Q1. 例题 6.1-3 提出了希尔伯特矩阵是严重病态矩阵。那么请同学们通过查询数值分析课本或上网搜索，回答矩阵条件数的定义，以及高阶矩阵条件数高的“病态”表现（写一个方面的表现即可）

此外，请利用例题的代码，完成 1 到 50 阶希尔伯特矩阵的条件数计算，并将条件数通过 plot 的形式展示出来。（在作业中写上实现的代码即可）。在完成作业的过程中，请思考 MATLAB 函数 cond 获取的条件数是否准确？还有什么其他更好的方法？

答：矩阵的条件数通常适用于可逆方阵，其一般定义为矩阵的范数乘以矩阵逆的范数，即

$$\mathbf{cond}(\boldsymbol{A}) = \|\boldsymbol{A}\| \cdot \left\|\boldsymbol{A}^{-1}\right\|$$

MATLAB 默认的条件数设定，为基于矩阵 2-范数的条件数，其等价于

$$\mathbf{cond}_2(\boldsymbol{A}) = \frac{\boldsymbol{\sigma}_{\max}(\boldsymbol{A})}{\boldsymbol{\sigma}_{\min}(\boldsymbol{A})}$$

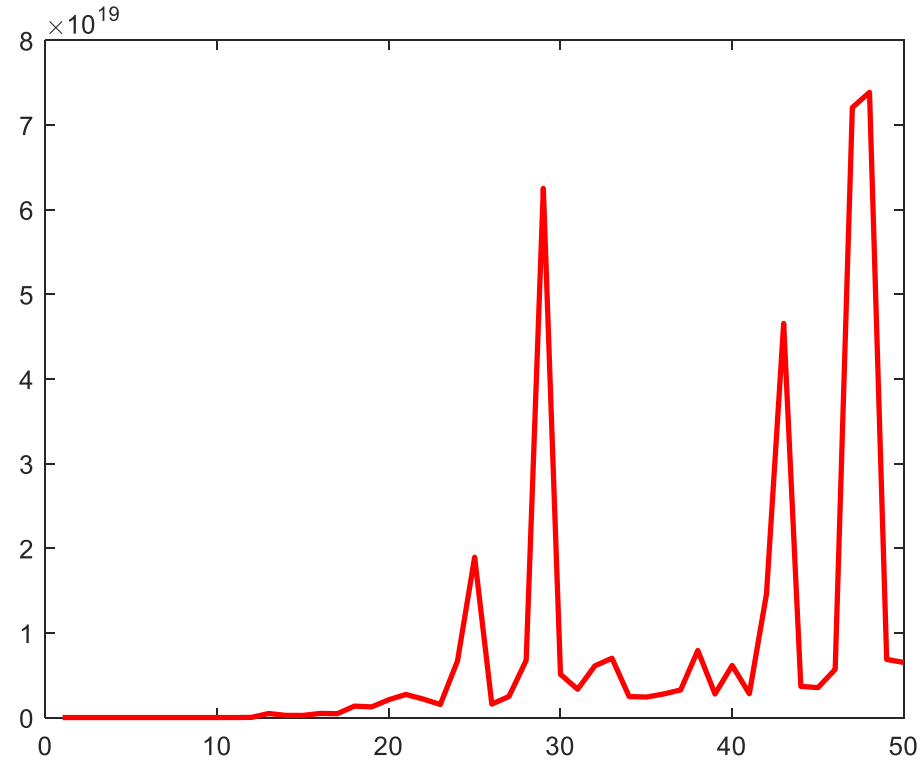
注意到因为A的可逆性，A的奇异值σ均为正值，最大奇异值与最小奇异值的比值即为基于 2-范数定义的矩阵条件数。对于实际问题中的高阶矩阵，当条件数非常大（大于1×10^{10}）时，称矩阵A为病态矩阵。

病态矩阵A容易影响线性反问题$\boldsymbol{Ax} = \boldsymbol{b}$的适定性，很小的问题误差就可能会导致很大的解的误差。甚至有可能导致很多数值方法无法收敛（如共轭梯度法）从而根本得不到线性反问题的解。

