Kalman filter QoS of WebRTC

1、理论推导

1.1 符号表示

X_bar：向量 X；

X_hat：X 的估计值；

X(i)：向量 X 的第 i 个分量；

[x y z]：包含元素 xyz 的行向量；

X_bar^T：向量 X_bar 的转置向量；

E{X}：随机变量 X 的期望；

1.2 推导过程

d(i) = t(i) – t(i-1) – (T(i) – T(i-1))

t(i)是当前数据包组 i 最后一个数据包的到达时间，T(i)是当前数据包 i 组最后一个数据包的发送时间，d(i)表示当前数据包组从发送到接收所消耗的时间间隔。如果 d(i) > 0 表示数据包组 i 相对于数据包组 i-1 在路上花费的时间更长，预示着有网络拥塞的可能。

d(i) = dL(i) / C(i) + m(i) + v(i)

dL(i)表示相邻数据包组的长度差，C(i)表示信道容量，m(i)表示排队时延，v(i)零均值高斯白噪声。其中[1/C(i) m(i)]是我们要求的值。这个式子也是接收端 kalman 滤波的理论基础。

推导：

定义：theta_bar(i) = [1/C(i)   m(i)]^T 为 i 时刻状态列向量；

h_bar(i) = [dL(i)   1]^T 为数据包长度差列向量；

theta_bar(i+1) = theta_bar(i) + u_bar(i); u_bar(i)表示零均值高斯平稳过程；

Q(i) = E{u_bar(i) * u_bar(i)^T} 表示 u_bar(i)的协方差；

diag(Q(i)) = [10^-13 10^-3]^T

估计过程：

theta_hat(i) = [1/C_hat(i)   m_hat(i)]^T；   目标估计值

z(i) = d(i) – h_bar(i)^T * theta_hat(i-1)；   计算残差

上述式子的意义：用上一时刻估计值估算本时刻的时间消耗：

d_hatL(i)/C + m_hat(i-1)

然后用当前观测值 d(i)和估算值求出残差 z(i)。

theta_hat(i) = theta_hat(i-1) + z(i) * k_bar(i)；目标估计值迭代过程；

i 时刻状态和 i-1 时刻的迭代关系：i-1 时刻的状态＋i 时刻的残差和 i 时刻的

kalman 增益的乘积，其中 i 时刻的 kalman 增益计算公式如下：

k_bar(i) = ((E(i-1) + Q(i)) * h_bar(i)) / (var_v_hat(i) + h_bar(i)^T * (E(i-1) + Q(i)) * h_bar(i))

E(i) = (I – k_bar(i) * h_bar(i)^T) * (E(i-1) + Q(i))

其中 I 是 2*2 单位阵。

var_v_hat(i) = max(beta * var_v_hat(i-1) + (1 – beta) * z(i)^2, 1)，

上述的方差使用指数平均滤波估算得出。

beta = (1 – chi)^(30/(1000 * f_max))

f_max = max{1/(t(j) – T(j-1))} for j in i-K+1, ... , i；过去 K 个数据包组中的最大码率； chi 是范围在[0.1 0.001]之内的过滤系数。

