { Dashing[{0.01, 0.01}] };
dash3 = { Dashing[{0.01, 0.01, 0.001, 0.01}] };
solid = { };
)
```

Now we define the r-dimensional Gaussian density as in (15) to (17).

```
(
Mahalanobis[r_,a_,A_,x_] := (1/2)(x-a).Inverse[A].(x-a);
J[r_,A_] := (2 Pi)^(-r/2) Det[A]^(-1/2);
Gaussian[r_,a_,A_,x_] := J[r,A] Exp[-Mahalanobis[r,a,A,x]];
)
```

This code produces the illustrations of Theorem 2 in Section 4.

```
(
$DefaultFont = {"Helvetica", 2};
Print["Doing combined plot.."];
figure1=Plot3D[Evaluate[
               Gaussian[2,{-1.5,0},{{4,2},{2,2}},{x,y}]],
             {x,-6,6}, {y,-5,5},
             AxesLabel -> {"x1   ","x2   ","density"},
             AxesEdge -> {Automatic,{-1,-1},Automatic},
             PlotRange -> {0,.08},
             ViewPoint -> {0,-2,2},
             PlotPoints -> 25,
             DisplayFunction -> Identity,
             ClipFill -> None];
```

```
figure2=Plot3D[Evaluate[
               Gaussian[2,{1.5,0},{{4,-2},{-2,2}},{x,y}]],
             {x,-6,6}, {y,-5,5},
             AxesLabel -> {"x1   ","x2   ","density"},
             AxesEdge -> {Automatic,{-1,-1},Automatic},
             PlotRange -> {0,.08},
             ViewPoint -> {0,-2,2},
             PlotPoints -> 25,
             DisplayFunction -> Identity,
             ClipFill -> None];
figure=Show[GraphicsArray[{figure1,figure2}]];
PSTeX[figure,"gaussianPlot"];
Print["Doing product plot.."];
figure3=Plot3D[Evaluate[
               Gaussian[2,{-1.5,0},{{4,2},{2,2}},{x,y}]
                 Gaussian[2,{1.5,0},{{4,-2},{-2,2}},{x,y}]],
             {x,-6,6}, {y,-5,5},
             AxesLabel -> {"x1   ","x2   ","likelihood"},
             AxesEdge -> {Automatic,{-1,-1},Automatic},
             PlotRange -> {0,.003},
             ViewPoint -> {0,-2,2},
             PlotPoints -> 25,
             ClipFill -> None];
PSTeX[figure3,"productPlot"];
)
.
Doing combined plot..

PSTeX::file: Graphics being processed (without prolog) to file
    "gaussianPlot.ps".
Doing product plot..

PSTeX::file: Graphics being processed (without prolog) to file
    "productPlot.ps".
```

Now we compute Example 3 in Section 6. The code is a basic loop implementation of the Kalman filter algorithm. Alternately, we could have defined transformation rules corresponding to Theorem 2 and had Mathematica evaluate (8) automatically.

We first define the true motion equations of the thrown-ball system and solve for the discrete-time equations.

```
(
stateVector[t_] := { x1[t], x2[t], x3[t], x4[t], x5[t], x6[t] };
system[t_,alpha_] :=
```

```
       { x1'[t] ==  x4[t],
         x2'[t] ==  x5[t],
         x3'[t] ==  x6[t],
         x4'[t] == -alpha x4[t] / m,
         x5'[t] == -alpha x5[t] / m,
         x6'[t] == -g + (-alpha x6[t] / m),
         x6[0] == i6, x5[0] == i5, x4[0] == i4,
         x3[0] == i3, x2[0] == i2, x1[0] == i1 };
soln[t_,alpha_] := DSolve[system[t,alpha], stateVector[t], t];
x[t_,0] = stateVector[t] /. soln[t,0][[1]];
x[t_,alpha_] = stateVector[t] /. soln[t,alpha][[1]];
Print["System solution for alpha = 0:"];
Print["x[t] = ", x[t,0]];
Print["System solution for alpha > 0:"];
Print["x[t] = ", x[t,a]];
)
.
LinearSolve::nosol: Linear equation encountered which has no solution.

