河海大學

计算机控制技术课程设计

班级：自动化 1 班

学号：1905010134

姓名：刘晨阳

指导老师：苏建元

2022. 12. 18

一、设计任务

1. 设计目的：

实现 8 路独立且缓慢变化模拟量的数据采集与处理，数据采集部分包括前向通道方案选择与设计、传感器的信号调理电路设计等，数据处理部分包括标度变换、数字滤波等。

2. 系统功能：

(1)实现 8 路 A/D 转换，并且每路采样 6 次。

(2)系统显示并实现：

① 数字量(3 位，其中 1 位显示通道号)，要求自动时循环显示；手动时分步显示。

② 模拟量有正负极性(电压表)，由电压表显示；

键盘和开关：系统启动/停止键、手动运行键、单极性和双极性转换开关。

模拟量由电位器给定。

③ 用 D/A 来验证试验结果，可以通过示波器来观察试验结果。

3. 操作说明：

(1)不闭合开关 K1 为自动 A/D 转换模式，8 个通道轮流切换输入模拟量，在数码管上观察通道号和对应的转换结果；

(2)输入模拟量可以由调节滑动变阻器得到，也可以由正弦波提供；由滑动变阻器提供时，开关 K2 拔向单极性；由正弦波提供时，开关 K2 拔向双极性。

(3)闭合开关 K1 为手动模式下，手工调节通道号，可在 D/A 验证电路中验证结果：

① 观察 D/A 电路电压表与 A/D 输入的模拟量是否一致(模拟量由调节滑动变阻器得到)；

② 观察 D/A 电路示波器与 A/D 输入的模拟量是否一致(模拟量由正弦波提供)。

二、设计思路分析

1. 设计关键问题

数据的采集有两种方法实现：A/D 转换和 V/F 转换。从转换方式上，A/D 转换又分为积分 A/D 转换器和逐次逼近式 A/D 转换器等；从接口形式上又分为并行 A/D 和串行 A/D。V/F 转换是将电压信号较换为频率信号，然后测出频率再计算出物理量，它需要用计数器来测量频率只适合信号较少的场合。

目前在以单片机为核心的测量控制系统中，A/D，D/A、存储器等功能部件流行串行接口，可供选择串行接口芯片的种类也日益增多。

采用以 51 单片机作为处理器，通过 8155 芯片扩展并行接口控制 ADC0808 进行模拟量的输入到 P0 口、数据进行数字滤波后输出到 PA 口，数码管接收后进行显示，其中通道号显示 ADC0808 的模拟量输入通道。在自动模式下在 1s 后循环显示，在手动模式下，通过手动控制在 P1 口进行检测来确定通道的显示以及选择。8155 芯片 PA 口数据同时输出到 DAC0832，进行模拟量转数字量的输出，在正弦波下可以观测到数据的处理效果。采用不同的数字滤波效果不同。

在 A/D 转换之后，考虑对数据进行数字滤波，数字滤波是用程序实现的,不需要增加硬设备,所以可靠性高,稳定性好,不存在阻抗匹配的问题;同时数字滤波可以对频率很低(如 0.01Hz)的信号实现滤波,克服了模拟滤波器的缺陷;数字滤波器可根据信号的不同,采用不同的滤波方法或滤波参数,具有灵活、方便、功能强的特点。

2. 总体结构

硬件框图如下:

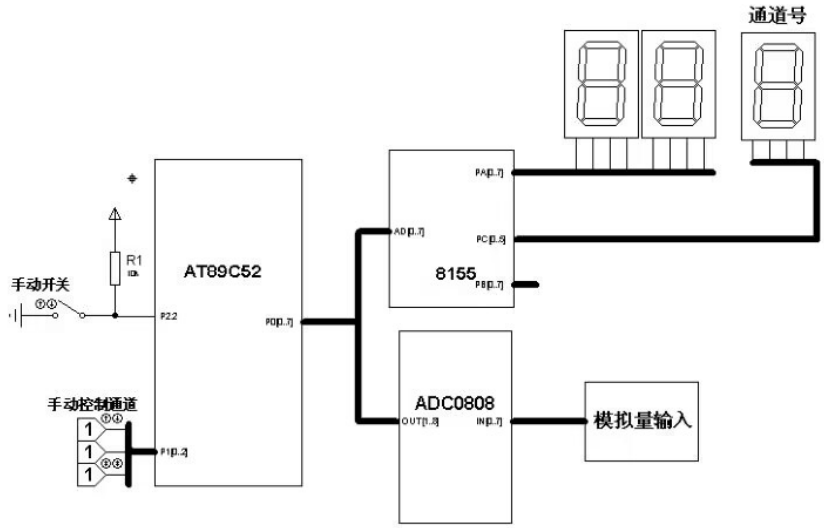

图 1：硬件框图

软件功能模块划分:

ADC0808：读取 8 路模拟量转换为数字量；

8155H： 扩展并行接口控制 ADC0808 的通道选通、数码管输出、DAC0832 数字量输入；

DAC0832：接受数字量，转换为模拟量，可以接受双极性以及单极性。

各其他算法模块、算法处理模块。

三、硬件设计

1. 系统电路图

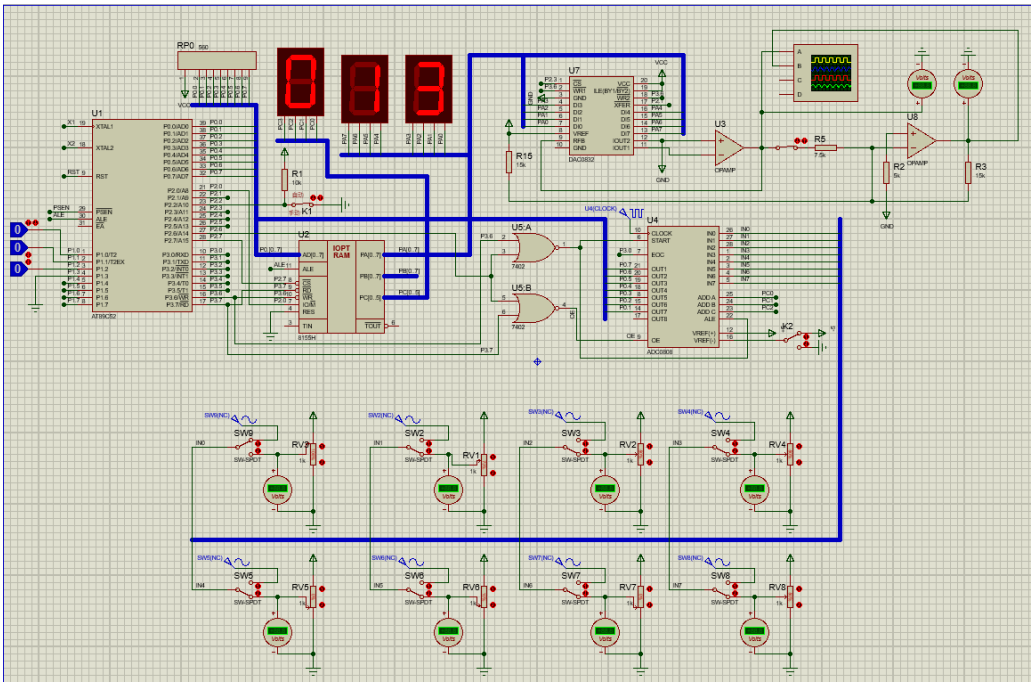

图 2：系统电路图

2. 关键电路分析

8155 芯片：

8155 是一种通用的多功能可编程 RAM/IO 扩展器，可编程是指其功能可由计算机的指令来加以改变。8155 片内不仅有 3 个可编程并行 I/O 接口(A 口、B 口为 8 位、C 口为 6 位)，而且还有 256 字节的 SRAM 和一个 14 位定时/计数器，常用作单片机的外部扩展接口，与键盘、显示器等外围设备连接。

8155 引脚图如下：

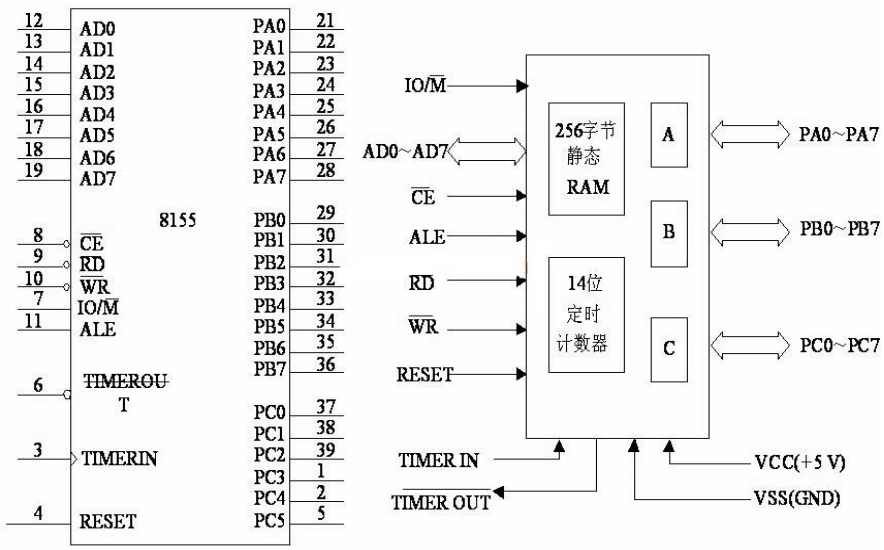

图 3：8155 芯片引脚图

实验中，AD0～AD7 为三态的地址/数据总线，与单片机的低 8 位地址/数据总线（P0 口）相连。单片机与 8155 之间的地址、数据、命令与状态信息都通过这个总线口传送。 $\overline{RD}$与 P3.7 口连接，控制对 8155 的读操作；$\overline{WR}$与 P3.6 口连接，控制对 8155 的写操作，都是低电平有效。$\overline{CS}$是片选信号线，低电平有效。IO/$\overline{M}$是 8155 的 RAM 存储器或 I/O 口选择线。当 IO/$\overline{M}$=0 时，则选择 8155 的片内 RAM，AD0～AD7 上地址为 8155 中 RAM 单元的地址；当 IO/$\overline{M}$=1 时，选择 8155 的 I/O 口，AD0～AD7 上的地址为 8155 I/O 口的地址。此外，8155 内部设有地址锁存器，在 ALE 的下降沿将单片机 P0 口输出的低 8 位地址信息及 IO/$\overline{M}$的状态都锁存到 8155 内部锁存器。因此，P0 口输出的低 8 位地址信号不需外接锁存器。PA0～PA7 与两个显示数字量的数码管及 DAC0832 的 DI0～DI7 引脚相连。PC0～PC5 作为通用的 I/O 口，与显示通道号的 8 位 7 段数码显示器相连。TIN 与 $\overline{TOUT}$分别是定时/计数器脉冲输入、输出端。