如果 z(i) > 3 * sqrt(var_v_hat(i))，那么将用后者取代 z(i)计算 theta_hat(i)。

2 代码注释

初始化过程：

```
OveruseEstimator::OveruseEstimator(const
OverUseDetectorOptions& options)
    : options_(options),
      num_of_deltas_(0),
      slope_(options_.initial_slope), // 1/C = (8.0 / 512.0)
      offset_(options_.initial_offset),  // m(i)初始化为0。
      prev_offset_(options_.initial_offset), // m(i-1)初始化为0。
      E_(),  // 元素为[100, 1e-1]的对角阵；
      process_noise_(), // [1e-13, 1e-3]的过程噪声向量；
      avg_noise_(options_.initial_avg_noise),  // 零均值噪声；
      var_noise_(options_.initial_var_noise),  // 方差＝50
      ts_delta_hist_() {
  memcpy(E_, options_.initial_e, sizeof(E_));
  memcpy(process_noise_, options_.initial_process_noise,
         sizeof(process_noise_));
}
void OveruseEstimator::Update(int64_t t_delta,
                              double ts_delta,
                              int size_delta,
                              BandwidthUsage current_hypothesis) {
```

```
  const double min_frame_period =
UpdateMinFramePeriod(ts_delta);
  const double t_ts_delta = t_delta - ts_delta; //计算d(i)
  double fs_delta = size_delta;  //计算dL(i)

  ++num_of_deltas_;
  if (num_of_deltas_ > kDeltaCounterMax) {
    num_of_deltas_ = kDeltaCounterMax;
  }

  // Update the Kalman filter. 更新对角线元素加上随即噪声。
  E_[0][0] += process_noise_[0];
  E_[1][1] += process_noise_[1];

  if ((current_hypothesis == kBwOverusing && offset_ < prev_offset_)
|| (current_hypothesis == kBwUnderusing && offset_ > prev_offset_))
{
    // for what?
    E_[1][1] += 10 * process_noise_[1];
  }

  const double h[2] = {fs_delta, 1.0};  //定义h_bar(i)
  // Eh = h_bar * E_;  // 1 * 2 行向量;
  const double Eh[2] = {E_[0][0]*h[0] + E_[0][1]*h[1],
                        E_[1][0]*h[0] + E_[1][1]*h[1]};  //计算期望？

  // z(i) = d(i) – h_bar(i)^T * theta_hat(i-1)的计算残差过程，
  // 其中[slope_  offset_] 为theta_hat(i-1), [fs_delata 1]为h_bar(i);
  const double residual = t_ts_delta - slope_*h[0] - offset_;

  const bool in_stable_state = (current_hypothesis == kBwNormal);
  // 残差上限不能超过噪声方差平方根的3倍，限制残差波动范围。
  const double max_residual = 3.0 * sqrt(var_noise_);
```

```
  // We try to filter out very late frames. For instance periodic
 // key frames doesn't fit the Gaussian model well.
  // 更新噪声估计，残差被限制在3.0 * sqrt(var_noise_)之内；
  if (fabs(residual) < max_residual) {
    UpdateNoiseEstimate(residual, min_frame_period,
                        in_stable_state);
  } else {
    UpdateNoiseEstimate(residual < 0 ? -max_residual :
                        max_residual,  min_frame_period,
                        in_stable_state);
  }

  const double denom = var_noise_ + h[0]*Eh[0] + h[1]*Eh[1];

  const double K[2] = {Eh[0] / denom, Eh[1] / denom}; // k_bar

  const double IKh[2][2] = {{1.0 - K[0]*h[0], -K[0]*h[1]},
                        {-K[1]*h[0], 1.0 - K[1]*h[1]}};
  const double e00 = E_[0][0];
  const double e01 = E_[0][1];

  // Update state.
  E_[0][0] = e00 * IKh[0][0] + E_[1][0] * IKh[0][1];
  E_[0][1] = e01 * IKh[0][0] + E_[1][1] * IKh[0][1];
  E_[1][0] = e00 * IKh[1][0] + E_[1][0] * IKh[1][1];
  E_[1][1] = e01 * IKh[1][0] + E_[1][1] * IKh[1][1];

  // The covariance matrix must be positive semi-definite.
  bool positive_semi_definite = E_[0][0] + E_[1][1] >= 0 &&
      E_[0][0] * E_[1][1] - E_[0][1] * E_[1][0] >= 0 && E_[0][0] >=
0;
  assert(positive_semi_definite);
  if (!positive_semi_definite) {
```

```
    LOG(LS_ERROR) << "The over-use estimator's covariance matrix"
    "is no longer semi-definite.";
  }

  // 得到估计结果: slope_为1/C, offset_为m(i)
  //  theta_hat(i) = theta_hat(i-1) + z(i) * k_bar(i);
  slope_ = slope_ + K[0] * residual;
  prev_offset_ = offset_;
  offset_ = offset_ + K[1] * residual;
}

double OveruseEstimator::UpdateMinFramePeriod(double
ts_delta) {
  double min_frame_period = ts_delta;
  if (ts_delta_hist_.size() >= kMinFramePeriodHistoryLength) {
    ts_delta_hist_.pop_front();
  }
  std::list<double>::iterator it = ts_delta_hist_.begin();
  for (; it != ts_delta_hist_.end(); it++) {
    min_frame_period = std::min(*it, min_frame_period);
  }
  ts_delta_hist_.push_back(ts_delta);
  return min_frame_period;
}

void OveruseEstimator::UpdateNoiseEstimate(double
residual, double ts_delta, bool stable_state) {
  if (!stable_state) {
    return;
  }
  // Faster filter during startup to faster adapt to the jitter level
  // of the network. |alpha| is tuned for 30 frames per second, but
  // is scaled according to |ts_delta|.
```

```
  double alpha = 0.01;
  if (num_of_deltas_ > 10*30) {
    alpha = 0.002;
  }
  // Only update the noise estimate if we're not over-using.
 // |beta| is a function of alpha and the time delta since the
 // previous update.
  const double beta = pow(1 - alpha, ts_delta * 30.0 / 1000.0);
  avg_noise_ = beta * avg_noise_ + (1 - beta) * residual;
  var_noise_ = beta * var_noise_
               + (1 - beta) * (avg_noise_ - residual)
                 * (avg_noise_ - residual);
  if (var_noise_ < 1) {
    var_noise_ = 1;
  }
}
```