2022赛季视觉部第六次培训——卡尔曼滤波

> 机械是肉体，电控是大脑，视觉是灵魂

为什么要使用卡尔曼滤波

在比赛中的自瞄系统中，如果想要准确预测地方车辆的运动轨迹，就必须要精确地求出对方车辆的速度。而卡尔曼滤波可以高效、较准确，实时地根据一系列带有时间戳的坐标点推算出敌方车辆的速度。

为了方便研究，我们可以先把这个二维空间内的速度求解问题转换到一维。
假设我们有一系列处在函数$f(x) = kx + b$上的一系列离散点，我们可以通过哪些方法通过这些离散点推算出$f(x)$的斜率k。

对于这一问题，最简单的方法是通过差分的方式求解速度。即通过公式：

$$v = \frac{x2 - x1}{t2 - t1}$$

这是最朴素的思路，但是问题在于：

1. 受噪声干扰大
2. 求出的速度与实际速度有一定延迟（由于求解的是平均速度，因此这一方法求出的速度事实上是时刻$\frac{t2+t1}{2}$的瞬时速度）
3. 输出的速度不连续，在不断跳跃。

下面这段代码是这一思路的简单实现：

```
#include <iostream>
#include <cstdio>
#include <string>
#include <vector>

#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <Eigen/Dense>
```

```cpp
#include <opencv2/core/eigen.hpp>

using namespace std;
using namespace cv;
using namespace Eigen;

int main()
{
    // generate data with noise
    const int N = 20;
    const double K = 2.5;
    Matrix<double, 1, N> noise = Matrix<double, 1, N>::Random();
    Matrix<double, 1, N> data = Matrix<double, 1, N>::LinSpaced(0, K
* (N - 1));
    data += noise;
    std::cout << data << std::endl;


    // calculate speed
    double speed_sum = 0;
    for (int i = 1; i < N; i++)
    {
        double speed = (data[i] - data[i - 1]);
        speed_sum += speed;
        std::cout << "step " << i << ": " << speed << std::endl;
    }
    std::cout << "final speed: " << speed_sum / (N - 1) << std::endl;
    return 0;
}
```

我们可以看一下他的效果：
这是输入：

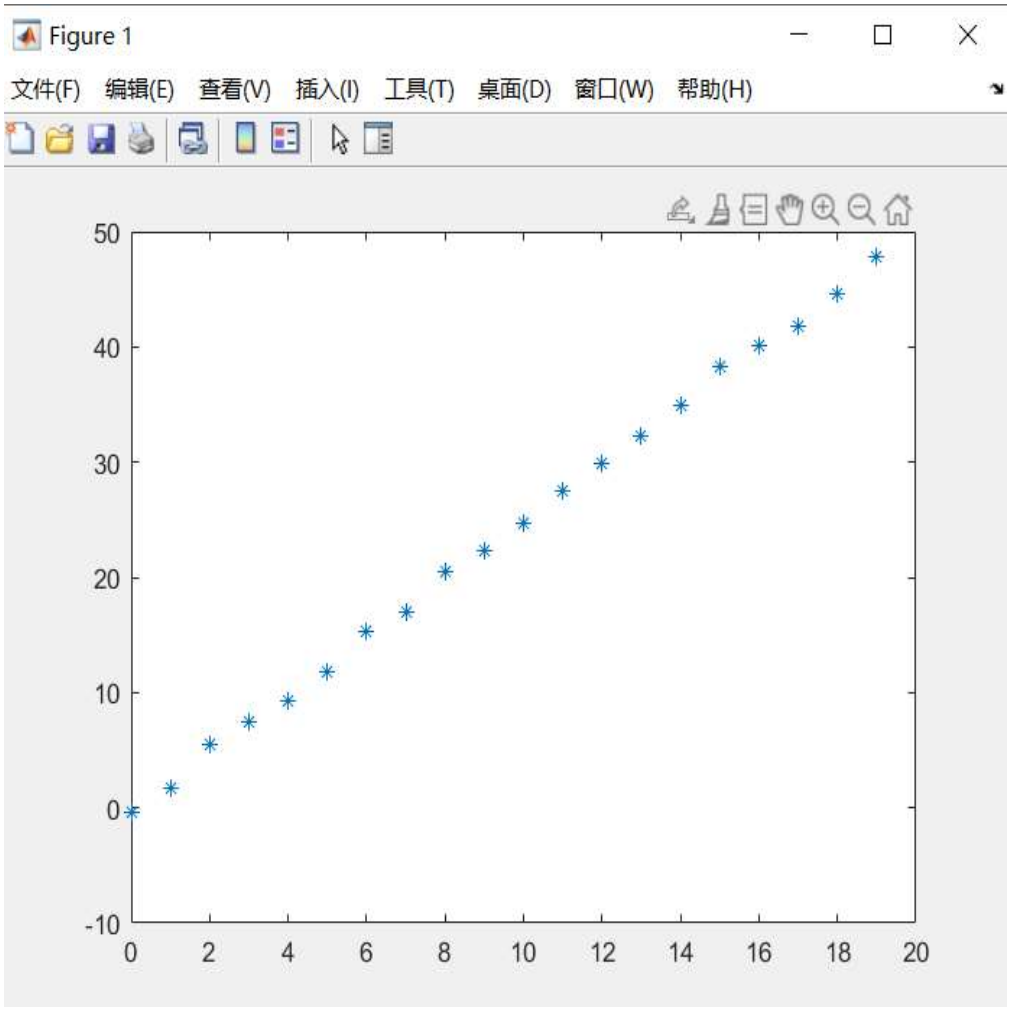


Figure 1
文件(F) 编辑(E) 查看(V) 插入(I) 工具(T) 桌面(D) 窗口(W) 帮助(H)
50
40
30
20
10
0
-10
0 2 4 6 8 10 12 14 16 18 20

这是输出：

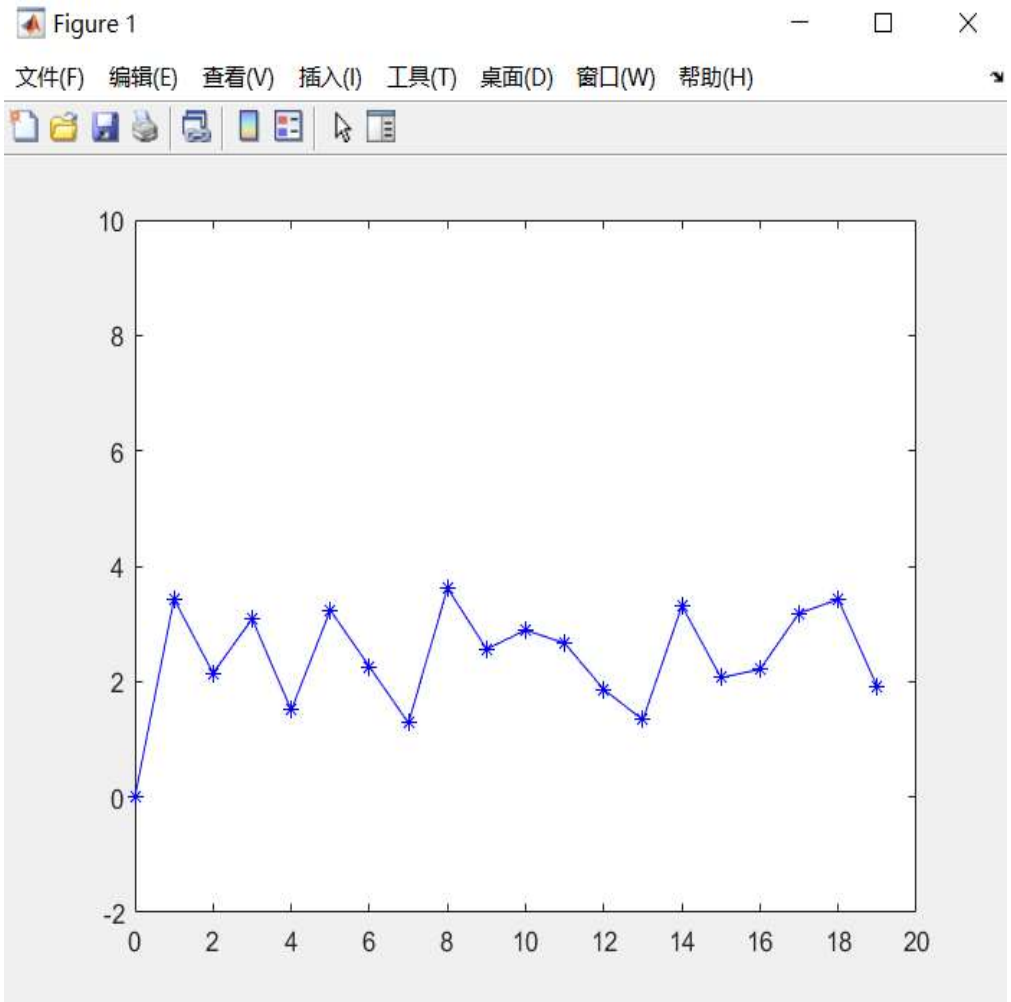


Figure 1
文件(F) 编辑(E) 查看(V) 插入(I) 工具(T) 桌面(D) 窗口(W) 帮助(H)
10
8
6
4
2
0
-2
0 2 4 6 8 10 12 14 16 18 20

由于对每一帧求出的速度求了平均值，他得出了还算不错的离线结果（速度）：2.53

但是在这一例子中，这种朴素的求解方法的问题也基本展现了出来：
1、数据抖动严重
2、无法事实得到可以接受的瞬时速度

因此，我们需要更优秀的算法求出运动物体的速度，他要有下面的特点：
1、实时计算速度
2、输出结果的波动较小
3、抗噪声能力强

目前，常用于计算速度的方法有：卡尔曼滤波和非线性优化两种。

卡尔曼滤波

直观地理解卡尔曼滤波

下面举一个常用的卡尔曼滤波的例子：
假设有一辆自动驾驶的小车想要通过一个隧道，他想要知道自己每一时刻在隧道中的位置，但是隧道中没有了GPS，因此你没有可以直接获得位置信息的手段。这种情况下，应该如何得到小车在隧道中的问题。