LinearSolve::nosol: Linear equation encountered which has no solution.
System solution for alpha = 0:
                                                 2
                                               g t
x[t] = {i1 + i4 t, i2 + i5 t, i3 + i6 t - ----, i4, i5, i6 - g t}
                                               2
System solution for alpha > 0:
             i4 m      i4 m          i5 m      i5 m
x[t] = {i1 + ---- - ----------, i2 + ---- - ----------,
              a        (a t)/m        a        (a t)/m
                     a E                     a E

                                g m
                     2   m (i6 + ---)
          i6 m   g m             a     g m t     i4        i5
>    i3 + ---- + ---- - ------------ - -----, --------, --------,
           a       2        (a t)/m      a     (a t)/m   (a t)/m
                  a      a E                 E         E

                   g m
                i6 + ---
       g m          a
>    -(---) + --------}
         a      (a t)/m
                E
```

Now we generate Figure 4. We plot two position coordinates of the state, x_2

and x_3, for the two different models. We take mass $m = 1$ and and time $0 \leq t \leq 1$. We define the initial condition to be $x(t_0) = (0, 0, 0, 5, 5, 5)'$.

```
(
i1=0; i2=0; i3=0; i4=5; i5=5; i6=5;
g=9.8; m=1;
figure=ParametricPlot[
          Evaluate[{{x[t,0][[2]],x[t,0][[3]]},{x[t,.5][[2]],x[t,.5][[3]]}}],
          {t,0,1},
          PlotRange -> Automatic,
          PlotStyle -> { solid, dash2 },
          AxesLabel -> {"distance\n(x2)","height\n(x3)"}];
PSTeX[figure, "ballPaths"]
)
.
PSTeX::file: Graphics being processed (without prolog) to file "ballPaths.ps".

Out[4]= -Graphics-
```

Next we measure some data from the system. First we define the measurement times, along with some constants. We also define the measurement matrix H. Then we run the system forward and "measure" the data Z from the *true* state vector.

```
(
g = 9.8; alpha = 0.5; m = 1;
t[0]=0; t[1]=1/8; t[2]=1/3; t[3]=1; t[4]=1.1;
H = {{1,0,0,0,0,0},
     {0,1,0,0,0,0},
     {0,0,1,0,0,0}};
Do[(
   (* get a random noise vector *)
   v[k+1] = { Random[NormalDistribution[0,1/2]],
              Random[NormalDistribution[0,1/2]],
              Random[NormalDistribution[0,1/2]] };
   (* measure the position coordinates with noise *)
   z[k+1] = H . x[t[k+1],0.5] + v[k+1];
   ),{k,0,2}];
)
```

Now that we have measured the data from the true system, we define the parameters of the *model* system. We define the matrices Φ and Γ which appear in the model, as well as the vector u. We take the definition of H as that given

previously, the true measurement matrix. Note that the assumed measurement model is correct.

```
(
Phi[t1_,t0_] := {{1,0,0,t1-t0,0,0},
                 {0,1,0,0,t1-t0,0},
                 {0,0,1,0,0,t1-t0},
                 {0,0,0,1,0,0},
                 {0,0,0,0,1,0},
                 {0,0,0,0,0,1}};
Unprotect[Gamma];
Gamma[t1_,t0_] := {{0,0,0},
                   {0,0,0},
                   {0,0,0},
                   {t1-t0,0,0},
                   {0,t1-t0,0},
                   {0,0,t1-t0}};
u[t1_,t0_] := { 0,
                0,
                -g (t1-t0)^2 / 2,
                0,
                0,
                -g (t1-t0)}
)
```

Next we define the means and covariance matrices for the densities of the unknown quantities, which we assumed to be Gaussian.

```
(
sigma[0]={0,0,0,5,5,5};
Sigma[0]=IdentityMatrix[6];
R = IdentityMatrix[3] / 4;
Q = IdentityMatrix[3];
)
```

Now we define C and c as in (12) and (14).