ADC0808 芯片：

ADC0808 的精度为 1/2LSB。在 AD 转换器内部有一个高阻抗斩波稳定比较器，一个带模拟开关树组的 256 电阻分压器，以及一个逐次通近型寄存器。8 路的模拟开关的通断由地址锁存器和译码器控制，可以在 8 个通道中任意访问一个单边的模拟信号。

ADC0808 芯片有 28 条引脚，采用双列直插式封装，如图所示：

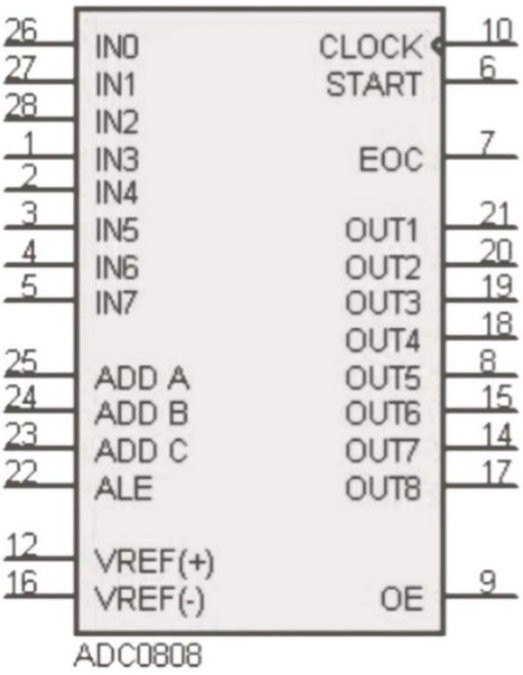

图 4：ADC0808 芯片引脚图

采取逐次逼近法转换过程是：初始化时将逐次逼近寄存器各位清零；转换开始时，先将逐次逼近寄存器最高位置 1，送入 D/A 转换器，经 D/A 转换后生成的模拟量送入比较器，称为 Vo，与送入比较器的待转换的模拟量 Vi 进行比较，若 Vo<Vi，该位 1 被保留，否则被清除。然后再置逐次逼近寄存器次高位为 1，将寄存器中新的数字量送 D/A 转换器，输出的 Vo 再与 Vi 比较，若 Vo<Vi，该位 1 被保留，否则被清除。重复此过程，直至逼近寄存器最低位。转换结束后，将逐次逼近寄存器中的数字量送入缓冲寄存器，得到数字量的输出。

通过或非门确认 ADC0808 通道 0 的地址为 0xbf00，通道二的地址为 0xbf01，以此类推，通道 7 的地址为 0xbf07。ADC 结束的标志位为 P3^0。ADC 转化的过程主要分三步：开始 ADC 转化，等待结束，读取转化的数据。

DAC0832 芯片：

DAC0832 是采样频率为八位的 D/A 转换芯片，集成电路内有两级输入寄存器，使 DAC0832 芯片具备双缓冲、单缓冲和直通三种输入方式，以便适于各种电路的需要(如要求多路 D/A 异步输入、同步转换等)。是含 8 位 A/D 转换器、8 路多路开关，以及与微型计算机兼容的控制逻辑的 CMOS 组件，其转换方法为逐次逼近型。

DAC0832 是 20 引脚的双列直插式芯片，其引脚图如下：

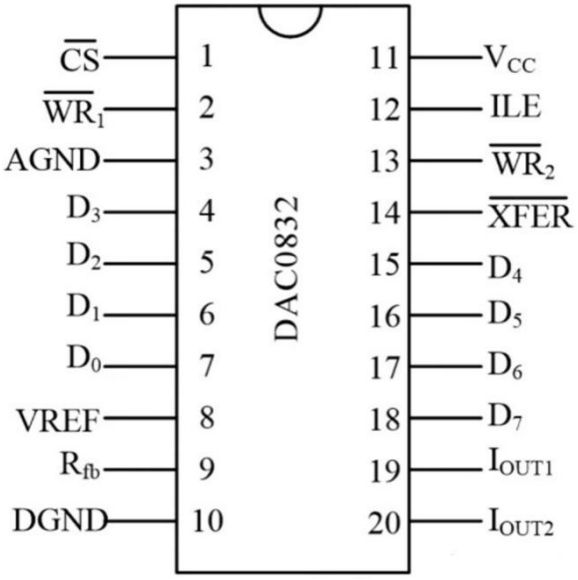

图 5：DAC0832 芯片引脚图

D10～D17 为 8 位数据输入线，Iout1 和 Iout2 为电流输出引脚，Iout1 值随 DAC 寄存器的内容线性变化，且二者之和为常数。DAC0832 转换输出端为电流,由于要求转换结果为电压，所以在 DAC0832 的输出端接运算放大器,将电流信号转换成电压信号。

四、软件设计

1. 主程序框图及说明

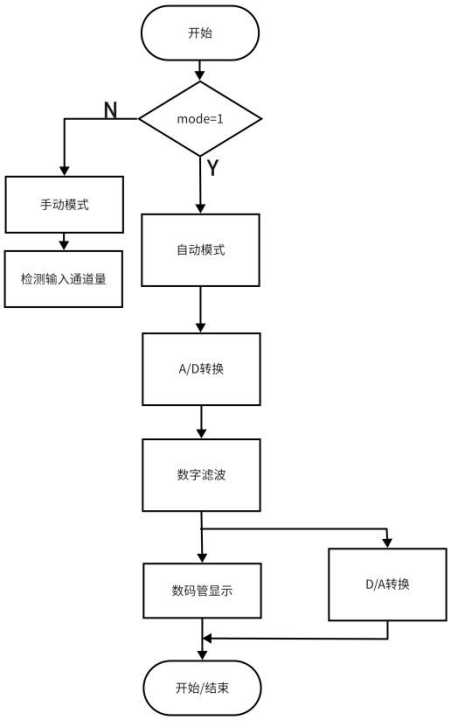

图 6：主程序流程图

五、仿真或实验结果分析与总结

仿真或实验结果分析:

1. 数字滤波部分

1.1 滤波实验总览

为了更清晰观察滤波效果，在 proteus 硬件仿真之前，我们先用 matlab 实现了限速滤波、限幅滤波、中位数滤波、算术平均滤波、递推平均滤波、限幅平均滤波、一阶滞后滤波、加权平均滤波、消抖滤波、限幅消抖滤波、卡尔曼滤波共 11 种滤波。在结合 proteus 的 C 语言仿真上，我们用 C 语言实现了限速滤波、限幅滤波、中位数滤波、算术平均滤波、递推平均滤波、中位数平均滤波、限幅平均滤波、一阶滞后滤波、加权平均滤波、消抖滤波、限幅消抖滤波共 11 种滤波（ram 不够用了）。考虑到汇编不常用，我们只用汇编实现了 4 种滤波。

我们对每一种滤波会先展示其编程实现，再根据仿真波形，介绍其特点。

1.2 具体滤波实验

1.2.1 限速滤波

```
function [data_filtered] = SpeedLimit_Filter(data_raw, limit_threshold)%限速滤波
    data_length = length(data_raw);
    data_filtered = zeros(1,data_length);
    for i = 1:2
        data_filtered(i) = data_raw(i);
    end
    for i = 3:data_length
        if abs(data_raw(i-1) - data_raw(i-2)) <= limit_threshold
            data_filtered(i) = data_raw(i-1);
        elseif abs(data_raw(i) - data_raw(i-1)) <= limit_threshold
            data_filtered(i) = data_raw(i);
        else
            data_filtered(i) = (data_raw(i) + data_raw(i-1))/2;
        end
    end
end
```

图 7 限速滤波 matlab 实现

```c
uchar SpeedLimit_Filter(uchar data_raw, uchar limit_threshold)//限速滤波
{
  static uchar data_buff[3];
  static uchar income_times;
  uchar data_temp;

  if(income_times<3)
  {
    data_buff[income_times] = data_raw;
    income_times = income_times + 1;
  }
  else
  {
    data_buff[0] = data_buff[1];
    data_buff[1] = data_buff[2];
    data_buff[2] = data_raw;
  }

  if(income_times<3)
  {
    data_temp = data_buff[2];
  }
  else
  {
    if(abs(data_buff[1] - data_buff[0]) <= limit_threshold)
    {
      data_temp = data_buff[1];
    }
    else
    {
      if(abs(data_buff[1] - data_buff[0]) > limit_threshold)
      {
        data_temp = (data_buff[2] + data_buff[1])/2;
      }
      else
      {
        data_temp = data_buff[2];
      }
    }
  }
  return data_temp;
}
```