在试图解决问题之前，先看看我们有哪些可以用来计算小车位置的数据：
1、轮胎编码器
2、惯性测量单元
3、供给小车前进的燃料稳定（可以理想为小车在做匀速直线运动）

把这些信息进行分类，我们大致有两种信息：

1. 测量量
    - 编码器
    - 惯性测量单元
2. 预测量
    - 匀速直线模型

这里引入一点简单的数学：

1. 通过**预测量**，我们可以得到如下的方程

$$x_2 = x_1 + v_1 \Delta t$$

$$v_2 = v_1$$

2. 通过**观测量**，我们可以得到如下的方程

$$x_2 = x_1 + \Delta x$$

$$v_2 = v_1 + a\Delta t$$

$$x_2 = v_1 \Delta t + \frac{1}{2} a t^2$$

其中x_1, v_1为上一时刻小车的位置和速度，x_2, v_2为这一时刻小车的位置和速度。
Δt为前后两时刻之间的时间差，Δx为编码器得到的汽车位移，a为惯性测量单元得到的小车加速度。

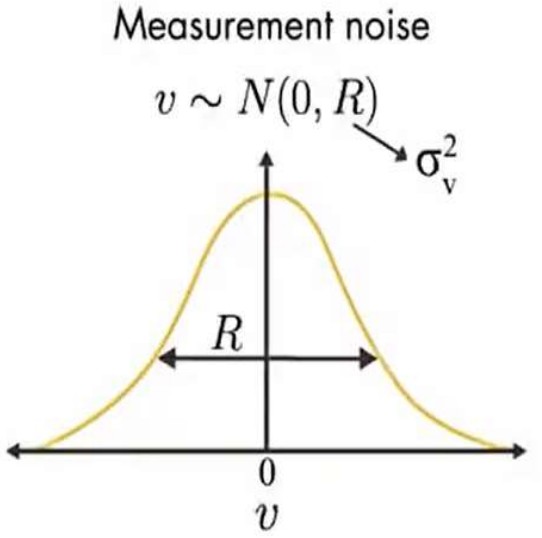
通过这些变量，公式，以及得到的传感器数据，观察上面的5个算式，我们可以用3种不同的算法得到小车的位置x，2种不同的方法得到小车的速度。

我们的目标是求出小车的位置，因此我们来看一下位置的三种求解方法：
第一个**预测的方法**利用牛顿运动定律约束前后之前位置的关系
后两个**观测的方法**利用传感器的数据间接推算小车位置
显然，这三个计算方法都不可避免的**存在误差**。

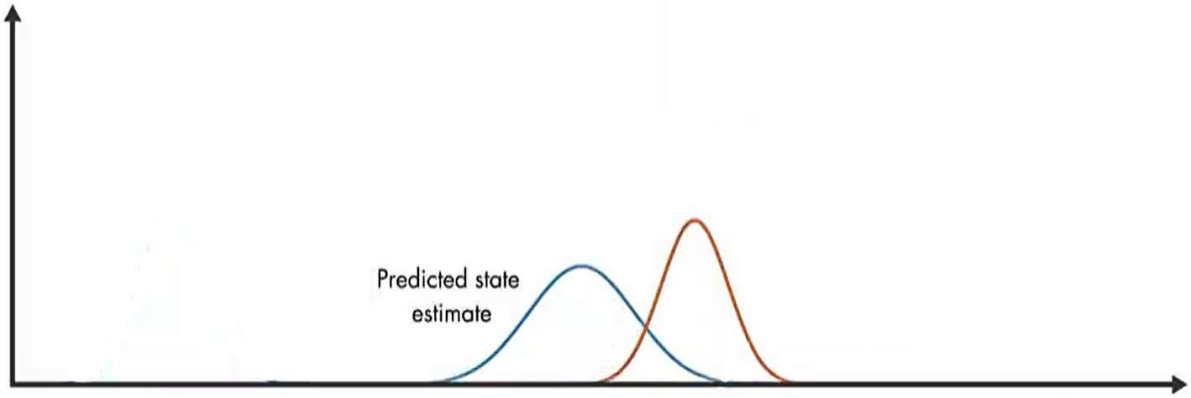
在我校的实验课程中，你应该了解过我们在测量和计算时会受到两种误差的干扰：

- **随机误差**
  由于在测定过程中一系列有关因素微小的随机波动而形成的具有相互抵偿性的误差。
  一般来说呈现正态分布
- **系统误差**
  一种非随机性误差。
  如违反随机原则的偏向性误差，在抽样中由登记记录造成的误差等。
  他会使得数据产生像一个方向的偏移

这里我们不妨假设我们的计算模型绝对正确，对数据的使用过程中也没有出现原理上的问题，也就是说**不存在系统误差**。
那么再考虑上述的三个计算结果，由于每个结果都存在**随机误差**，因此他们的实际分布应该是一个概率上的**正态分布**。

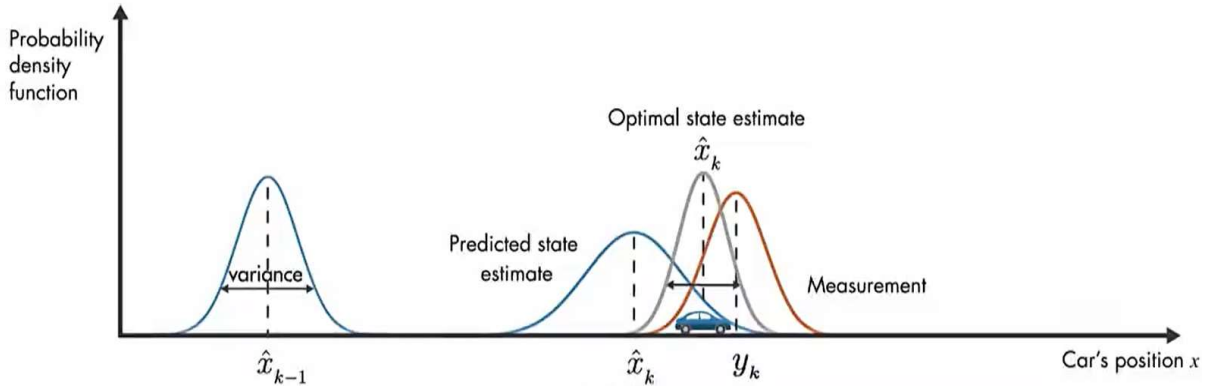


如果呈现在坐标轴上，他们应该大致分布成这样：



可以看到，这会是多个独立的正态分布曲线，每个曲线代表一种方式计算出的小车的位置的分布区间。

而卡尔曼滤波所做的，就是推算出一个合适的每个计算结果的权重，并以此预测物体的未来状态



如图，他通过预测量和观测量一同推算出小车最可能出现在的最终位置。

简单来讲，卡尔曼滤波器就是根据上一时刻的状态，预测当前时刻的状态，将预测的状态与当前时刻的测量值进行加权，加权后的结果才认为是当前的实际状态，而不是仅仅听信当前的测量值。

这一部分这篇教程讲的很清楚，可以参考

卡尔曼滤波的相关理论

发表卡尔曼滤波的论文名为: A New Approach to Linear Filtering and Prediction Problems，有能力的同学可以试着阅读原版论文。

马尔科夫性：简单的一阶马氏性认为，k时刻状态只与$k-1$时刻有关，而与之前的无关。
卡尔曼滤波就是基于马尔科夫性的滤波器方法。
(另一种方法是依然考虑k时刻状态与之前所有状态的关系，这就是非线性优化为主体的优化框架)

卡尔曼滤波的完整推导十分复杂，故这里直接跳过推倒部分。想要使用卡尔曼滤波，你只需要知道下面一组公式。

Prediction
$x^{'} = Ax + u$
$P^{'} = APA^{T} + R$
Measurement update
$y = z - Cx^{'}$
$S = CPC^{T} + Q$
$K = PC^{T}S^{-1}$
$x = x^{'} + Ky$
$P = (I - KC)P$

下面我们来解释这组公式：

先说明一下公式中的几个变量的意义：

- x：$k-1$时刻的滤波值，也就是$k-1$时刻的状态。
- $x^{'}$： x的先验估计
- A：状态转移方程，即x_{k-1}与x_k之间的转移关系
- P：预测误差协方差
- R：预测过程噪声偏差的方差
- z：k时刻的观测数据
- C：观测矩阵，观测量与预测量之间的关系
- Q：测量过程噪声偏差的方差
- K：卡尔曼增益

正如前文中所说，卡尔曼滤波对预测量和观测量进行加权平均。这一性质在上述公式中具体表现为：整个运算分为预测量更新和观测量更新两个部分完成。

我们通过输入上一时刻的状态x_{k-1}和这一时刻的观测z，通过给定的预测方差与观测方差，更新卡尔曼增益，进而得到估算的这一时刻的状态x。这就是这一组公式的内在含义。

卡尔曼滤波的实现

还是用之前的$f(x) = kx + b$的例子为例：

$$X_k = \begin{bmatrix} x_k \\ k \end{bmatrix}$$

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

那么

$$X_k = AX_{k-1}$$

这就是我们在这一例子中的状态转移矩阵。

而

$$Z_k = \begin{bmatrix} z \end{bmatrix}$$

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

$$Z_k = CX_k$$

这就是我们的观测矩阵。

下面我们使用Eigen来辅助实现卡尔曼滤波：

```
#include <iostream>
#include <cstdio>
#include <string>
#include <vector>
#include <ctime>

#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <Eigen/Dense>
#include <opencv2/core/eigen.hpp>

using namespace std;
using namespace cv;
using namespace Eigen;

int main()
{
    srand((unsigned int) time(NULL));

    // generate data with noise
    const int N = 20;
    const double k = 2.5;
    Matrix<double, 1, N> noise = Matrix<double, 1, N>::Random();
    Matrix<double, 1, N> data = Matrix<double, 1, N>::LinSpaced(0, k
* (N - 1));
    data += noise;
    std::cout << data << std::endl;


    // calculate speed
    const int Z_N = 1, X_N = 2;

    Matrix<double, X_N, 1> X;
    Matrix<double, X_N, X_N> A;
    Matrix<double, X_N, X_N> P;
```

```
    Matrix<double, X_N, X_N> R;
    Matrix<double, X_N, Z_N> K;
    Matrix<double, Z_N, X_N> C;
    Matrix<double, Z_N, Z_N> Q;

    X << data[0], 0;
    A << 1, 1, 0, 1;
    C << 1, 0;
    R << 2, 0, 0, 2;
    Q << 10;

    for (int i = 1; i < N; i++)
    {
        // 更新预测
        Matrix<double, X_N, 1> X_k = A * X;
        P = A * P * A.transpose() + R;

        // 更新观测
        K = P * C.transpose() * (C * P * C.transpose() + Q).inverse
();
        Matrix<double, Z_N, 1> Z{data[i]};
        X = X_k + K * (Z - C * X_k);
        P = (Matrix<double, X_N, X_N>::Identity() - K * C) * P;

        std::cout << "step " << i << ": " << X[1] << std::endl;
    }

    std:cout << "final speed: " << X[1] << std::endl;


    return 0;
}
```