```
(
CFun[Q_,A_,B_] := B - B.Transpose[Q].Inverse[A + Q.B.Transpose[Q]].Q.B;
cFun[Q_,a_,A_,b_,B_] := b + CFun[Q,A,B].Transpose[Q].Inverse[A].(a-Q.b);
)
```

We now define the Kalman filter relations and loop over the data. After this step the Σ_k and Λ_k matrices and the σ_k and λ_k vectors will have been calculated. In practice we could update our estimates after each measurement.

```
(
Do [(
   Lambda[k] = Gamma[t[k],t[k-1]] . Q . Transpose[Gamma[t[k],t[k-1]]]
               + Phi[t[k],t[k-1]] . Sigma[k-1] . Transpose[Phi[t[k],t[k-1]]];
   Sigma[k] = CFun[H, R, Lambda[k]];
   lambda[k] = Phi[t[k],t[k-1]] . sigma[k-1] + u[t[k],t[k-1]];
   sigma[k] = cFun[H,z[k],R,lambda[k],Lambda[k]];
   ),{k,1,3}];
)
```

Now we plot the estimated final two state coordinates.

```
(
(* define the prediction from a t_k time to an arbitrary time *)
meanOfPrediction[lowK_,tVar_] :=
           Phi[tVar,t[lowK]] . sigma[lowK] + u[tVar,t[lowK]];
covOfPrediction[lowK_,tVar_] :=
         Gamma[tVar,t[lowK]].Q.Transpose[Gamma[tVar,t[lowK]]]
         + Phi[tVar,t[lowK]].Sigma[lowK].Transpose[Phi[tVar,t[lowK]]];
(* loop and compute each segment of the estimate's mean *)
Do [(
   fig[k] = ParametricPlot[Evaluate[Take[meanOfPrediction[k-1,tVar],{2,3}]],
                           {tVar,t[k-1],t[k]},
                           DisplayFunction -> Identity,
                           AxesLabel -> {"distance\n(x2)","height\n(x3)"}];
   covEllipse[k] = Graphics[{GrayLevel[.8], Disk[
                             Take[meanOfPrediction[k-1,t[k]],{2,3}],
                             N[Sqrt[ covOfPrediction[k-1,t[k]] [[3,3]] ]]] } ];
   ),{k,1,4}];
(* plot the true path of the ball *)
truePath = ParametricPlot[
           Evaluate[{x[t,.5][[2]],x[t,.5][[3]]}],
           {t,0,1.1},
           PlotRange -> {{-1,6},{-1.5,2.0}},
           PlotStyle -> dash2,
           AxesLabel -> {"distance\n(x2)","height\n(x3)"}];
(* plot circles along the true path at the measurement times,     *)
(* and covariances of measurement errors centered at these points *)
timePoints = Graphics[{{GrayLevel[0.8],
                  Disk[{x[t[1],.5][[2]],x[t[1],.5][[3]]},1/2],
```

```
                    Disk[{x[t[2],.5][[2]],x[t[2],.5][[3]]},1/2],
                    Disk[{x[t[3],.5][[2]],x[t[3],.5][[3]]},1/2]},
                   {GrayLevel[0.1],
                    Disk[{x[t[1],.5][[2]],x[t[1],.5][[3]]},.05],
                    Disk[{x[t[2],.5][[2]],x[t[2],.5][[3]]},.05],
                    Disk[{x[t[3],.5][[2]],x[t[3],.5][[3]]},.05]}
                  }];
(* plot squares at the measured data points *)
fillSquare[p_,w_] := Rectangle[{p[[1]]-w,p[[2]]-w},{p[[1]]+w,p[[2]]+w}];
dataZ = Graphics[{
                    GrayLevel[0.1],
                    fillSquare[Take[z[1],-2],.05],
                    fillSquare[Take[z[2],-2],.05],
                    fillSquare[Take[z[3],-2],.05]
                  }];
(* combine plots to create Figure 5 *)
measuredFig = Show[truePath,timePoints,dataZ,truePath];
PSTeX[measuredFig, "measuredData"];
(* combine plots to create Figure 6 *)
estimateFig = Show[truePath,covEllipse[1],covEllipse[2],covEllipse[3],
                    fig[1],fig[2],fig[3],fig[4],truePath];
PSTeX[estimateFig, "estimatedPaths"]
)
.
PSTeX::file: Graphics being processed (without prolog) to file
    "measuredData.ps".