图 8 限速滤波 C 语言实现

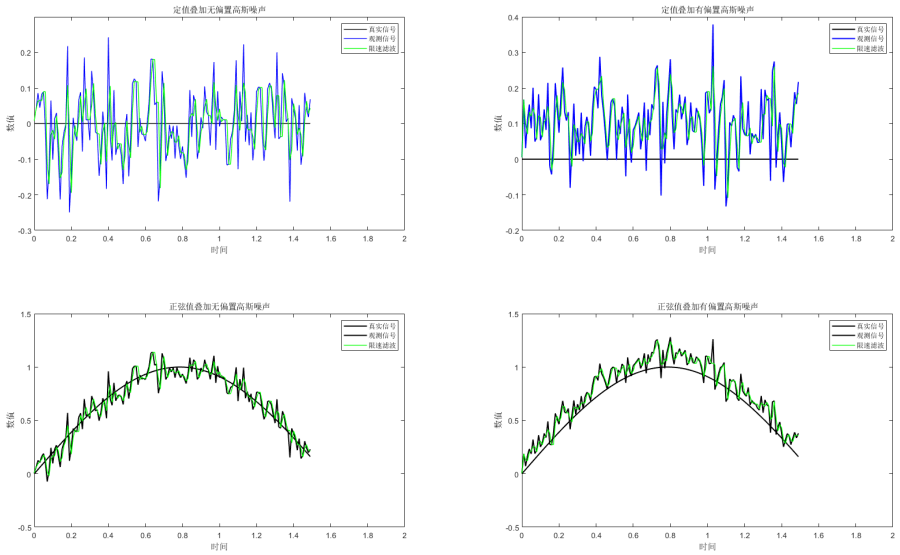

图 9 matlab 限速滤波效果

根据上图，我们可以看到限速滤波实现了抑制数据跳变过快的功能，滤波后的曲线更加平滑，距离真实值也更加接近。

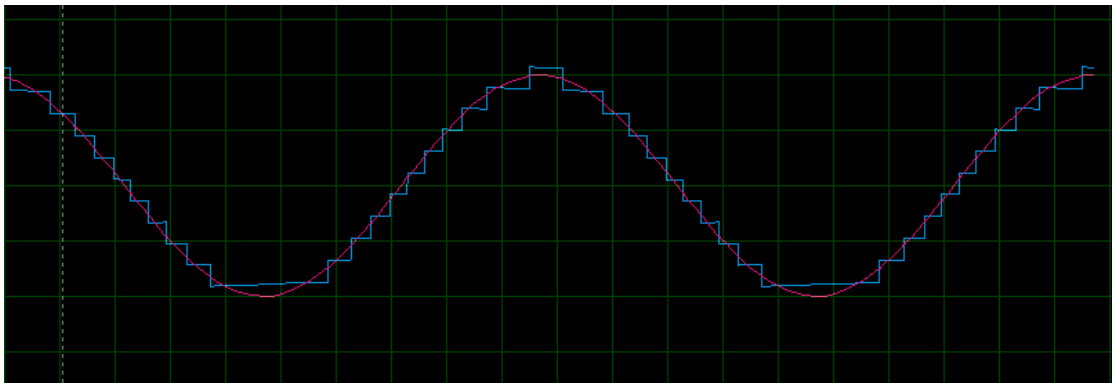

图 10 proteus 限速滤波实现

由于 proteus 中并没有给数据输入端加入干扰信号，因此就滤波效果而言，很难做出分辨，但由于其是在线计算输出的，相较 matlab 可以更好观测其动态特性和滞后性。限速滤波没有明显滞后性（其实稍微有点）。限速滤波的优点我没有明显体会到，他当然可以限速，其他很多滤波也可以，他相比其他部分滤波好一点的地方一个是编程简单，一个是滞后很小。但相较于优点，缺点太明显了，只有一个可调参数，规则十分生硬，不好理解，总的来说就是不利于调优，我目前不倾向于在平常实践中采用这种滤波。

1.2.2 限幅滤波

```
function [data_filtered] = Limiting_Filter(data_raw, limit_threshold)%限幅滤波
    data_length = length(data_raw);
    data_filtered = zeros(1,data_length);
    data_filtered(1) = data_raw(1);
    for i = 2:data_length
        data_filtered(i) = data_raw(i);
        if abs(data_raw(i) - data_raw(i-1)) > limit_threshold
            data_filtered(i) = data_filtered(i-1);
        end
    end
end
```

图 11 限幅滤波 matlab 实现

```
uchar Limiting_Filter(uchar data_raw, uchar limit_threshold)//限幅滤波
{
  static uchar data_buff[2];
  static uchar income_times;
  uchar data_temp;

  if(income_times<2)
  {
    data_buff[income_times] = data_raw;
    income_times = income_times + 1;
  }
  else
  {
    data_buff[0] = data_buff[1];
    data_buff[1] = data_raw;
  }

  if(abs(data_buff[1] - data_buff[0]) <= limit_threshold)
  {
    data_temp = data_buff[1];
  }
  else
  {
    data_buff[1] = data_buff[0];
    data_temp = data_buff[1];
  }
  return data_temp;
}
```

图 12 限幅滤波 C 语言实现

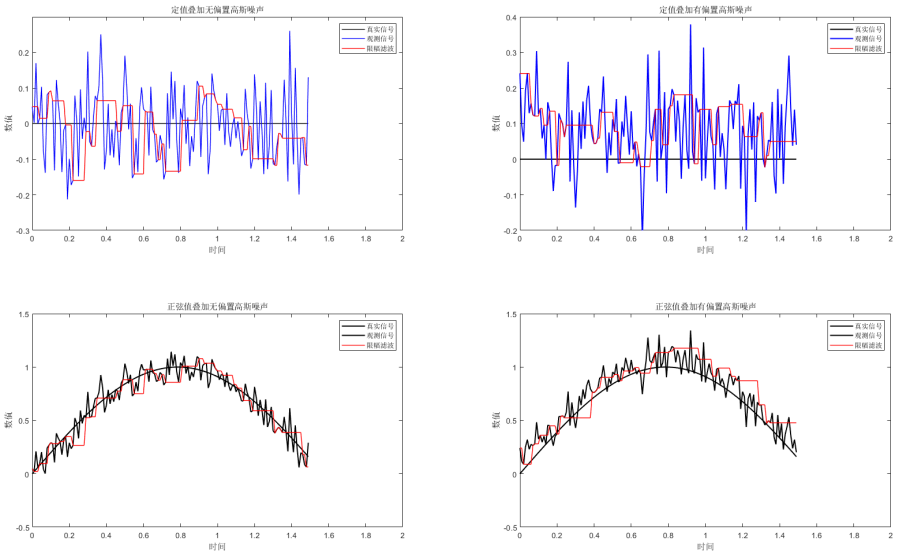

图 13 matlab 限幅滤波效果

根据上图，我们可以看到限幅滤波实现了限制两次采样间数据跳变过大的问题。滤波后的噪声明显减小了。

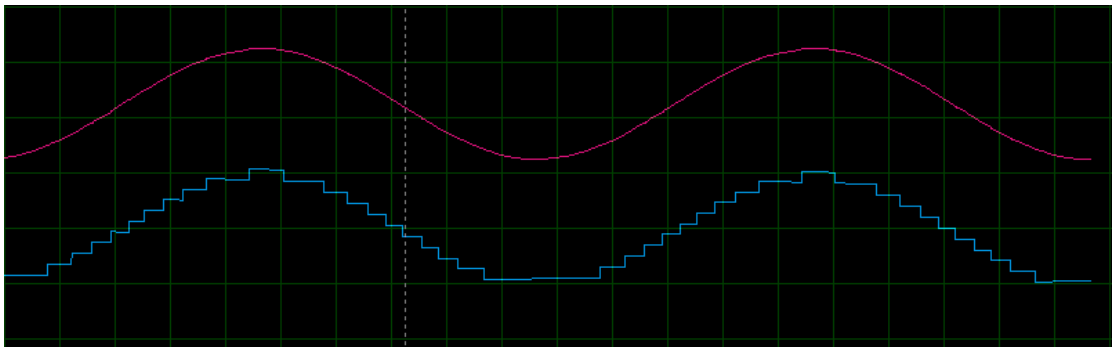

图 14 proteus 限幅滤波实现

根据上图，我们可以看到波形也没有明显滞后。并且，我认为，限幅也就实现了限速。而且限幅滤波编程起来比限速更简单，更容易理解。限幅滤波不应该乱用，如果限幅值选的太小就反而隐藏了数据，限幅值应当选择较大（标准是实际系统不应该产生如此大的跳变），来滤除掉偶然的毛刺。我曾在一次电子设计中运用过这个滤波算法，当时我们的硬件设计的不好，信号出现比较大的毛刺，我用限幅滤波滤除了毛刺。

1.2.3 中位数滤波

```matlab
function [data_filtered] = Median_Filter(data_raw, buffer_size)%中位数滤波
    data_length = length(data_raw);
    data_filtered = zeros(1,data_length);
    data_buffer = zeros(1,buffer_size);
    for i = 1:buffer_size-1
        data_filtered(i) = data_raw(i);
    end
    for i = buffer_size:data_length
        data_buffer = sort(data_raw(:,i-buffer_size+1:i));
        data_filtered(i) = data_buffer((buffer_size+1)/2);
    end
end
```