我们同样看一下他的效果：
这是输入：

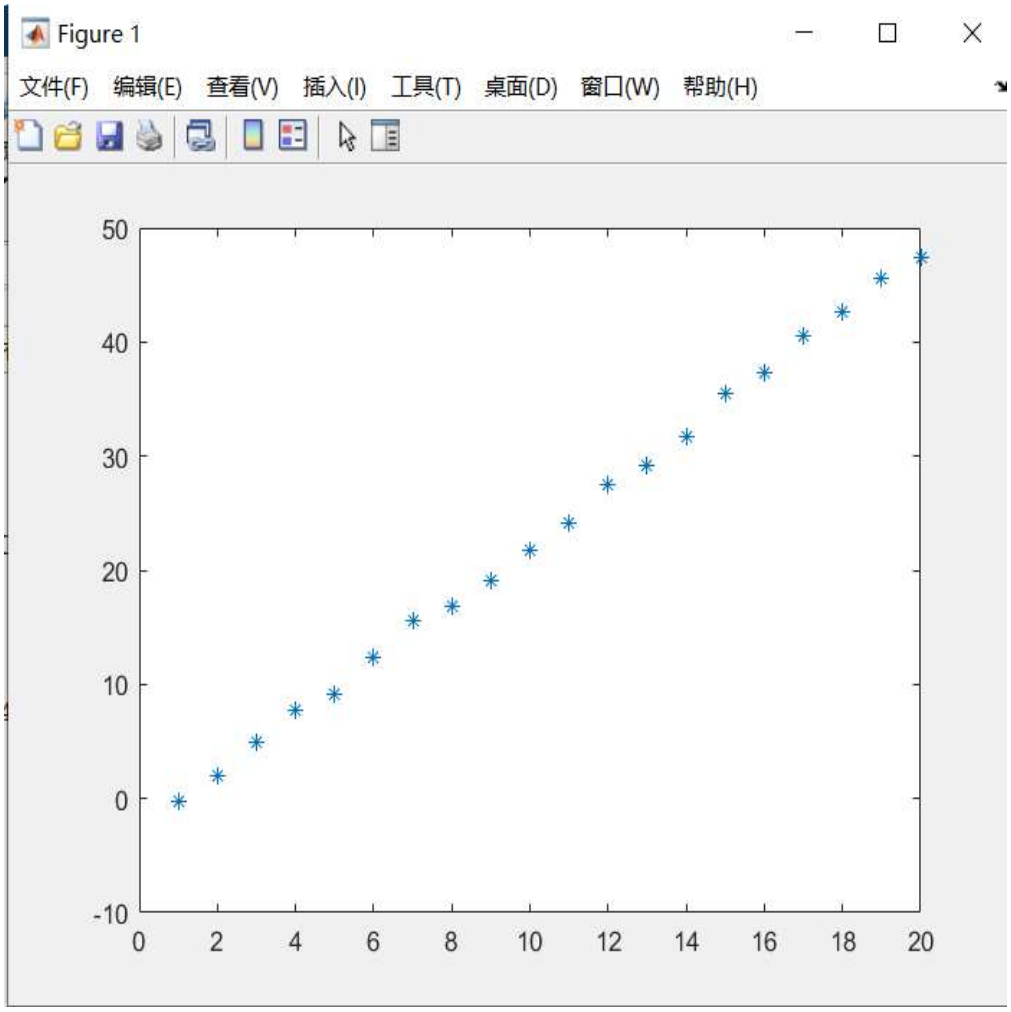
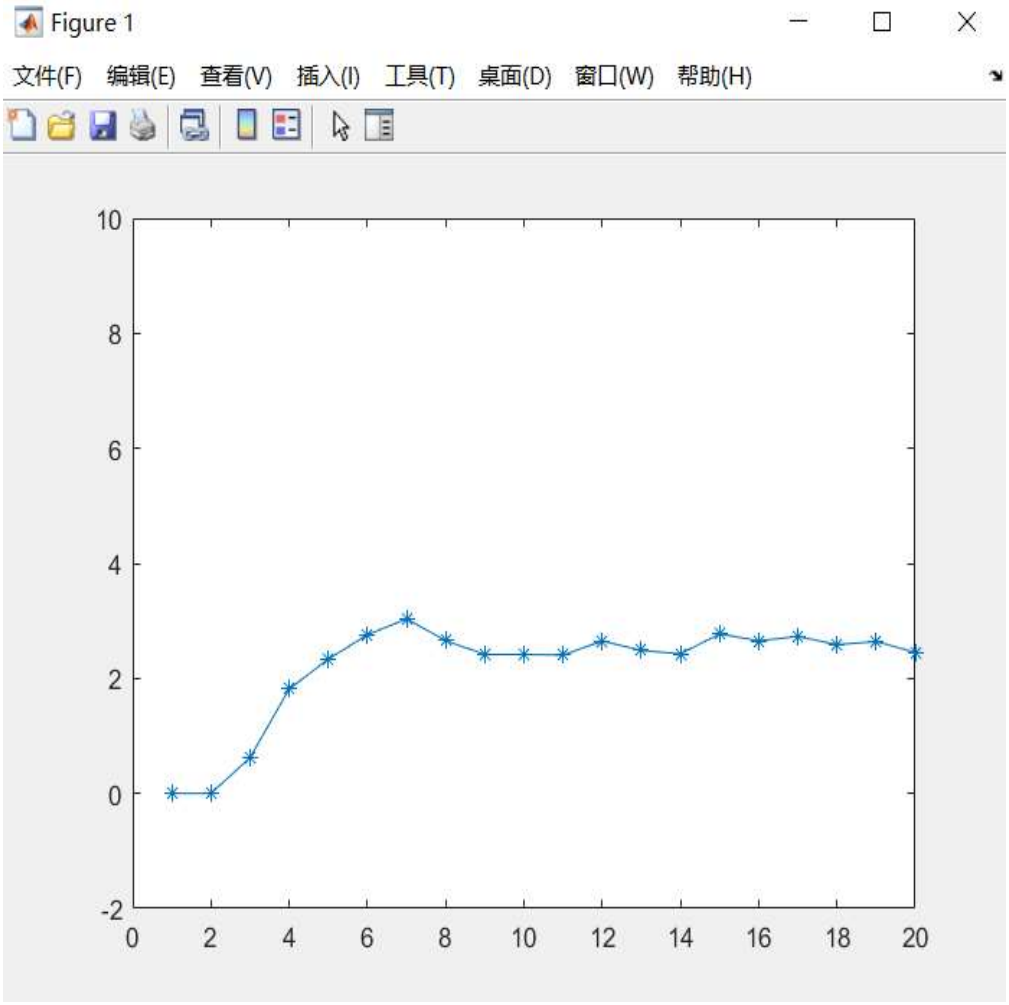


Figure 1
文件(F) 编辑(E) 查看(V) 插入(I) 工具(T) 桌面(D) 窗口(W) 帮助(H)
50
40
30
20
10
0
-10
0 2 4 6 8 10 12 14 16 18 20

这是输出：



本次测试的输出为：2.45391

由于没有对每一次的计算结果取平均，因此单单看最终的输出他似乎并没有明显优于差分法求解速度。
但是观察卡尔曼滤波的输出曲线，我们会发现

1. 他在逐渐逼近实际速度2.5，
2. 卡尔曼滤波的输出整体较平滑，并没有明显的跳变，噪声对他的扰动并不大

需要注意的是，程序中的R和Q分别是预测噪声协方差和测量噪声协方差，这两个参数是经验常数，需要滤波器使用者根据实际情况自己调整。
他们的物理意义为，协方差越大，代表计算结果误差越大。因此，如果认为一个预测量可能会有巨大的误差，那么就可以试着调大他的协方差。

一般来说:

- 如果想要增高一个预测量的实时性，那么可以减小他的协方差矩阵，但后果是这个变量的预测输出会变的更抖
- 如果想要提高一个预测量的平滑度，那么可以增大他的协方差矩阵，但后果是这个变量的实时性会变低

附录一中储存了一系列不同参数的测试结果。

卡尔曼滤波的不足

- 卡尔曼滤波有马尔可夫性，只考虑了上一个时间节点，而并没有考虑$0 \rightarrow k-1$项数据的特征
- 卡尔曼滤波在只适用于线性模型，而对于非线性模型并没有良好的效果

对于第一条不足，我们有对应的**优化类算法**解决
对于第二条不足，后来者对卡尔曼滤波进行了各种改良，创造了**EKF(拓展卡尔曼滤波)**，**UKF(无迹卡尔曼滤波)**等不同的卡尔曼滤波算法。

拓展卡尔曼滤波

为什么需要EKF算法

前面说到，由于**KF(卡尔曼滤波)**对非线性模型的表现并不良好，因此**EKF(拓展卡尔曼滤波算法)**应运而生。
下面仍然用一组例子来说明为什么需要拓展卡尔曼滤波算法。
我们把之前例子中的函数$f(x)$换成$f(x)=kx^2$，这是一个非常简单的非线性模型，但可以看到，这里前后两项之间的转移方程已经变成了

$$x_k = k(\sqrt{\frac{x_{k-1}}{k}}+1)^2$$

这样的一个非线性转移已经无法使用传统的卡尔曼滤波中的矩阵运算描述了。当然，我们仍然可以用一个线性模型去逼近它。

下面是新的数据生成代码：

```
    // generate data with noise
    const int N = 100;
    const double k = 1;
    Matrix<double, 1, N> noise = Matrix<double, 1, N>::Random();
    Matrix<double, 1, N> data = Matrix<double, 1, N>::LinSpaced(0, k
* (N - 1));
    data += noise;
    data = (data.array() * data.array()).matrix();
```