然而，病态矩阵在很多的实际应用数学问题中却是普遍存在的，并不是因为模型或信息获取的错误而导致的。病态矩阵的解决方法有很多种，奇异值修正或各种基于正则化的方法都是可行的。

以下为希尔伯特矩阵条件数计算及绘图代码，本题如果使用了两层或更多的 for 循环，且没有在生成矩阵前用 zeros 预先进行空间申请将被扣分。使用 MATLAB 函数 hilb 不扣分但不推荐。并且，不同阶的希尔伯特矩阵具有包含关系，精炼的代码是无需反复生成所有阶的希尔伯特矩阵的。（如果反复重新生成希尔伯特矩阵,时间将多花一些）

```
tic;
cond_n = zeros(1,50);
N = 50;
n=repmat(1:N,N,1);
m=n';
A=1./(n+m-1);
for i = 1:50
    cond_n(i) = cond(A(1:i,1:i));
end
plot(1:50,cond_n,'-r','LineWidth',2);
toc;
```



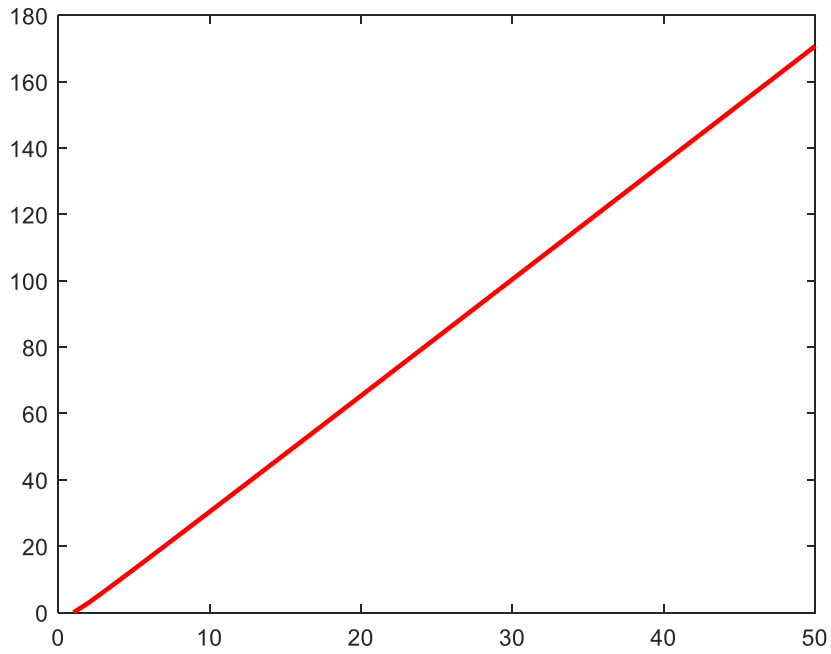
从此图可以看到，当阶数大于 20 后，条件数时大时小，但总体都超过了10^{18}的数量级，属于病态矩阵的范畴。

但是当希尔伯特矩阵阶较大时，已是高度病态矩阵，奇异值准确度下降很大，cond 函数的精确程度也将大打折扣，即使是另外两种条件数同样受到了病态矩阵的影响和侵扰。在符号运算效率偏低的情况下，有一种解决方案是基于希尔伯特逆矩阵的显式定义来进行计算。希尔伯特逆矩阵被证明等于：

$$(A^{-1})_{ij} = (-1)^{i+j}(i+j-1)\binom{n+i-1}{n-j}\binom{n+j-1}{n-i}\binom{i+j-1}{i-1}^2$$

我们也可以使用 MATLAB 函数 invhilb 来代入其逆矩阵，分别计算 2-范数解决问题。取对数后，增长几乎为线性，与理论分析完全一致（指数级增长）

```
tic;
N = 50;
cond_n = zeros(1,N);
for i = 1:N
    cond_n(i) = norm(hilb(i))*norm(invhilb(i));
end
plot(1:N,log(cond_n),'-r','LineWidth',2);
toc;
```