图 15 中位数滤波 matlab 实现

```c
uchar Median_Filter(uchar data_raw, uchar buffer_size)//中位数滤波
{
  static uchar data_buff[5];
  static uchar income_times;
  uchar data_temp;
  uchar i;

  if(income_times<buffer_size)
  {
    data_buff[income_times] = data_raw;
    income_times = income_times + 1;
  }
  else
  {
    for(i=0;i<income_times-1;income_times++)
    {
      data_buff[i] = data_buff[i+1];
    }
    data_buff[income_times-1] = data_raw;
  }

  if(income_times<buffer_size)
  {
    data_temp = data_buff[income_times];
  }
  else
  {
    data_temp = Mid_Sort(data_buff,buffer_size);
  }

  return data_temp;
}
```

图 16 中位数滤波 C 语言实现

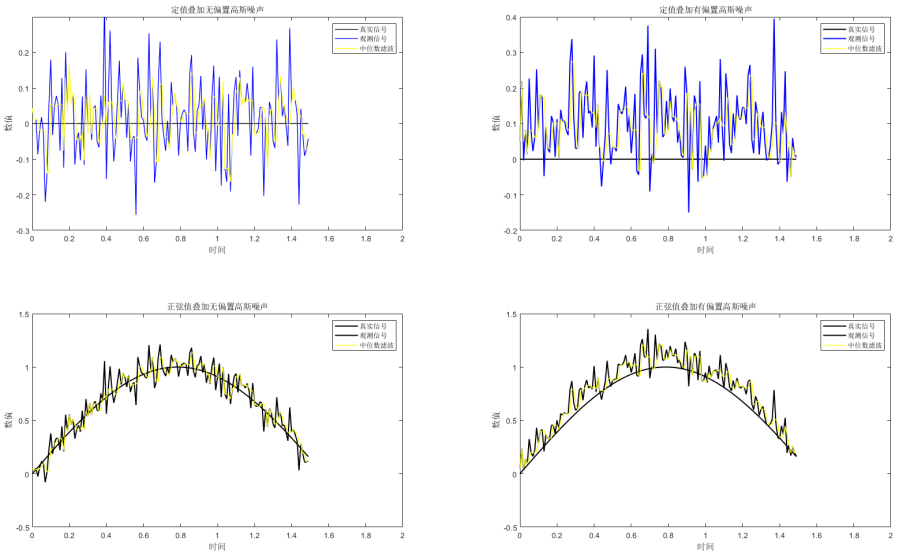

图 17 matlab 中位数滤波效果

可以看到中位数滤波一定程度上抑制了随机噪声波动。

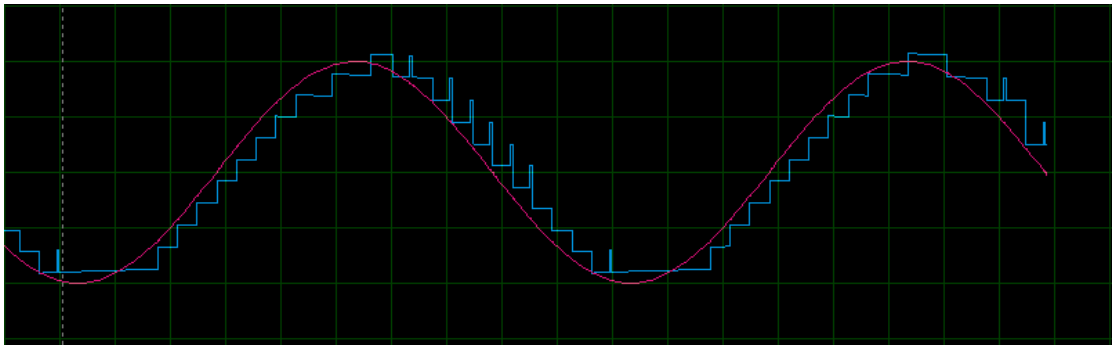

图 18 proteus 中位数滤波效果

中位数滤波效果不错，但感觉相对于滑动平均或者一阶滞后计算量较大。它首先需要排序，然后要判断奇数偶数，然后再找到中位数。编程麻烦。

1.2.4 算数平均滤波

```
function [data_filtered] = Average_Filter(data_raw, buffer_size)%算数平均滤波
    data_length = length(data_raw);
    data_filtered = zeros(1,data_length);
    for i = buffer_size:buffer_size:data_length
        sum = 0;
        for j = i-buffer_size+1:i
            sum = sum + data_raw(j);
        end
        for j = i-buffer_size+1:i
            data_filtered(j) = sum/buffer_size;
        end
        if i > data_length - buffer_size + 1
            sum = 0;
            for j = i:data_length
                sum = sum  + data_raw(j);
            end
            for j = i:data_length
                data_filtered(i) = sum/(data_length-i+1);
            end
        end
    end
end
```

图 19 算数平均滤波 matlab 实现

```c
uchar Average_Filter(uchar data_raw, uchar buffer_size)//算数平均滤波
{
  static uchar data_buff[5];
  static uchar income_times;
  static uchar load_times;
  static uchar data_temp;
  uchar i;
  uchar sum = 0;

  if(income_times<buffer_size)
  {
    data_buff[income_times] = data_raw;
    income_times = income_times + 1;
  }
  else
  {
    for(i=0;i<income_times-1;income_times++)
    {
      data_buff[i] = data_buff[i+1];
    }
    data_buff[income_times-1] = data_raw;
  }

  if(income_times<buffer_size)
  {
    data_temp = data_buff[income_times];
  }
  else
  {
    if(++load_times==buffer_size)
    {
      buffer_size = 0;
      for(i=0;i<buffer_size;i++)
      {
        sum = sum + data_buff[i];
      }
      data_temp = sum/buffer_size;
    }
  }

  return data_temp;
}
```

图 20 算数平均滤波 C 语言实现

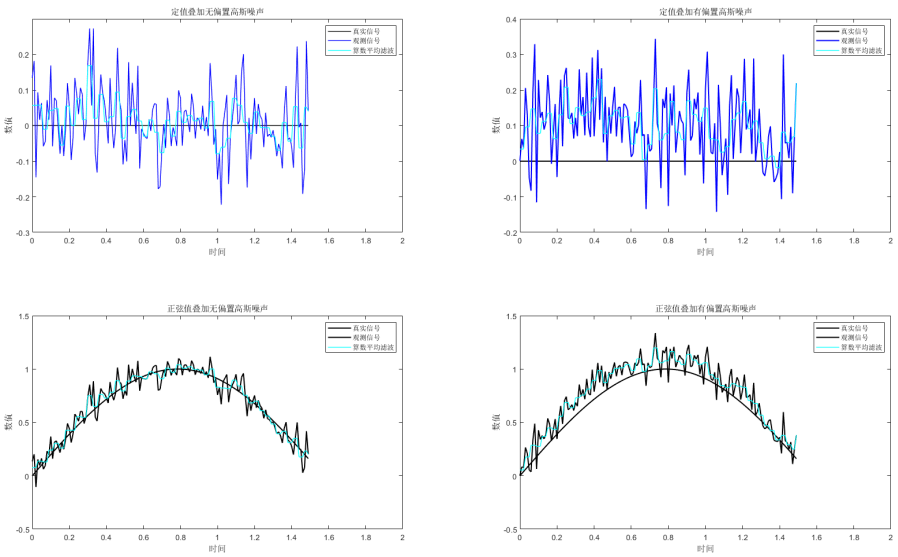

图 21 matlab 算数平均滤波效果

可以看到算数平均滤波一定程度上抑制了随机噪声波动。

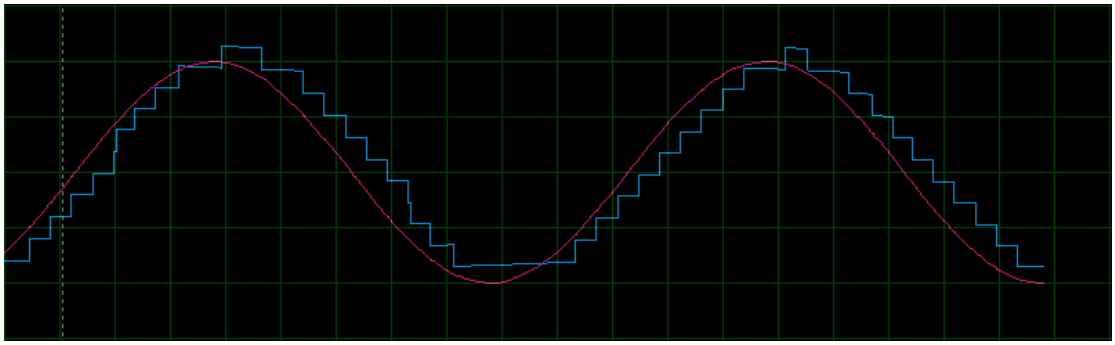

图 22 proteus 算数平均滤波效果

可以看到算数平均滤波是有较大延迟的，这是因为计算平均值必然等效地引入历史数据。因此，比较适合变化较慢的过程。就算是较慢的过程我也不推荐采用算数平均滤波算法。一方面是滞后，一方面是不灵活。

1.2.5 滑动平均滤波

```
function [data_filtered] = MovingAverage_Filter(data_raw, buffer_size)%滑动平均滤波
    data_length = length(data_raw);
    data_filtered = zeros(1,data_length);
    for i = 1:buffer_size-1
        data_filtered(i) = data_raw(i);
    end
    for i = buffer_size:data_length
        data_filtered(i) = mean(data_raw(:,i-buffer_size+1:i));
    end
end
```