测试数据如图：

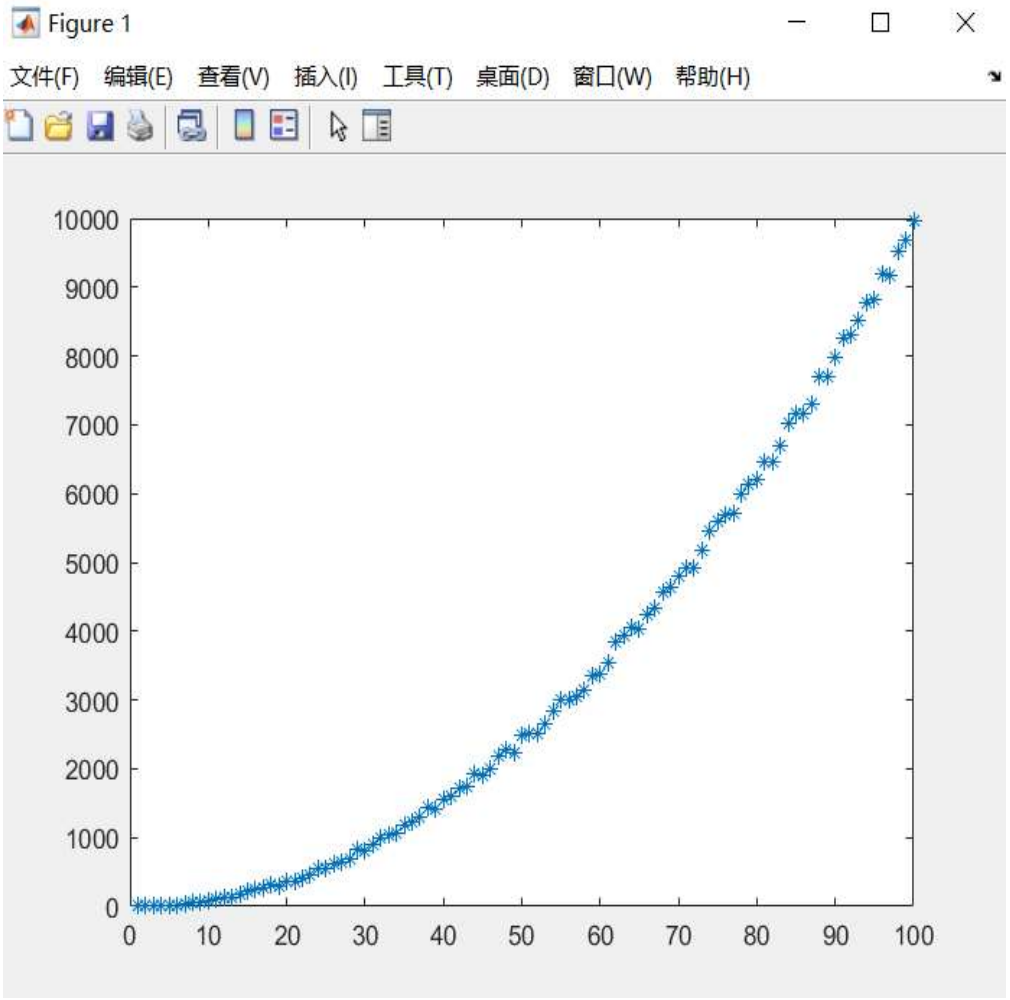
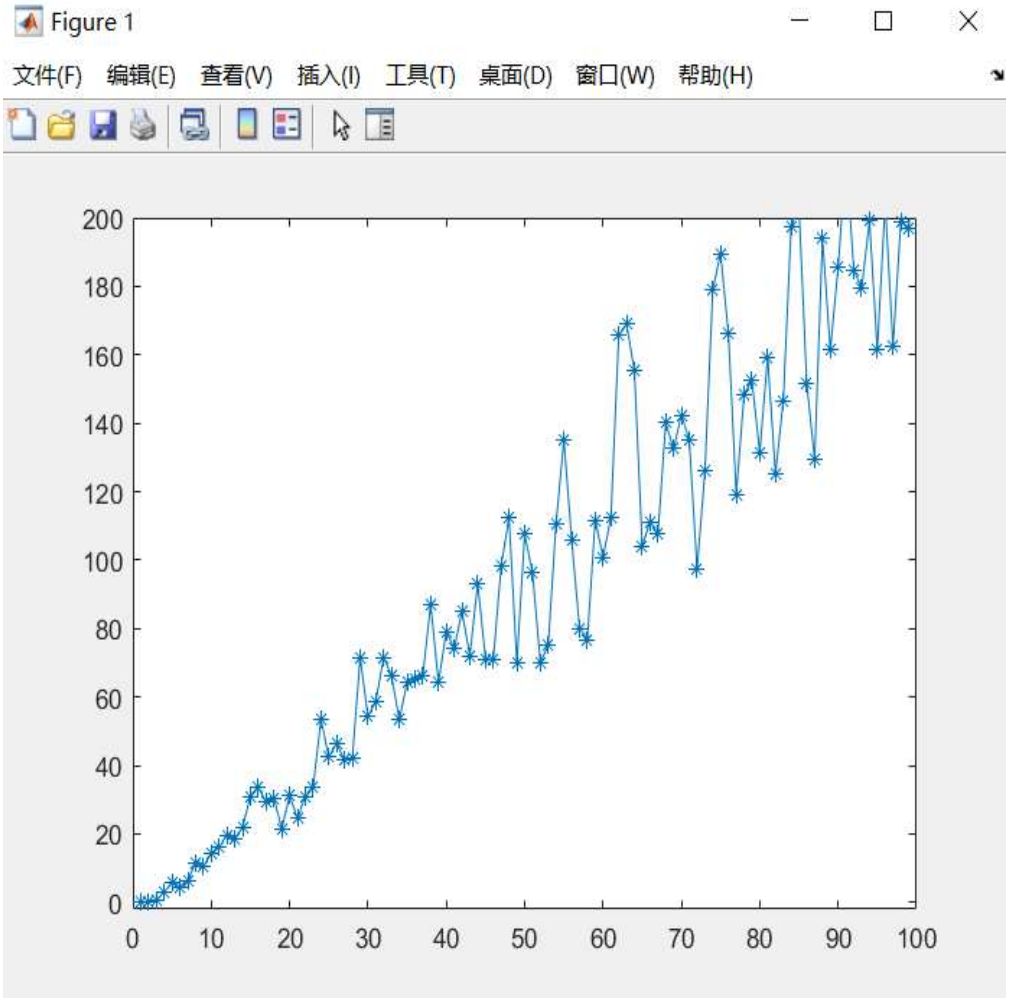


Figure 1
文件(F) 编辑(E) 查看(V) 插入(I) 工具(T) 桌面(D) 窗口(W) 帮助(H)
10000
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
0 10 20 30 40 50 60 70 80 90 100

输出结果如图：



可以看到数据受到的扰动极大，且滤出的速度反复在滞后和超前之间跳跃，这是因为滤波器的预测模型和实际模型不匹配导致的。

EKF算法做出的改变

非线性模型和线性模型最直接的区别在于：**状态转移方程**和**观测方程**由线性变为非线性。

这一特点在数学公式下可以表现得比较直接。
线性情况下，我们可以用**矩阵运算**直接表示出状态转移方程和观测方程：

$$X^{'} = AX + U$$

$$Z = CX + V$$

而在非线性情况下，我们失去了矩阵工具，因此只能通过函数来表示：

$$X^{'} = F(X) + U$$

$$Z = H(X) + V$$

而EKF的核心思想就在于，通过数学上的近似手段，把函数$F(x)$和函数$H(x)$转化为矩阵运算FX和HX。这部分近似的原理并不难理解，下面我们来引入一些简单的数学。

问题：对于一个函数$F(x)$，已知$F(x_0)$，如何近似求解$F(x_0+\Delta x)$。
我们对$F(x)$在x_0进行泰勒展开：

$$F(x)=F(x_0)+\frac{F^{'}(x_0)}{1!}(x-x_0)+\frac{F^{''}(x_0)}{2!}(x-x_0)^2+\frac{F^{(3)}(x)}{3!}(x-x_0)^3+\frac{F^{(n)}(\epsilon)}{n!}(x-x_0)^n$$

一般来说，在非线性度较低的情况下，近似只需取一阶泰勒展开即可。
因此，我们可以得到这样的近似结果：

$$F(x_0+\Delta x)\approx F(x_0)+F^{'}(x_0)\Delta x$$

我们可以用这样的近似方式，将状态转移方程和观测方程近似为状态转移矩阵和观测矩阵。
设状态转移矩阵为F，那么F_{ij}的值就是第i个变量的状态转移方程F_i对第j个变量的偏导。

一般来说，假设我们的状态向量为：

$$X=\begin{bmatrix}x_1\\x_2\\\cdots\\x_n\end{bmatrix}$$

那么我们会有这样的转移方程组：

$$x_i^{'}=F_{x_i}(x_1,x_2,\ldots,x_n)$$

通过一阶泰勒展开近似，可以得到这样的近似转移方程组：

$$x_i^{'}=\sum_j\frac{dF_i}{dx_j}x_j$$

如果这样的公式描述不够直观，我们可以用一个例子更清晰地描述。
还是以函数$f(x)=kx^2$为例：
对于他，我们可以列写状态向量

$$X=\begin{bmatrix}f(x)\\x\\k\end{bmatrix}$$

X_i的转移方程为

$$f^*(x)=k(x+\Delta x)^2$$

$$x^*=x+\Delta x$$

$$k^*=k$$

对他们使用一阶泰勒展开近似：

$$f^*(x)=0+(2kx+2k)x+(x+\Delta x)^2k$$

$$x^* = 0 + 1 * x + 0$$

$$k^* = 0 + 0 + 1 * k$$

那么我们就可以用一个矩阵来表示这个转移了：

$$F = \begin{bmatrix} 0 & (2kx + 2k) & (x + \Delta x)^2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$X_k = FX_{k-1}$$

EKF算法的实现

可以看到在EKF算法的过程中，我们需要对变量求导得到矩阵F和矩阵H。
程序自动求导实现起来并不容易，因此我们一般使用 `ceres` 工具库的 `Jet` 类型辅助完成求导的工作。
`ceres` 库的安装可以参考 `ceres` 官网提供的教程：http://www.ceres-solver.org/installation.html