PSTeX::file: Graphics being processed (without prolog) to file
    "estimatedPaths.ps".

Out[10]= -Graphics-
```

From note 3 in Section 5, a more efficient algorithm for computing Λ, Σ, λ, and σ is as follows. As a check we print the differences between these results and the previously computed ones.

```
(
KSigma[0] = Sigma[0];
Ksigma[0] = sigma[0];
Do [(
   (* Lambda and lambda are computed as before *)
   Lambda[k] = Gamma[t[k],t[k-1]] . Q . Transpose[Gamma[t[k],t[k-1]]]
               + Phi[t[k],t[k-1]] . KSigma[k-1] . Transpose[Phi[t[k],t[k-1]]];
   lambda[k] = Phi[t[k],t[k-1]] . Ksigma[k-1] + u[t[k],t[k-1]];
   (* compute Sigma and sigma using the Kalman gain matrix *)
```

```
   KalmanGain[k] = Lambda[k] . Transpose[H] . Inverse[
                                    H . Lambda[k] . Transpose[H] + R];
   KSigma[k] = (IdentityMatrix[6] - KalmanGain[k] . H) . Lambda[k];
   Ksigma[k] = lambda[k] + KalmanGain[k] . (z[k] - H . lambda[k]);
   Print[" "]; Print["Differences for k=",k];
   Print[MatrixForm[N[Sigma[k]-KSigma[k]]],"   ", N[sigma[k]-Ksigma[k]]];
   ),{k,1,3}];
)
.
Differences for k=1
0   0   0   0   0   0   {0., 0., 0., 0., 0., 0.}

0   0   0   0   0   0

0   0   0   0   0   0

0   0   0   0   0   0

0   0   0   0   0   0

0   0   0   0   0   0

Differences for k=2
0   0   0   0   0   0   {0., 0., 0., 0., 0., 0.}

0   0   0   0   0   0

0   0   0   0   0   0

0   0   0   0   0   0

0   0   0   0   0   0

0   0   0   0   0   0

Differences for k=3
0   0   0   0   0   0   {0., 0., 0., 0., 0., 0.}

0   0   0   0   0   0

0   0   0   0   0   0

0   0   0   0   0   0

0   0   0   0   0   0

0   0   0   0   0   0
```

As a final check on the equations, we now generate some random vectors and matrices and check Theorem 2. We use matrices of integers to get exact arithmetic, and compute the ratio of the r.h.s. and l.h.s. of Theorem 2.

```
(
RandomMatrix[r_,s_] := Table[Random[Integer,{-100,100}],{i,1,r},{j,1,s}];
RandomVector[r_] := Table[Random[Integer,{-100,100}],{i,1,r}];
RandomPosDef[r_] := ( tmp=RandomMatrix[r,r]; tmp=tmp . Transpose[tmp];
                      If[Det[tmp]>0,tmp,RandomPosDef[r]] );
Do[(
   r = Random[Integer,{1,6}];
   s = Random[Integer,{1,6}];
   a = RandomVector[r];
   A = RandomPosDef[r];
   b = RandomVector[s];
   B = RandomPosDef[s];
   Q = RandomMatrix[r,s];
   x = RandomVector[s];
   Print[Gaussian[r,a,A,Q.x] Gaussian[s,b,B,x] /
     (Gaussian[s,cFun[Q,a,A,b,B],CFun[Q,A,B],x] Gaussian[r,a,A+Q.B.Transpose[Q],Q.b])];
   ),{test,1,10}]
)
.
1
1
1
1
1
1
1
1
1
1
```