图 23 滑动平均滤波 matlab 实现

```c
uchar MovingAverage_Filter(uchar data_raw, uchar buffer_size)//滑动平均滤波
{
  static uchar data_buff[5];
  static uchar income_times;
  uchar data_temp;
  uchar i;
  uchar sum = 0;

  if(income_times<buffer_size)
  {
    data_buff[income_times] = data_raw;
    income_times = income_times + 1;
  }
  else
  {
    for(i=0;i<income_times-1;income_times++)
    {
      data_buff[i] = data_buff[i+1];
    }
    data_buff[income_times-1] = data_raw;
  }

  if(income_times<buffer_size)
  {
    data_temp = data_buff[income_times];
  }
  else
  {
      for(i=0;i<buffer_size;i++)
      {
        sum = sum + data_buff[i];
      }
      data_temp = sum/buffer_size;

  }

  return data_temp;
}
```

图 24 滑动平均滤波 C 语言实现

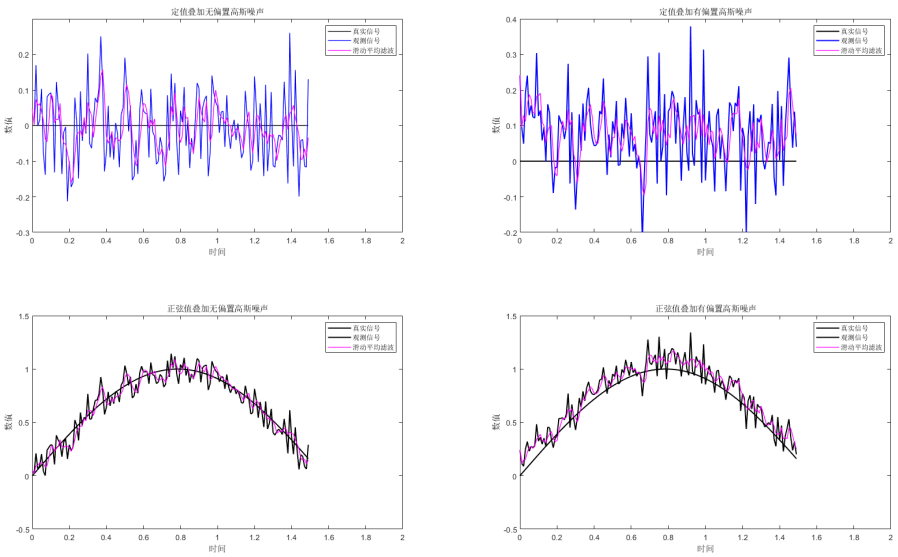

图 25 matlab 滑动平均滤波效果

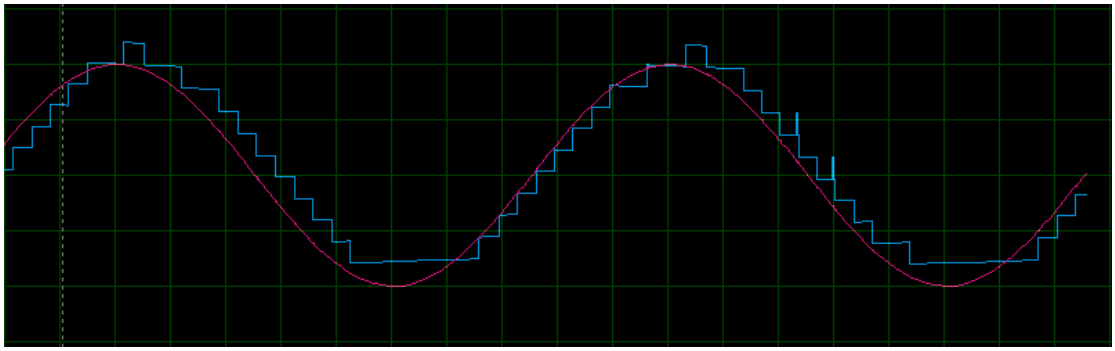

图 26 proteus 滑动平均滤波效果

由上图可知，滑动平均滤波也可以达到抑制干扰的效果，滞后还减小了。实际上，我很推荐这个滤波算法。首先因为平均数滤波的思想很好理解，其次理论上可以完全消除高频周期干扰。但也有缺点，实现它需要维护一个队列结构，比较浪费资源。

1.2.6 中位数平均滤波

```
196 uchar MidAvg_Filter(uchar data_raw, uchar buffer_size)//中位值平均滤波
197 {
198   static uchar data_buff[5];
199   static uchar income_times;
200   uchar data_temp;
201   uchar i;
202   uchar sum = 0;
203   uchar temp_buff[5];
204
205   if(income_times<buffer_size)
206   {
207     data_buff[income_times] = data_raw;
208     income_times = income_times + 1;
209   }
210   else
211   {
212     for(i=0;i<income_times-1;income_times++)
213     {
214       data_buff[i] = data_buff[i+1];
215     }
216     data_buff[income_times-1] = data_raw;
217   }
218
219   if(income_times<buffer_size)
220   {
221     data_temp = data_buff[income_times];
222   }
223   else
224   {
225       for(i=0;i<buffer_size;i++)
226       {
227         temp_buff[i] = data_buff[i];
228       }
229       Bubble_Sort(temp_buff,buffer_size);
230       for(i=1;i<buffer_size-1;i++)
231       {
232         sum = sum + temp_buff[i];
233       }
234       data_temp = sum/(buffer_size-2);
235   }
236
237   return data_temp;
238 }
```

图 27 中位数平均滤波 C 语言实现

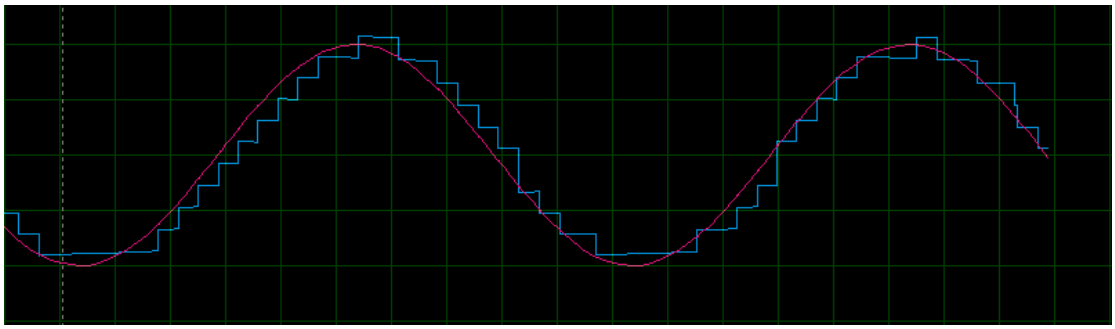

图 28 proteus 中位数平均滤波效果

我的评价：不如限幅再平均，中位数的掐头去尾太耗资源，收益和限幅差不多，还没限幅指向性明确。

1.2.7 限幅平均滤波

```
function [data_filtered] = LimitingAverage_Filter(data_raw, buffer_size, limit_threshold)%限幅平均滤波
    data_length = length(data_raw);
    data_filtered = zeros(1,data_length);
    data_filtered(1) = data_raw(1);
    for i = 2:data_length
        data_filtered(i) = data_raw(i);
        if abs(data_raw(i) - data_filtered(i-1)) > limit_threshold
            data_filtered(i) = data_raw(i-1);
        end
        if i >= buffer_size
            data_filtered(i) = mean(data_filtered(:,i-buffer_size+1:i));
        end
    end
end
```

图 29 限幅平均滤波 matlab 实现

```
uchar LimitAvg_Filter(uchar data_raw, uchar buffer_size, uchar limit_threshold)//限幅平均滤波
{
  static uchar data_buff[5];
  static uchar income_times;
  uchar data_temp;
  uchar i;
  uchar sum = 0;
  uchar temp_buff[5];

  if(income_times<buffer_size)
  {
    data_buff[income_times] = data_raw;
    income_times = income_times + 1;
  }
  else
  {
    for(i=0;i<income_times-1;income_times++)
    {
      data_buff[i] = data_buff[i+1];
    }
    data_buff[income_times-1] = data_raw;
  }

  if(income_times<buffer_size)
  {
    data_temp = data_buff[income_times];
  }
  else
  {
    temp_buff[0] = data_buff[0];
    for(i=1;i<buffer_size;i++)
    {
      if(abs(data_buff[i]-data_buff[i-1])<=limit_threshold)
      {
        temp_buff[i] = data_buff[i];
      }
      else
      {
        temp_buff[i] = data_buff[i-1];
      }
      sum = sum + temp_buff[i];
    }
```