由于唐欣阳~~学姐~~实现的EKF滤波器过于优雅，笔者没有信心写出更好的版本，因此我们以他的代码为示例代码：

```
//
// Created by xinyang on 2021/3/15.
//

#include <ceres/jet.h>
#include <Eigen/Dense>

template<int N_X, int N_Y>
class AdaptiveEKF {
    using MatrixXX = Eigen::Matrix<double, N_X, N_X>;
    using MatrixYX = Eigen::Matrix<double, N_Y, N_X>;
    using MatrixXY = Eigen::Matrix<double, N_X, N_Y>;
    using MatrixYY = Eigen::Matrix<double, N_Y, N_Y>;
    using VectorX = Eigen::Matrix<double, N_X, 1>;
    using VectorY = Eigen::Matrix<double, N_Y, 1>;

public:
    explicit AdaptiveEKF(const VectorX &X0 = VectorX::Zero())
            : Xe(X0), P(MatrixXX::Identity()), Q(MatrixXX::Identity
()), R(MatrixYY::Identity()) {}

    template<class Func>
    VectorX predict(Func &&func) {
        ceres::Jet<double, N_X> Xe_auto_jet[N_X];
```

```
            for (int i = 0; i < N_X; i++) {
                Xe_auto_jet[i].a = Xe[i];
                Xe_auto_jet[i].v[i] = 1;
            }
            ceres::Jet<double, N_X> Xp_auto_jet[N_X];
            func(Xe_auto_jet, Xp_auto_jet);
            for (int i = 0; i < N_X; i++) {
                Xp[i] = Xp_auto_jet[i].a;
                F.block(i, 0, 1, N_X) = Xp_auto_jet[i].v.transpose();
            }
            P = F * P * F.transpose() + Q;
            return Xp;
        }

        template<class Func>
        VectorX update(Func &&func, const VectorY &Y) {
            ceres::Jet<double, N_X> Xp_auto_jet[N_X];
            for (int i = 0; i < N_X; i++) {
                Xp_auto_jet[i].a = Xp[i];
                Xp_auto_jet[i].v[i] = 1;
            }
            ceres::Jet<double, N_X> Yp_auto_jet[N_Y];
            func(Xp_auto_jet, Yp_auto_jet);
            for (int i = 0; i < N_Y; i++) {
                Yp[i] = Yp_auto_jet[i].a;
                H.block(i, 0, 1, N_X) = Yp_auto_jet[i].v.transpose();
            }
            K = P * H.transpose() * (H * P * H.transpose() + R).inverse
    ();
            Xe = Xp + K * (Y - Yp);
            P = (MatrixXX::Identity() - K * H) * P;
            return Xe;
        }

        VectorX Xe;      // 估计状态变量
        VectorX Xp;      // 预测状态变量
        MatrixXX F;      // 预测雅克比
        MatrixYX H;      // 观测雅克比
        MatrixXX P;      // 状态协方差      *
        MatrixXX Q;      // 预测过程协方差   *    -
        MatrixYY R;      // 观测过程协方差   *
        MatrixXY K;      // 卡尔曼增益
        VectorY Yp;      // 预测观测量
    };
```

一个小栗子

回到我们之前的例子，如何滤除用函数$f(x) = kx^2$生成的一系列带有噪声的点的增长速度。下面这段程序利用唐欣阳的EKF类实现了这一功能：

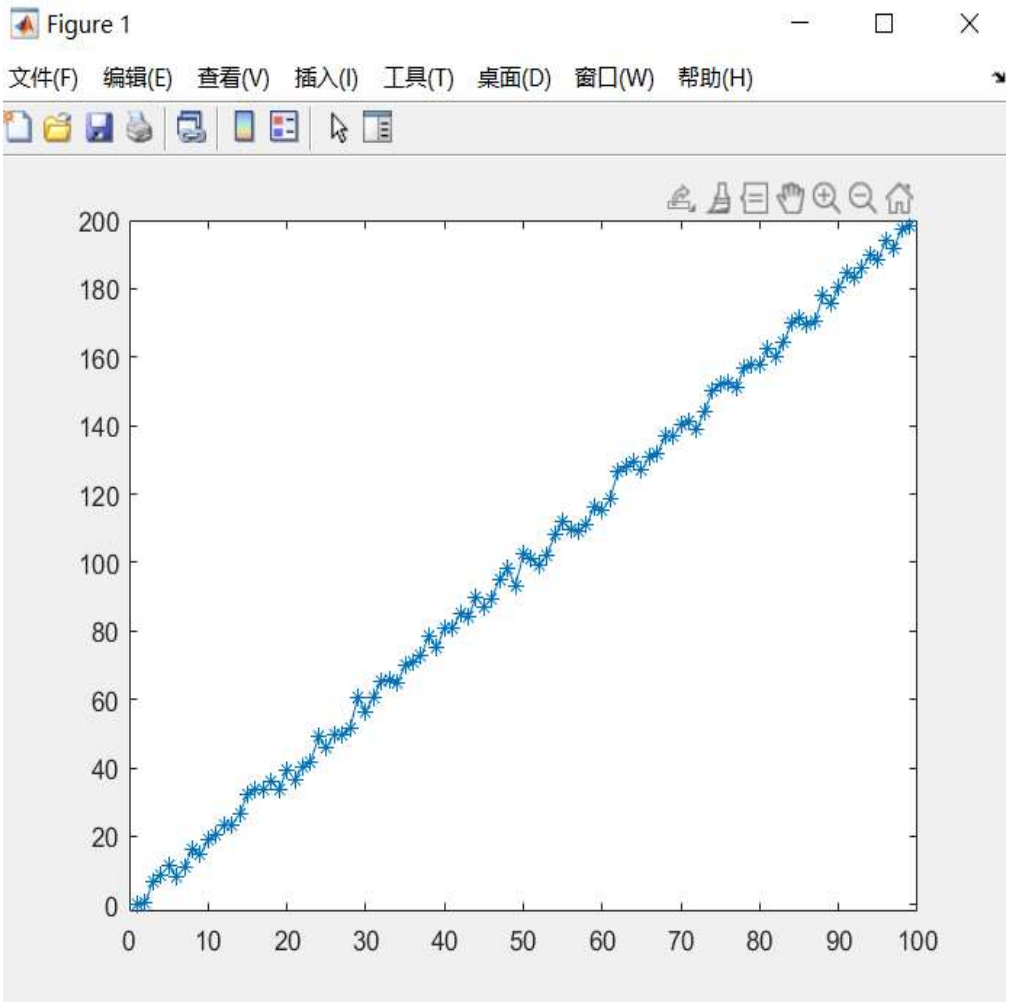
```
#include <iostream>
#include <cstdio>
#include <string>
#include <vector>
#include <ctime>

#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <Eigen/Dense>
#include <opencv2/core/eigen.hpp>
#include <ceres/jet.h>

using namespace std;
using namespace cv;
using namespace Eigen;
using ceres::Jet;

// EKF类，作者：唐欣阳
//
// Created by xinyang on 2021/3/15.
//
template<int N_X, int N_Y>
class AdaptiveEKF {
    using MatrixXX = Eigen::Matrix<double, N_X, N_X>;
    using MatrixYX = Eigen::Matrix<double, N_Y, N_X>;
    using MatrixXY = Eigen::Matrix<double, N_X, N_Y>;
    using MatrixYY = Eigen::Matrix<double, N_Y, N_Y>;
    using VectorX = Eigen::Matrix<double, N_X, 1>;
    using VectorY = Eigen::Matrix<double, N_Y, 1>;

public:
    explicit AdaptiveEKF(const VectorX &X0 = VectorX::Zero())
            : Xe(X0), P(MatrixXX::Identity()), Q(MatrixXX::Identity
()), R(MatrixYY::Identity()) {}

    template<class Func>
    VectorX predict(Func &&func) {
        ceres::Jet<double, N_X> Xe_auto_jet[N_X];
        for (int i = 0; i < N_X; i++) {
            Xe_auto_jet[i].a = Xe[i];
            Xe_auto_jet[i].v[i] = 1;
        }
        ceres::Jet<double, N_X> Xp_auto_jet[N_X];
```

```
            func(Xe_auto_jet, Xp_auto_jet);
            for (int i = 0; i < N_X; i++) {
                Xp[i] = Xp_auto_jet[i].a;
                F.block(i, 0, 1, N_X) = Xp_auto_jet[i].v.transpose();
            }
            P = F * P * F.transpose() + Q;
            return Xp;
        }

        template<class Func>
        VectorX update(Func &&func, const VectorY &Y) {
            ceres::Jet<double, N_X> Xp_auto_jet[N_X];
            for (int i = 0; i < N_X; i++) {
                Xp_auto_jet[i].a = Xp[i];
                Xp_auto_jet[i].v[i] = 1;
            }
            ceres::Jet<double, N_X> Yp_auto_jet[N_Y];
            func(Xp_auto_jet, Yp_auto_jet);
            for (int i = 0; i < N_Y; i++) {
                Yp[i] = Yp_auto_jet[i].a;
                H.block(i, 0, 1, N_X) = Yp_auto_jet[i].v.transpose();
            }
            K = P * H.transpose() * (H * P * H.transpose() + R).inverse
    ();
            Xe = Xp + K * (Y - Yp);
            P = (MatrixXX::Identity() - K * H) * P;
            return Xe;
        }

        VectorX Xe;      // 估计状态变量
        VectorX Xp;      // 预测状态变量
        MatrixXX F;      // 预测雅克比
        MatrixYX H;      // 观测雅克比
        MatrixXX P;      // 状态协方差      *
        MatrixXX Q;      // 预测过程协方差   *    -
        MatrixYY R;      // 观测过程协方差   *
        MatrixXY K;      // 卡尔曼增益
        VectorY Yp;      // 预测观测量
    };


    constexpr int Z_N = 2, X_N = 3;
    /**
     * x[0] : y
     * x[1] : x
     * x[2] : k
     */
    struct Predict
```

```
{
    template <typename T>
    void operator () (const T x0[X_N], T x1[X_N])
    {
        x1[0] = x0[2] * (x0[1] + delta_x) * (x0[1] + delta_x);
        x1[1] = x0[1] + delta_x;
        x1[2] = x0[2];
    }

    double delta_x = 1;
}predict;

struct Measure
{
    template <typename T>
    void operator () (const T x0[X_N], T z0[Z_N])
    {
        z0[0] = x0[0];
        z0[1] = z0[1];
    }
}measure;

int main()
{
    srand(0);

    // generate data with noise
    const int N = 100;
    const double k = 1;
    Matrix<double, 1, N> noise = Matrix<double, 1, N>::Random();
    Matrix<double, 1, N> data = Matrix<double, 1, N>::LinSpaced(0, k
* (N - 1));
    data += noise;
    data = (data.array() * data.array()).matrix();
    std::cout << "a = [" << data << "]" << std::endl;


    // calculate speed
    AdaptiveEKF<X_N, Z_N> ekf;
    ekf.Q <<    1, 0, 0,
                0, 1, 0,
                0, 0, 1;
    ekf.R << 1, 0, 0, 1;
    ekf.Xe << data[0], 0, 0;

    std::cout << "b = [0 ";
    for (int i = 1; i < N; i++)
    {
```

```
            ekf.predict(predict);
            Matrix<double, Z_N, 1> watch;
            watch << data[i], i;
            auto xe = ekf.update(measure, watch);

            std::cout << 2 * xe[1] * xe[2] << " ";
        }
        std::cout << "]";


        return 0;
    }
```