Q2. 某年春节，数院学子 D 神希望为朋友圈点赞的朋友发红包。他表示，点赞序数为素数的朋友将获得红包。但点赞朋友共有 300 人，D 神发现逐个手算这些素数很费时间，而且容易出错。

现需要你设计算法效率尽量高的一个 MATLAB 程序，找出 1~300 的所有素数，并把它们存在一个向量中。

请在作业纸上写上你的算法思路、MATLAB 代码，以及你所找出的素数列表。需要注意的是，MATLAB 代码要尽可能降低算法复杂度，并适当利用向量化操作避免不必要的 for 循环的使用。

答：计算 1~300 内的素数，效率比较低的算法可能会有$O(n^2)$的算法复杂度，即每个数都将比自己小的数进行整除测试。这种代码的效率会比较低。使用函数 `primes` 或 `isprime` 都相当于没有自主实现，视为与此复杂度相同。（其实 `primes` 复杂度是$O(n\log(\log\sqrt{n}))$，循环内包括 `isprime` 的算法复杂度是$O(n\log\sqrt{n})$）

一种很直接的改良是判断n是否是素数时，以n作为被除数，只需要尝试去除$2\sim\lfloor\sqrt{n}\rfloor$就可以了（括号为下取整号），这样可以将算法复杂度降低至$O(n^{1.5})$，尝试除的数甚至可以进一步改良到所有$2\sim\lfloor\sqrt{n}\rfloor$的素数作为除数即可，算法复杂度可以进一步降低至$O(n\log\sqrt{n})$。按照类似 C++的编码思路，可以写出如下的一组代码：

```
clear
prime_num = [];      %初始化素数集合为空集
for i = 2:300
    flag = 1;        % flag = 1表示这个数仍然可能是素数
```

```
    for j = prime_num %只需要按照当前的素数集和来做循环
        if(j > sqrt(i))% 并不需要除大于根号i的数
            break;
        end
        if(mod(i,j)==0)% 若出现整除则i不是素数
            flag = 0;
            break;
        end
    end
    if(flag == 1)%如果i是素数则把它增加到素数列表
        prime_num = [prime_num,i];
    end
end
prime_num
```

程序运行结果共找到了 62 个素数。

```
prime_num =
  Columns 1 through 21
     2     3     5     7    11    13    17    19    23    29    31    37
41    43    47    53    59    61    67    71    73

  Columns 22 through 42
    79    83    89    97   101   103   107   109   113   127   131   137
139   149   151   157   163   167   173   179   181

  Columns 43 through 62
   191   193   197   199   211   223   227   229   233   239   241
251   257   263   269   271   277   281   283   293
```

那么下面推荐更加高效的 Eratosthenes 筛法。思路即将 2~300 存在一个集合中，而后将所有$1\sim\sqrt{300}$之间的素数的倍数（不包含 1 倍）“筛掉”，剩下的就是 2~300 之间的素数。因此因为$\lfloor\sqrt{300}\rfloor=17$，因此我们需要筛掉的包括 2、3、5、7、11、13、17 的所有倍数。这个代码的算法复杂度也是$O(n\log\sqrt{n})$（甚至可以认为是$O(n\log(\log\sqrt{n}))$），代码如下：

```
prime_flag = ones(1,300);
%一个向量记录是否为素数，1代表素数，初始值均设为1
prime_flag(1) = 0; % 1不是素数
prime_divisor = [2,3,5,7,11,13,17]; %小于等于根号300的所有素数
for i = prime_divisor
    prime_flag(i*i:i:end) = 0; %筛选掉所有i的i倍或以上的倍数
end
prime_num = find(prime_flag==1) %找到所有“仍是素数”的数
```

这个代码的运行结果与刚才的代码完全相同。在老师的笔记本电脑上，第一个代码的运行时间是 0.000765 秒，而第二个代码的运行时间为 0.000241 秒。向量化操作和更高效的算法大大的节约了算法运行的时间。