图 30 限幅平均滤波 C 语言实现

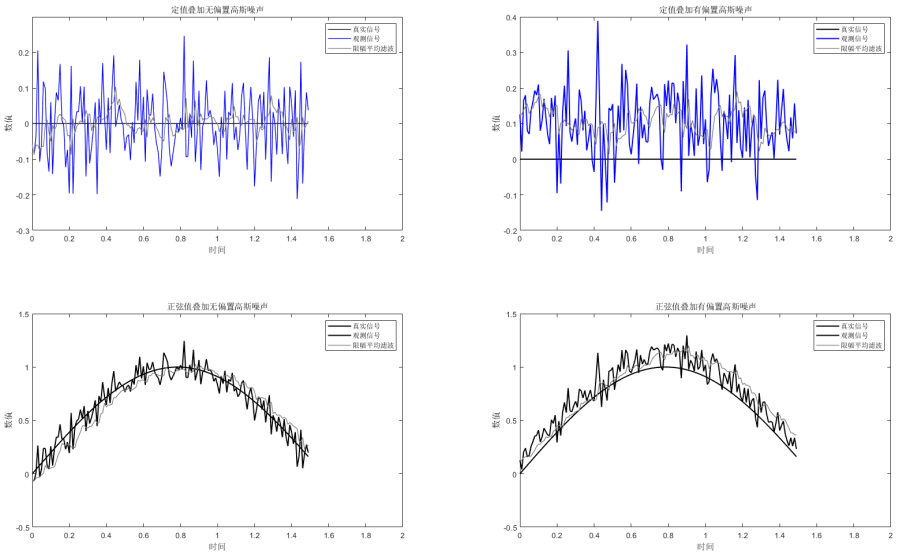

图 31 matlab 限幅平均滤波效果

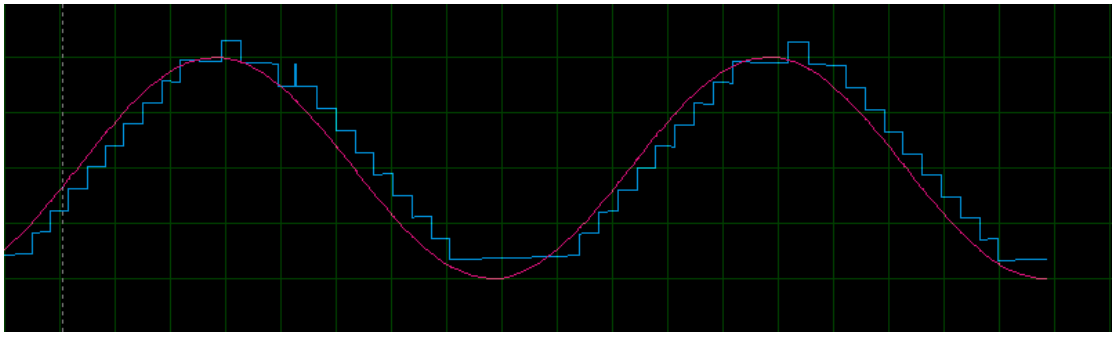

图 32 proteus 限幅平均滤波效果

效果不错。我很推崇限幅+其他滤波。

1.2.8 一阶滞后滤波

```
function [data_filtered] = FirstOrderLag_Filter(data_raw, ratio)%一阶滞后滤波
    data_length = length(data_raw);
    data_filtered = zeros(1,data_length);
    data_filtered(1) = data_raw(1);
    for i = 2:data_length
        data_filtered(i) = ratio*data_raw(i) + (1-ratio)*data_filtered(i-1);
    end
end
```

图 33 一阶滞后滤波 matlab 实现

```
289 uchar FirstOrderLag_Filter(uchar data_raw, float ratio)//一阶滞后滤波
290 {
291   static uchar data_buff[3];
292   static uchar income_times;
293   uchar data_temp;
294 
295   if(income_times<2)
296   {
297     data_buff[income_times] = data_raw;
298     income_times = income_times + 1;
299   }
300   else
301   {
302     data_buff[0] = data_buff[1];
303     data_buff[1] = data_raw;
304   }
305 
306   if(income_times<2)
307   {
308     data_temp = data_buff[income_times];
309   }
310   else
311   {
312     data_temp = ratio*data_buff[1] + (1-ratio)*data_buff[0];
313   }
314 
315   return data_temp;
316 }
```

图 34 一阶滞后滤波 C 语言实现

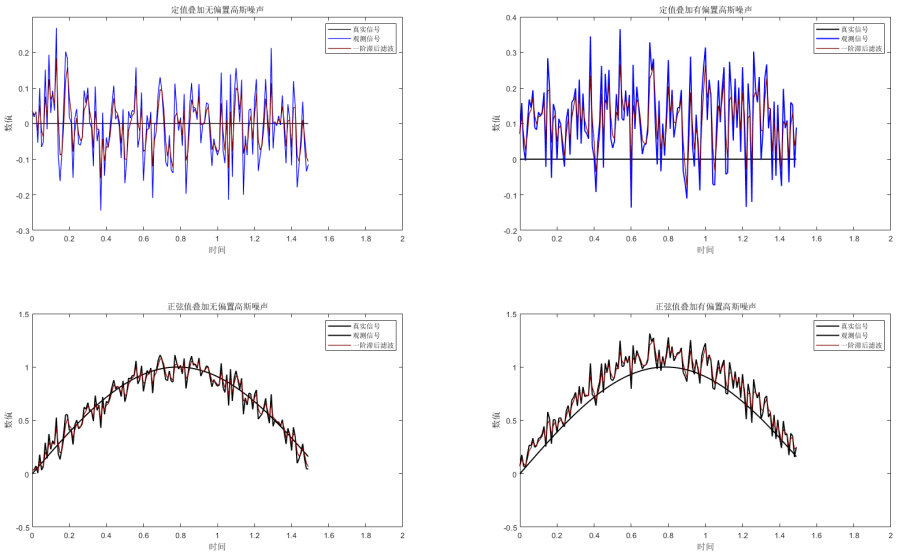

图 35 matlab 一阶滞后滤波效果

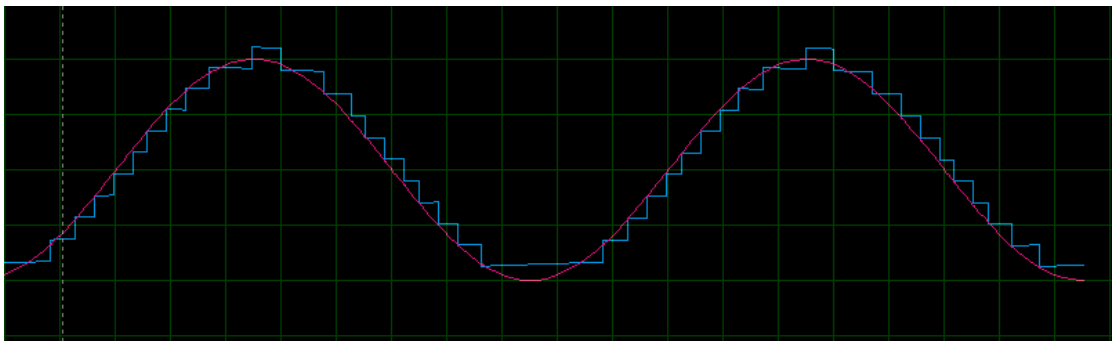

图 36 proteus 一阶滞后滤波效果

一阶滞后滤波效果不错，程序非常简单，不吃资源，我非常非常喜欢。一阶滞后滤波就是加权平均滤波的 buff=2 版本，但编程实现起来简单得多。我会首先使用一阶滞后滤波，很少会真的考虑用加权平均滤波（都有这闲工夫，不如用更高级的滤波，一阶滞后滤波就讲究个简单有效）。

1.2.9 加权平均滤波

```
function [data_filtered] = WeightedMovingAverage_Filter(data_raw, buffer_size, ratio_buffer)%滑动平均滤波
    data_length = length(data_raw);
    data_filtered = zeros(1,data_length);
    for i = 1:buffer_size-1
        data_filtered(i) = data_raw(i);
    end
    for i = buffer_size:data_length
        for j = 1:buffer_size
            data_filtered(i) = data_filtered(i) + ratio_buffer(j)*data_raw(i-j+1);
        end
    end
end
```

图 37 加权平均滤波 matlab 实现

```c
uchar WeightedMovAvg_Filter(uchar data_raw, uchar buffer_size, float ratio_buffer[])//加权平均滤波
{
  static uchar data_buff[5];
  static uchar income_times;
  float data_temp;
  uchar i;

  if(income_times<buffer_size)
  {
    data_buff[income_times] = data_raw;
    income_times = income_times + 1;
  }
  else
  {
    for(i=0;i<income_times-1;income_times++)
    {
      data_buff[i] = data_buff[i+1];
    }
    data_buff[income_times-1] = data_raw;
  }

  if(income_times<2)
  {
    data_temp = data_buff[income_times];
  }
  else
  {
    for(i=0;i<buffer_size;i++)
    {
      data_temp = data_temp + ratio_buffer[i] * data_buff[i];
    }
  }

  return (uchar)data_temp;
}
```