下面是他的表现：



可以看到，EKF对于函数$f(x) = x^2$的滤波结果几乎逼近它的导数$f^{'}(x) = 2x$。
相比与普通的线性卡尔曼滤波，他在平滑程度和抗噪声表现上都显得更加优秀。

EKF算法的不足

在大多数情况下，EKF已经能够满足我们的大多数需求，如果硬要说它的不足的话：

- EKF算法具有卡尔曼滤波一类的算法共同的不足，只由$k-1$项推断第k项而不考虑历史结果

- EKF在线性性极差的环境下表现不够优秀

对于第一条不足，我们目前主要通过**非线性优化**解决。
对于第二条不足，我们有更强大的滤波器**UKF**，感兴趣的同学可以自行了解。

附录一

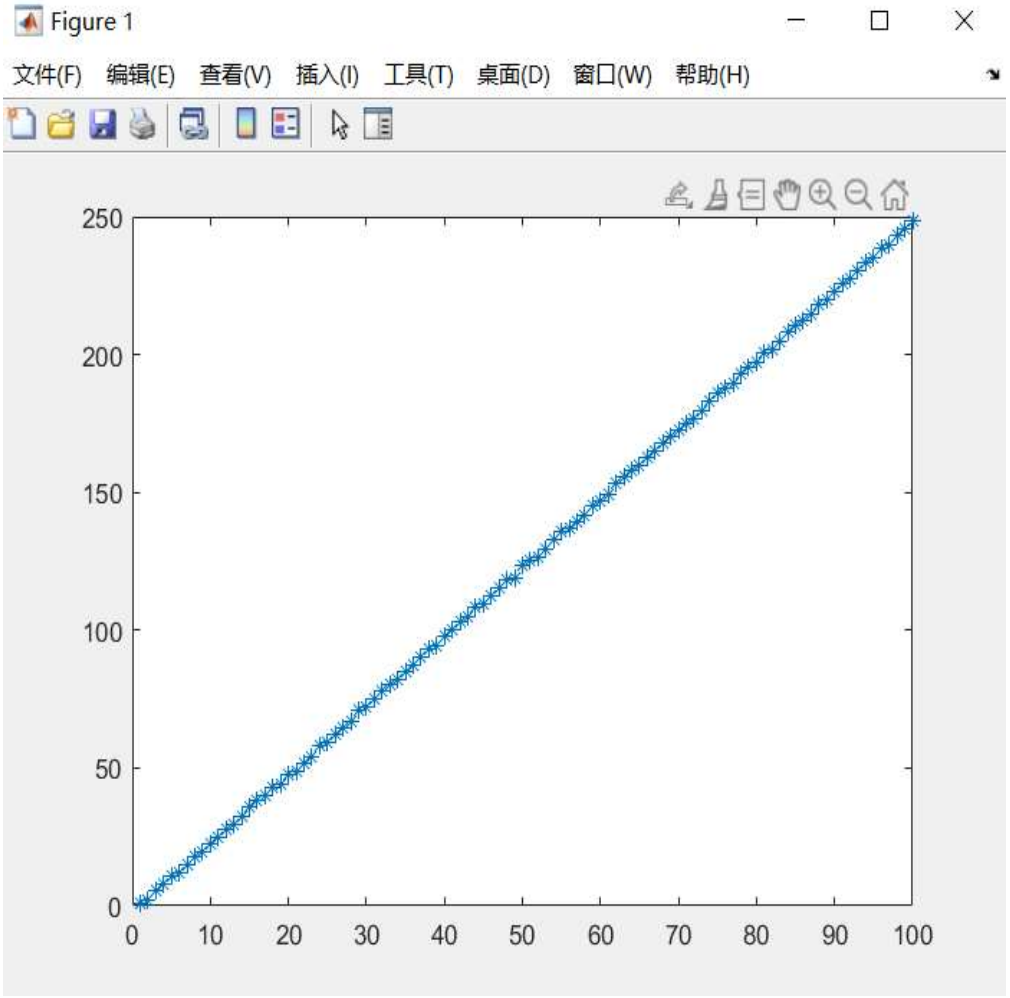
下面为一系列普通卡尔曼滤波器的调参对应的结果，为了提升可观测性，**我们把随机序列的长度增加到100，并固定随机数的种子**：

测试代码

```
#include <iostream>
#include <cstdio>
#include <string>
#include <vector>
#include <ctime>

#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <Eigen/Dense>
#include <opencv2/core/eigen.hpp>

using namespace std;
using namespace cv;
using namespace Eigen;

int main()
{

    srand(0);

    // generate data with noise
    const int N = 100;
    const double k = 2.5;
    Matrix<double, 1, N> noise = Matrix<double, 1, N>::Random();
    Matrix<double, 1, N> data = Matrix<double, 1, N>::LinSpaced(0, k
* (N - 1));
    data += noise;
    std::cout << "a = [" << data << "]" << std::endl;


    // calculate speed
    const int Z_N = 1, X_N = 2;
```

```
    Matrix<double, X_N, 1> X;
    Matrix<double, X_N, X_N> A;
    Matrix<double, X_N, X_N> P;
    Matrix<double, X_N, X_N> R;
    Matrix<double, X_N, Z_N> K;
    Matrix<double, Z_N, X_N> C;
    Matrix<double, Z_N, Z_N> Q;

    X << data[0], 0;
    A << 1, 1, 0, 1;
    C << 1, 0;
    R << 2, 0, 0, 2;
    Q << 10;

    std::cout << "b = [0 ";
    for (int i = 1; i < N; i++)
    {
        // 更新预测
        Matrix<double, X_N, 1> X_k = A * X;
        P = A * P * A.transpose() + R;

        // 更新观测
        K = P * C.transpose() * (C * P * C.transpose() + Q).inverse
();
        Matrix<double, Z_N, 1> Z{data[i]};
        X = X_k + K * (Z - C * X_k);
        P = (Matrix<double, X_N, X_N>::Identity() - K * C) * P;

        std::cout << X[1] << " ";
    }
    std::cout << "]";

    return 0;
}
```