图 38 加权平均滤波 C 语言实现

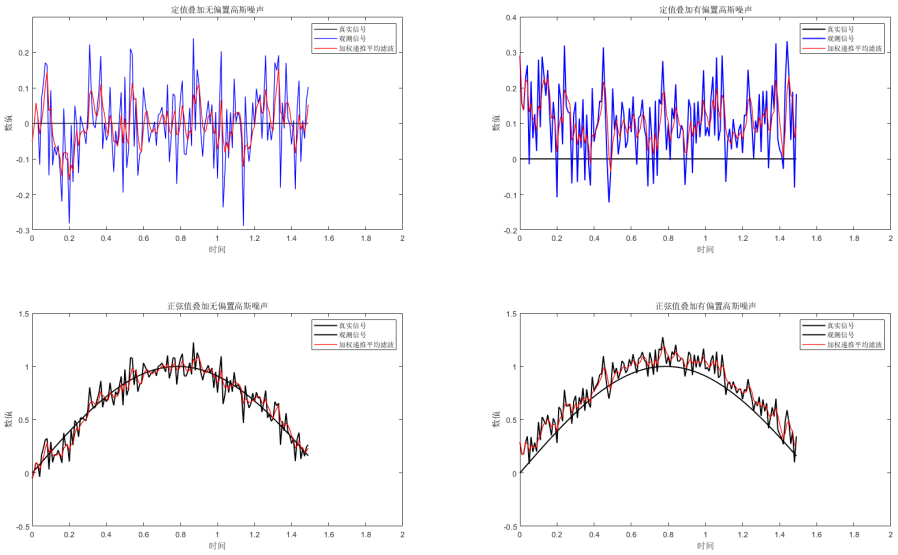

图 39 matlab 加权平均滤波效果

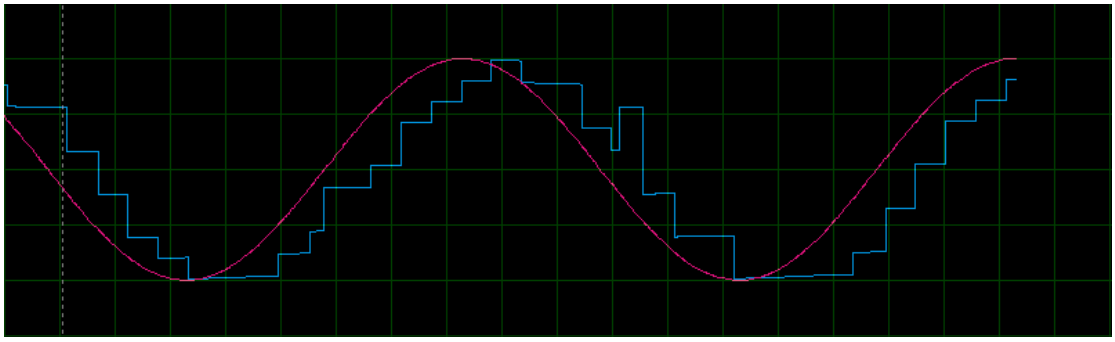

图 40 proteus 加权平均滤波效果

加权平均滤波好处就是调参自由度大，调的不好就是上面的样子。坏处就是需要维护队列，麻烦。

1.2.10 消抖滤波

```
function [data_filtered] = Glitch_Filter(data_raw, buffer_size)%消抖滤波
    data_length = length(data_raw);
    data_filtered = zeros(1,data_length);
    data_value = data_raw(1);
    buffer_counter = 0;
    for i = 2:data_length
        if data_raw(i) ~= data_value
            buffer_counter = buffer_counter + 1;
            if buffer_counter >= buffer_size
                data_value = data_raw(i);
                buffer_counter = 0;
            end
        end
        data_filtered(i) = data_value;
    end
end
```

图 41 消抖滤波 matlab 实现

```
354 uchar Glitch_Filter(uchar data_raw, uchar buffer_size)//消抖滤波
355 {
356   static uchar data_buff;
357   static uchar income_times;
358   static uchar diff_times;
359   uchar data_temp;
360 
361   if(income_times<1)
362   {
363     data_buff = data_raw;
364   }
365   else
366   {
367     if(data_raw!=data_buff)
368     {
369       if(++diff_times>buffer_size)
370       {
371         diff_times = 0;
372         data_buff = data_raw;
373       }
374     }
375   }
376   data_temp = data_buff;
377 
378   return data_temp;
379 }
```

图 42 消抖滤波 C 语言实现

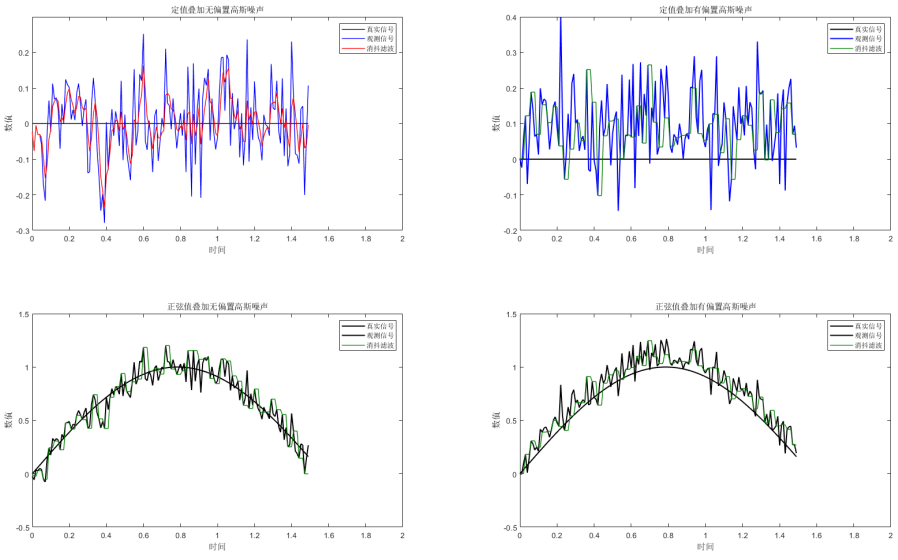

图 43 matlab 消抖滤波效果

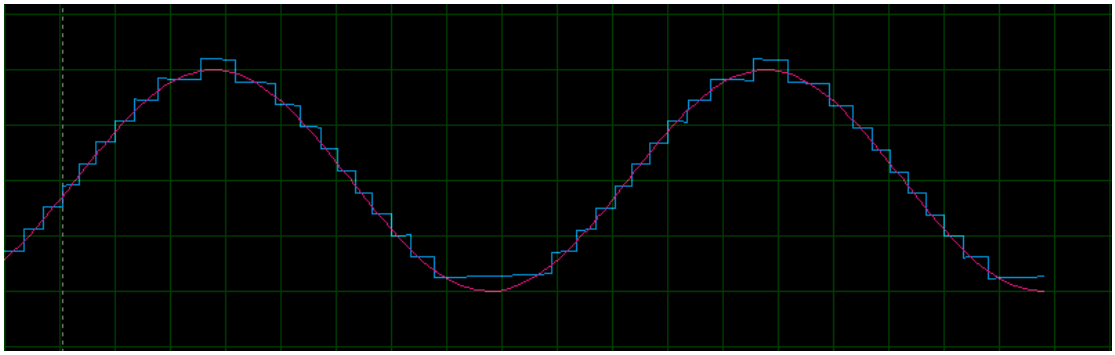

图 44 proteus 消抖滤波效果

消抖滤波可以抑制检测信号的抖动，但个人认为不应该应用到连续的物理量测量，而应该用于类似按键检测的消抖。因为，消抖滤波加大信号延迟，这对于按键检测无关紧要，而对于系统状态量测量最好不要有。而且算法实现复杂。

1.2.11 限幅消抖滤波

```
function [data_filtered] = LimitingGlitch_Filter(data_raw, limit_threshold, buffer_size)%限幅消抖滤波
    data_length = length(data_raw);
    data_filtered = zeros(1,data_length);
    data_filtered(1) = data_raw(1);
    data_value = data_raw(1);
    buffer_counter = 0;
    for i = 2:data_length
        data_filtered(i) = data_raw(i);
        if abs(data_raw(i) - data_filtered(i-1)) > limit_threshold
            data_filtered(i) = data_filtered(i-1);
        end
        if data_filtered(i) ~= data_value
            buffer_counter = buffer_counter + 1;
            if buffer_counter >= buffer_size
                data_value = data_filtered(i);
                buffer_counter = 0;
            end
        end
        data_filtered(i) = data_value;
    end
end
```

图 45 限幅消抖滤波 matlab 实现

```c
uchar LimitGlitch_Filter(uchar data_raw, uchar buffer_size, uchar limit_threshold)//限幅消抖滤波
{
  static uchar data_buff;
  static uchar income_times;
  static uchar data_last;
  static uchar diff_times;
  uchar data_temp;
  uchar data_now;

  if(income_times<buffer_size)
  {
    data_now = data_raw;
    data_buff = data_now;
  }
  else
  {
    if(abs(data_now - data_last)>limit_threshold)
    {
      data_now = data_last;
      if(data_now!=data_buff)
      {
        if(++diff_times>buffer_size)
        {
          diff_times = 0;
          data_buff = data_raw;
        }
      }
    }
  }

  data_temp = data_buff;
  data_last = data_temp;

  return data_temp;
}
```