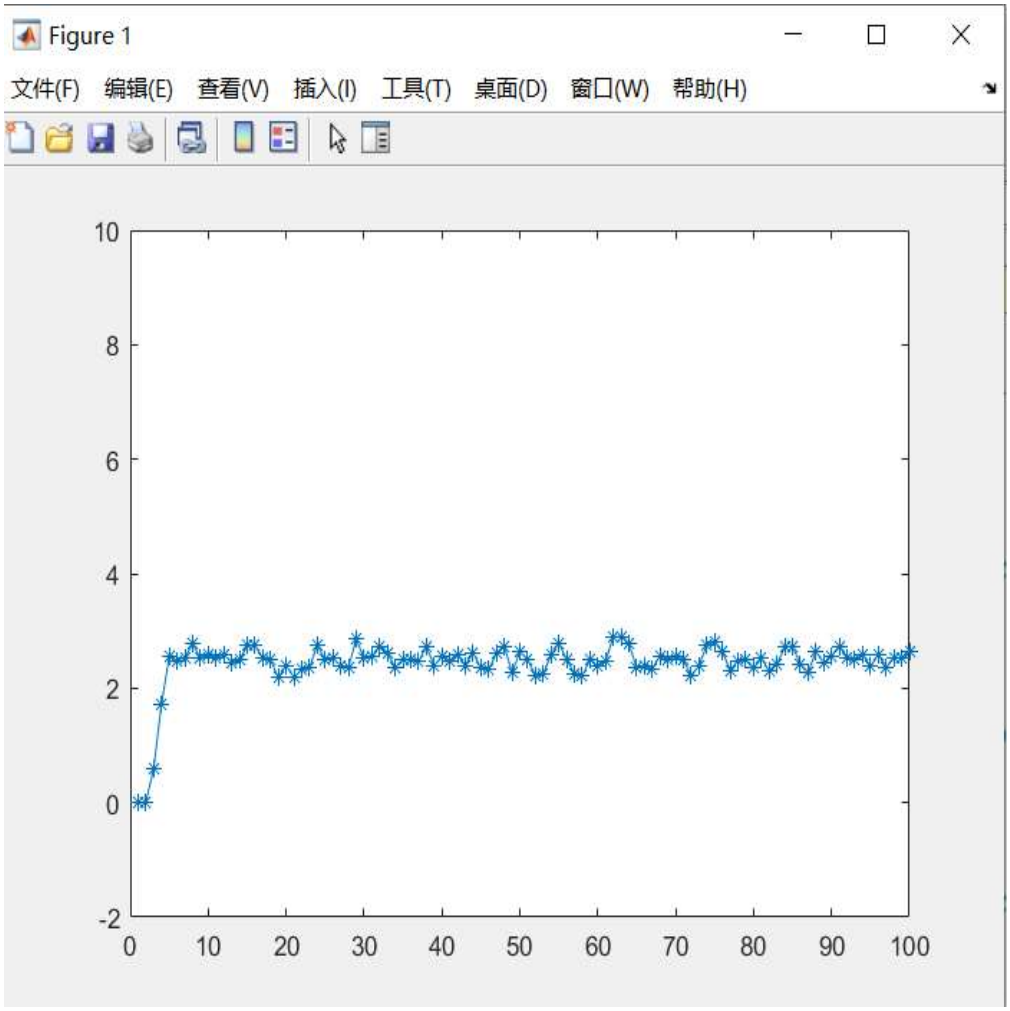
输出序列：



测试1

参数：

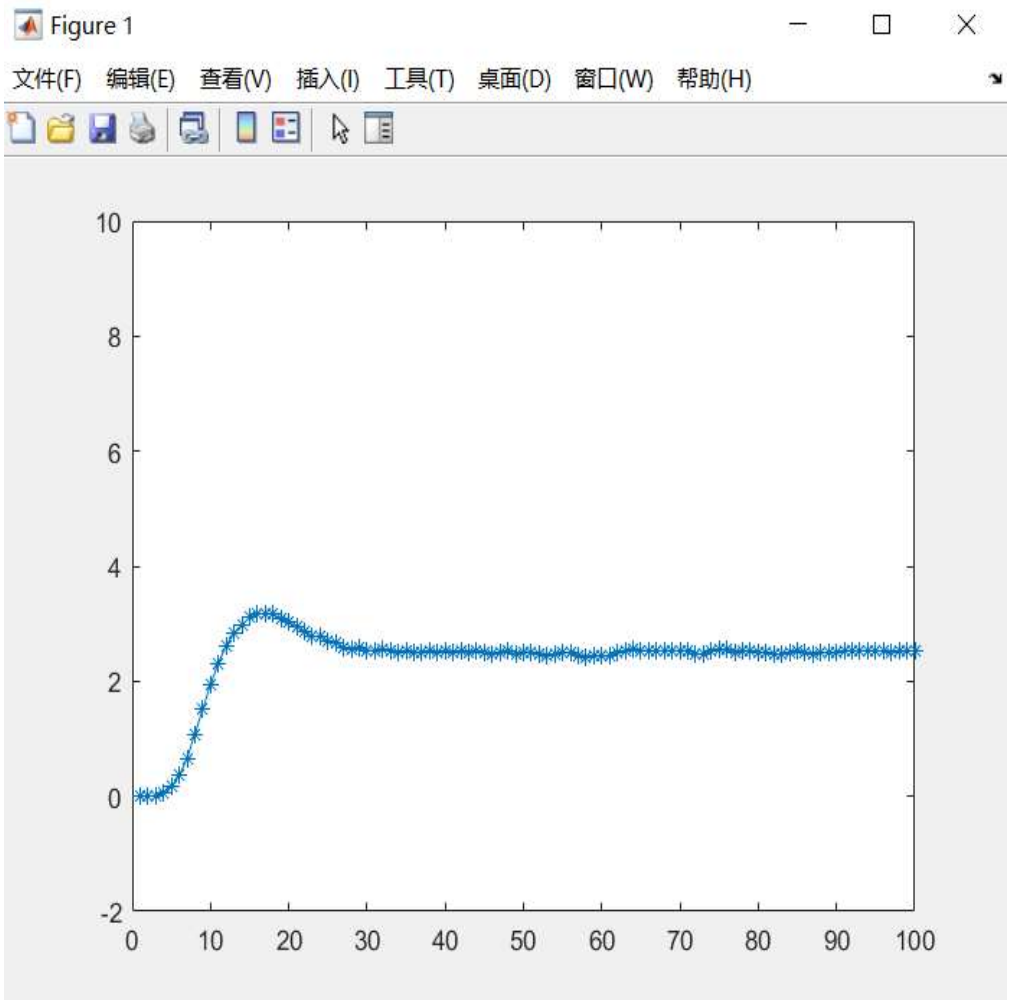
$$R = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} Q = \begin{bmatrix} 10 \\ 0 \end{bmatrix}$$



测试2

参数:

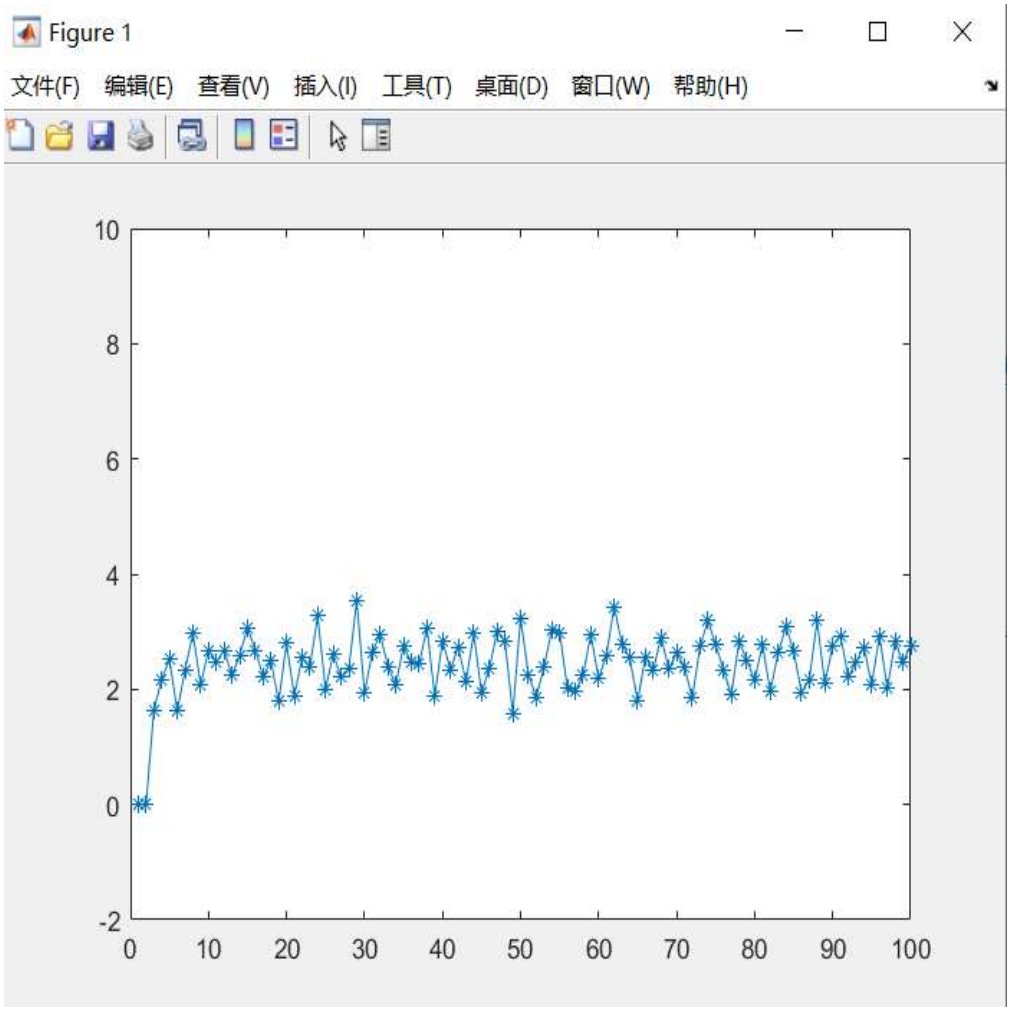
$$R = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} Q = \begin{bmatrix} 1000 \\ 0 \end{bmatrix}$$



测试3

参数:

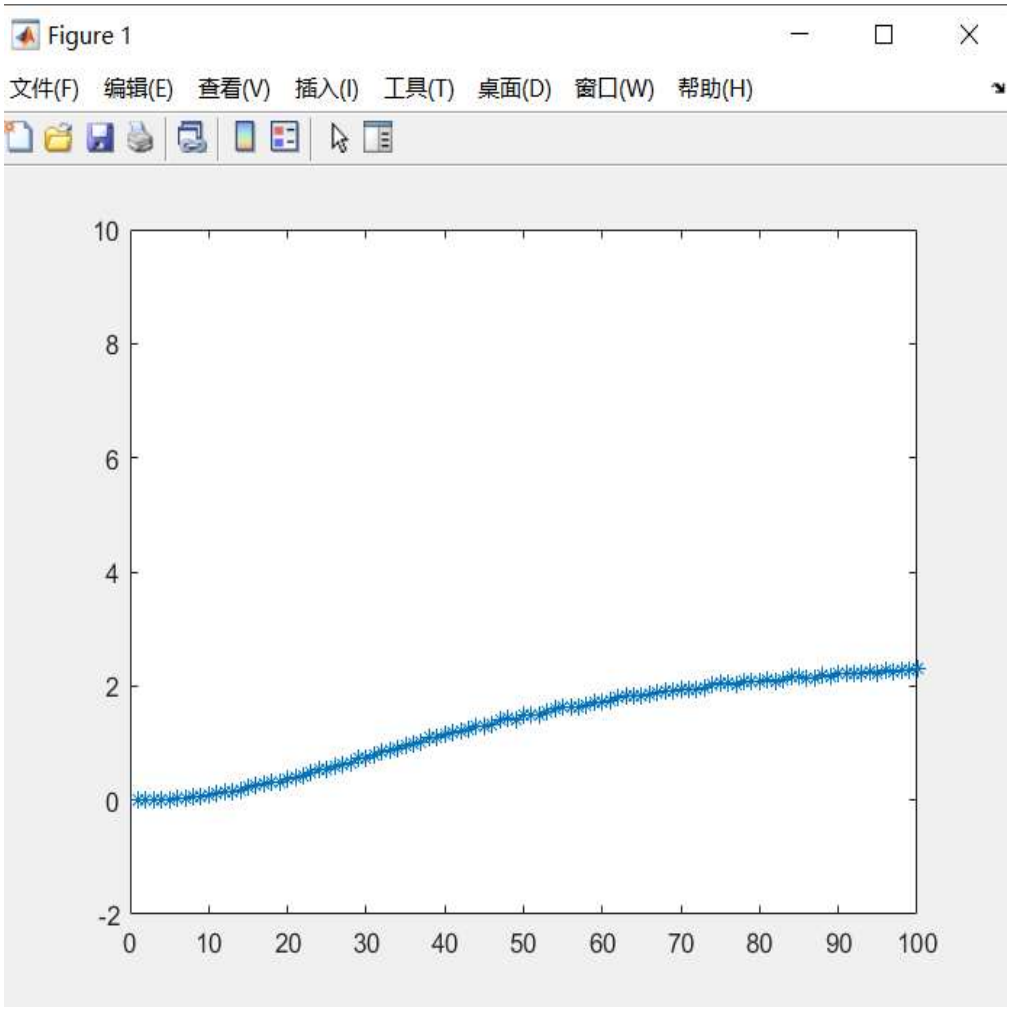
$$R = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} Q = \begin{bmatrix} 0.01 \\ 0 \end{bmatrix}$$



测试4

参数:

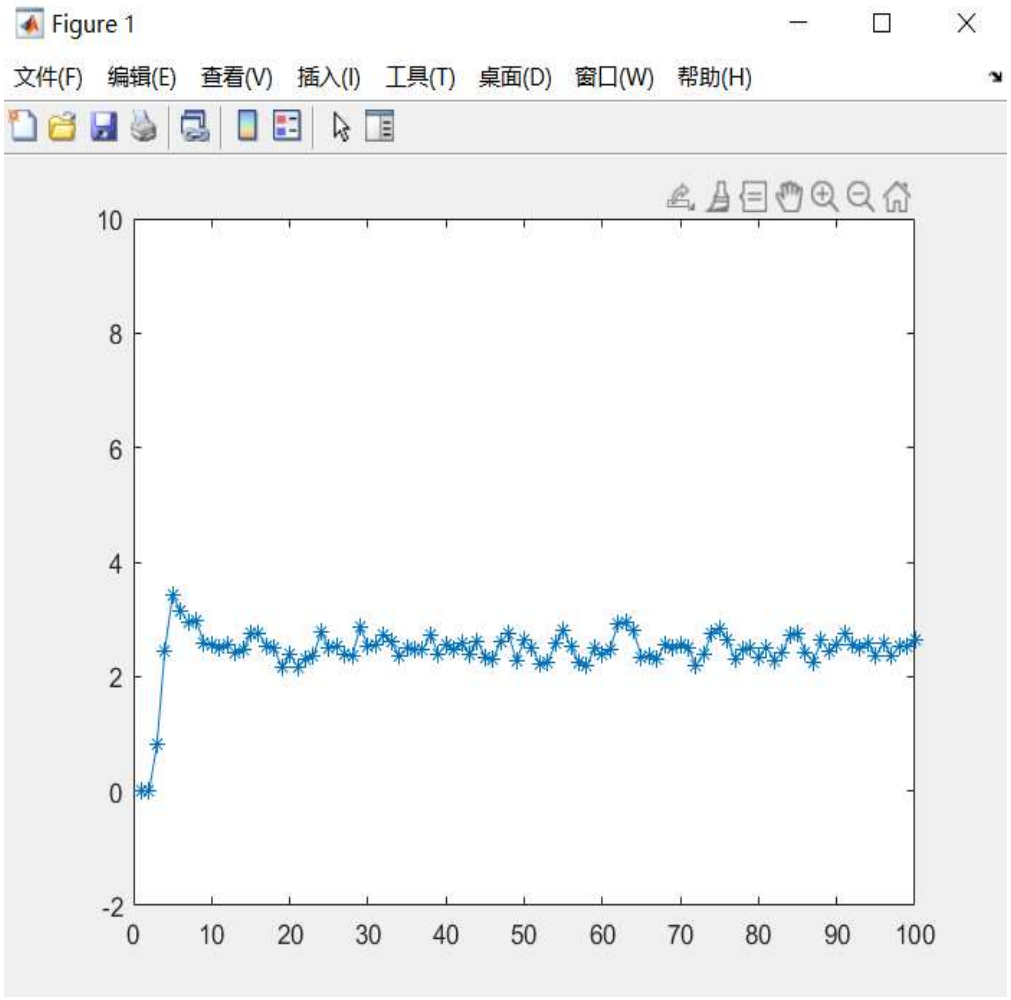
$$R = \begin{bmatrix} 2000 & 0 \\ 0 & 2 \end{bmatrix} Q = \begin{bmatrix} 10 \\ 0 \end{bmatrix}$$



测试5

参数:

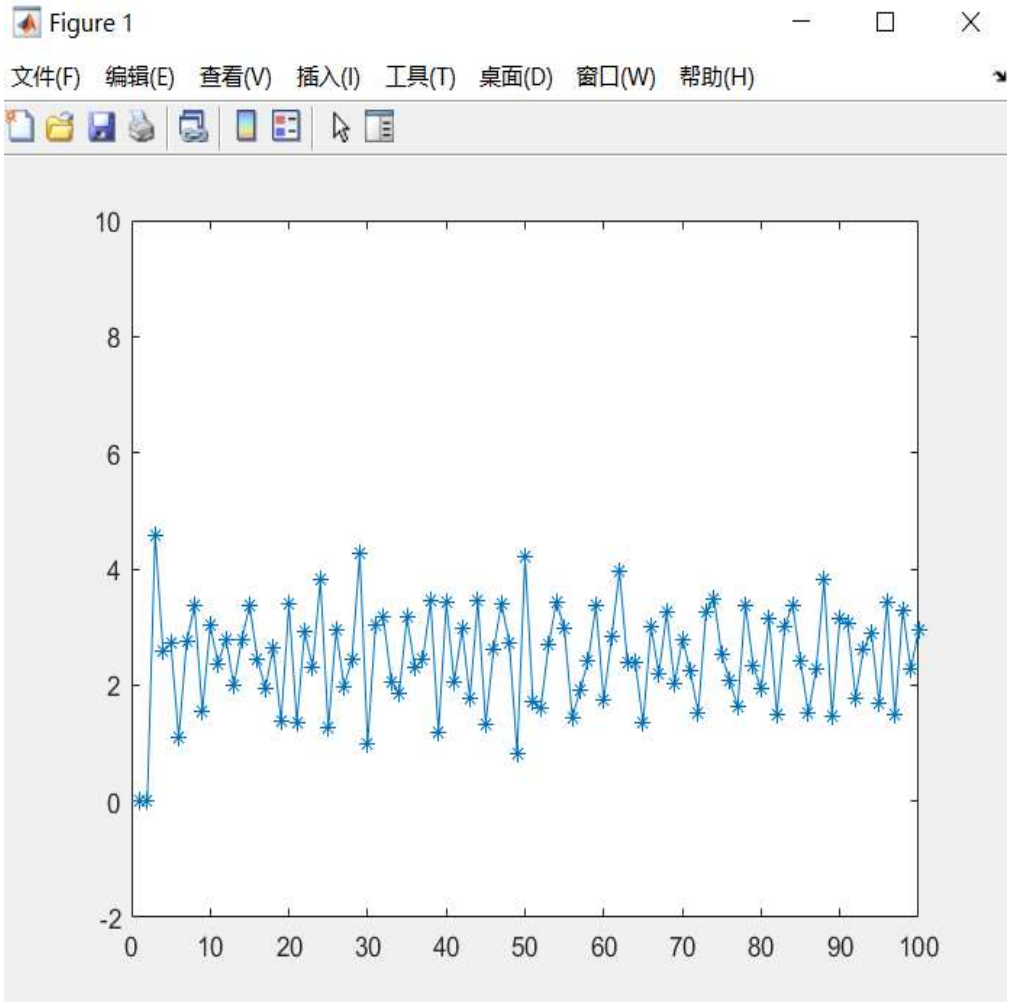
$$R = \begin{bmatrix} 0.02 & 0 \\ 0 & 2 \end{bmatrix} Q = \begin{bmatrix} 10 \\ 0 \end{bmatrix}$$



测试5

参数:

$$R = \begin{bmatrix} 2 & 0 \\ 0 & 2000 \end{bmatrix} Q = \begin{bmatrix} 10 \\ 0 \end{bmatrix}$$



测试6

参数:

$$R = \begin{bmatrix} 2 & 0 \\ 0 & 0.02 \end{bmatrix} Q = \begin{bmatrix} 10 \\ 0 \end{bmatrix}$$

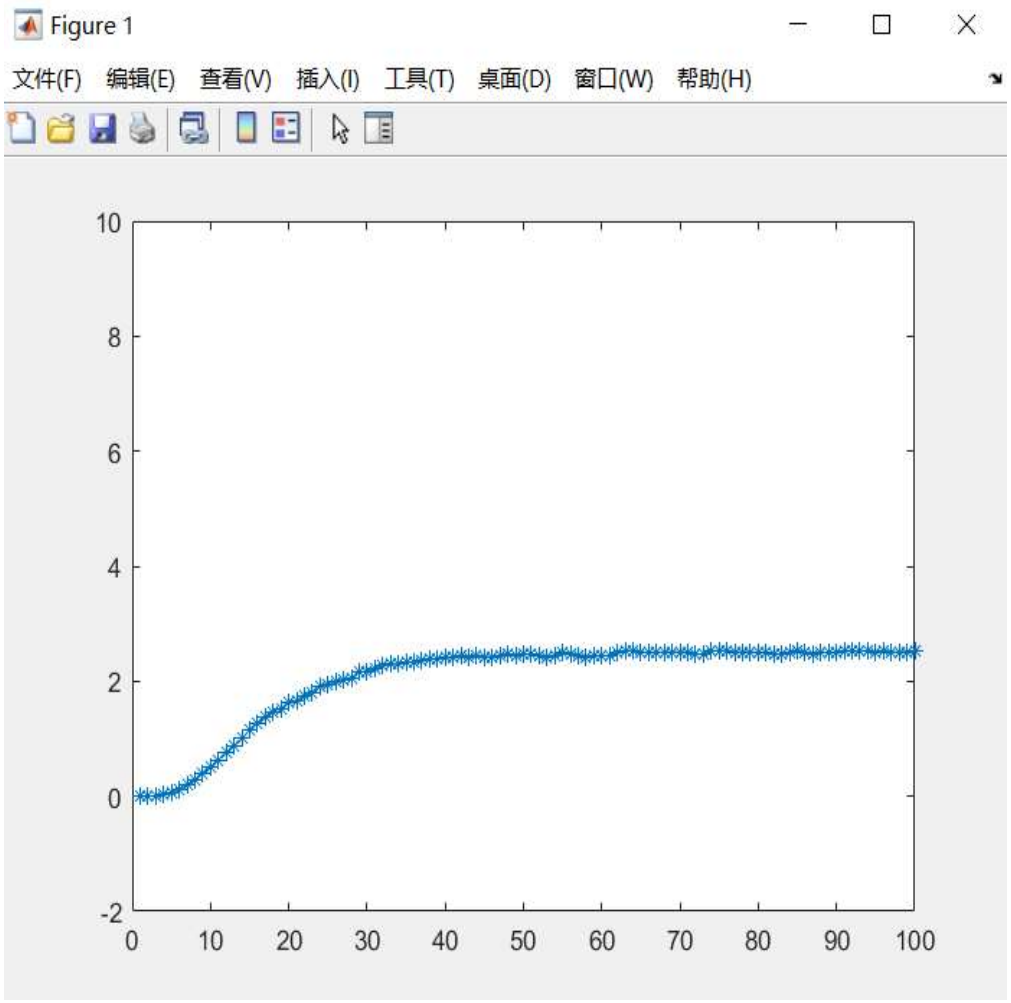


Figure 1
文件(F) 编辑(E) 查看(V) 插入(I) 工具(T) 桌面(D) 窗口(W) 帮助(H)
10
8
6
4
2
0
-2
0 10 20 30 40 50 60 70 80 90 100