图 46 限幅消抖滤波 C 语言实现

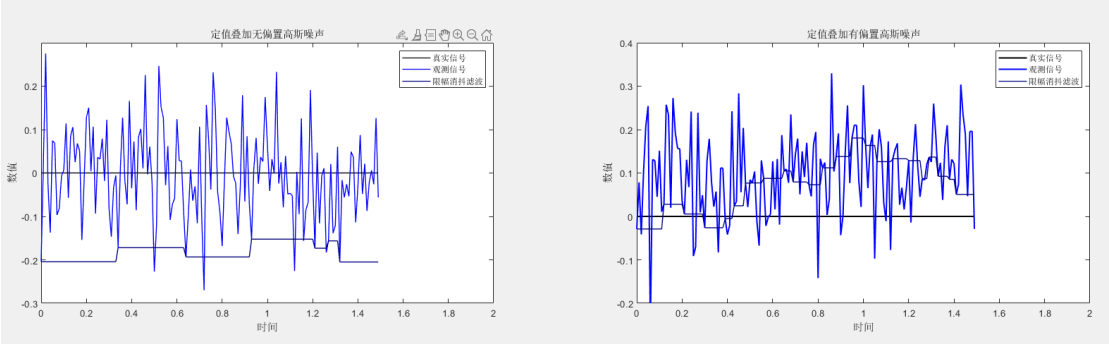

图 47 matlab 限幅消抖滤波效果

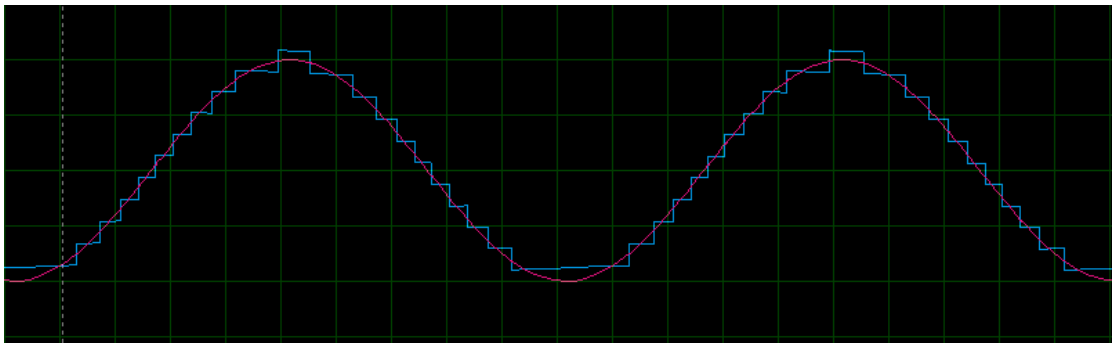

图 48 proteus 限幅消抖滤波效果

效果也不错，但结果不平滑。

1.2.12 卡尔曼滤波

```
clear;
clc;
%生成一段时间t
t = 0:0.01:1;
L = length(t);

%生成真实信号x，和观测信号y
%初始化
x = zeros(1,L);
y = x;
%生成信号:x = t^2 , y = x + N(0,0.1)
for i = 1:L
    x(i) = t(i)^2;
    y(i) = x(i) + normrnd(0,0.1);
end
% plot(t,x,t,y,'LineWidth',2);

%滤波算法
%建立模型
%注意，卡尔曼滤波的模型都是线性模型，其中的噪声都是正态分布的

%根据观测数据，进行粗略建模1
%X(K) = X(K-1) + Q
%Y(K) = X(K) + R
%Q,R~N(0,1)
F1 = 1;
H1 = 1;
Q1 = 0.05;
R1 = 1;
%先验数据:X均值，P方差
Xminus1 = zeros(1,L);
Pminus1 = zeros(1,L);
%后验数据:X均值，P方差
Xplus1 = zeros(1,L);
Xplus1(1) = 1;
Pplus1 = zeros(1,L);
Pplus1(1) = 0.01^2;
%算法主体 plus后验，minus先验
%X(K)minus = F*X(K-1)plus
%P(K)minus = F*P(K-1)*F' + Q
%K = P(K)minus*H'*inv(H*P(K)minus*H' + R)
%X(K)plus = X(K)minus + K*(y(K) - H*X(K)minus)
%P(K)plus = (1 - K*H)*P(K)minus
for k=2:L
    Xminus1(k) = F1*Xplus1(k-1);
    Pminus1(k) = F1*Pplus1(k-1)*F1' + Q1;
    K1 = Pminus1(k)*H1'/(H1*Pminus1(k)*H1' + R1);
    Xplus1(k) = Xminus1(k) + K1*(y(k) - H1*Xminus1(k));
    Pplus1(k) = (1 - K1*H1)*Pminus1(k);
end

%乘勒展开提高维数，建模2
%X(K) = X(K-1) + X'(K-1)*dt + 0.5*X''(K-1)dt^2 + Q0
%X'(K) = X'(K-1) + X''(K-1)dt + Q1
%X''(K) = X''(K-1) + Q2
%Y(K) = X(K) + R
%Q,R~N(0,1)
dt = t(2)-t(1);
F2 = [1,dt,0.5*dt^2;
      0,1,dt;
      0,0,1];
H2 = [1,0,0];
Q2 = [1,0,0;
     0,0.01,0;
     0,0,0.001];
R2 = 0.1;
%先验数据:X均值，P方差
Xminus2 = zeros(3,L);
Pminus2 = zeros(3,3*L);
%后验数据:X均值，P方差
Xplus2 = zeros(3,L);
Xplus2(:,1) = [1;0;0];
Pplus2 = zeros(3,3*L);
Pplus2(1:3,1:3) = [0.01,0,0;
                   0,0.01,0;
                   0,0,0.01^2];
for k = 2:L
    Xminus2(:,k) = F2* Xplus2(:,k-1);
    Pminus2(:,3*k-2:3*k) = F2*Pplus2(:,3*(k-1)-2:3*(k-1))*F2' + Q2;
    K2 = Pminus2(:,3*k-2:3*k)*H2'/(H2*Pminus2(:,3*k-2:3*k)*H2' + R2);
    Xplus2(:,k) = Xminus2(:,k) + K2*(y(k) - H2*Xminus2(:,k));
    Pplus2(:,3*k-2:3*k) = (eye(3) - K2*H2)*Pminus2(:,3*k-2:3*k);
end

plot(t,x,'k');
hold on;
plot(t,y,'b');
hold on;
plot(t,Xplus1,'g');
hold on;
plot(t,Xplus2(1,:),'r');
```

卡尔曼滤波 matlab 实现

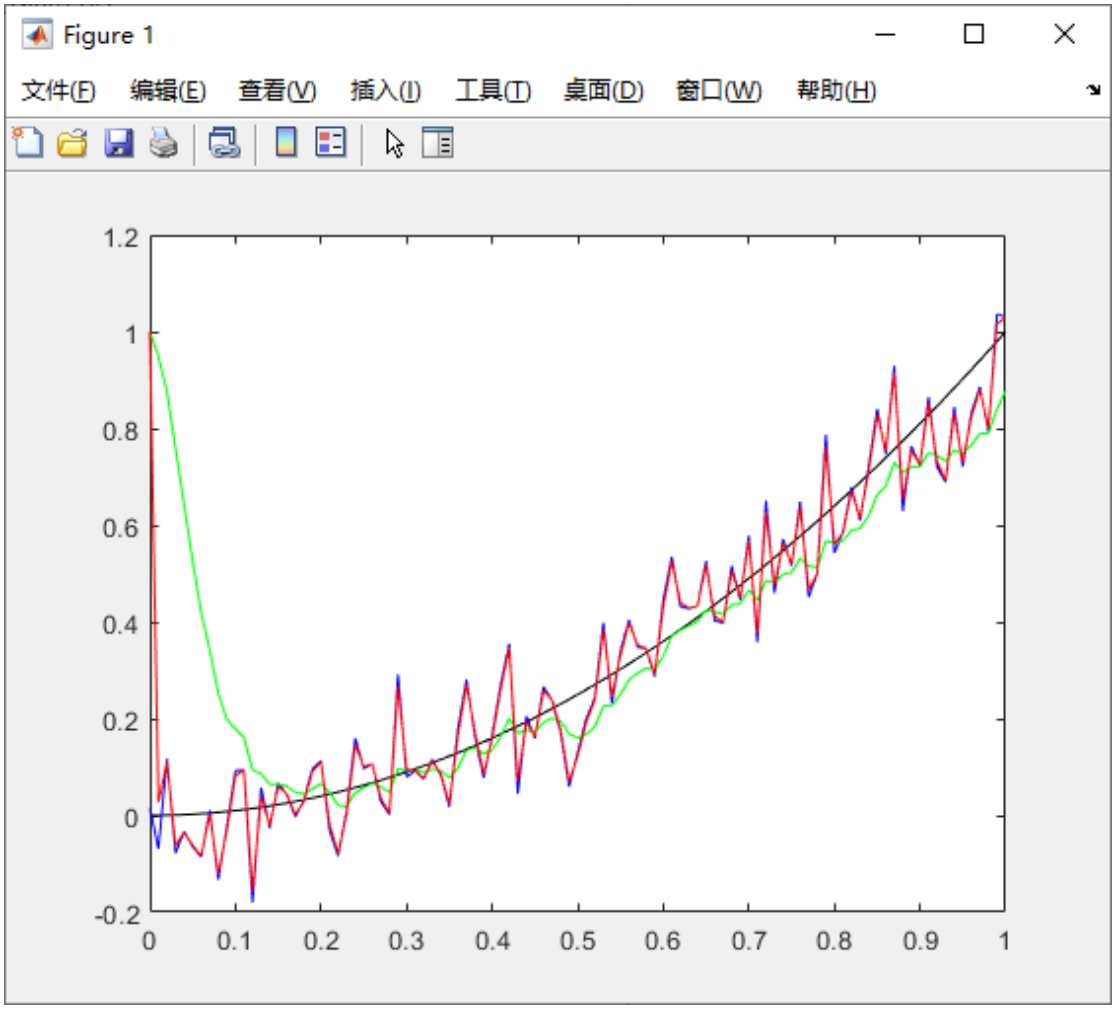

图 49 matlab 卡尔曼滤波效果

如果是有基本建模的线性系统，用卡尔曼滤波可以理论上实现对信号在高斯噪声干扰下的无偏估计。

个人总结：

本次课程设计我承担了滤波编程环节的全部工作，和其他环节的部分工作。我一直对数字信号处理部分很感兴趣。这个兴趣最早来源于一个竞赛的培训练习经历。那次培训中，我和队友要做出一个单片机温度控制系统，但由于我们的硬件水平不好，信号总是存在不明干扰，非常影响我们的系统工作。白天我们试图排查硬件问题无果，晚上我气不过，把所有程序重构了一遍，并且决定采用软件的方法，把干扰尽量都滤掉。我当时运用限幅滤波和滑动平均滤波将干扰基本过滤干净，系统表现非常好。当时我并不知道这些算法的名字，就是感觉着写出来的，毕竟他们的思想还比较简单。

这次课程设计让我重新审视了所有常见的滤波算法，并通过实践，加深了对他们的认识。相信下次遇到干扰问题，我能更快地选择出合适的滤波算法，并实现它。

由于 51 单片机硬件资源有限（我已经把 ram 写超了，把一些量定义为 xdata 才编译通过），我没有把我所有想探索的状态估计算法都实现。比如卡尔曼滤波和扩展卡尔曼滤波我只在 matlab 上进行了实验。此外还有粒子滤波、非线性观测器等我曾用过的方法都还没能实现。也是不小的遗憾。

我会把探索滤波算法作为一个感兴趣的支线任务，慢慢推进，完成他们的数学证明、

matlab 实现、单片机实现。我想这会是非常有意义的过程。

参考文献

[1] 廖道争，施保华. 计算机控制技术[M]. 北京：机械工业出版社，2022