1. (a)

$$\begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} \\ 0 & \frac{1}{6} & \frac{1}{6} \\ 0 & 0 & \frac{1}{60} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \\ \frac{1}{2} \end{bmatrix}$$

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 9 \\ -36 \\ 30 \end{bmatrix}$$

(b)

$$\begin{bmatrix} 1.0 & \frac{1}{2} & \frac{1}{3} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \end{bmatrix} = \begin{bmatrix} 1 & 0.5 & 0.33 \\ 0.50 & 0.33 & 0.25 \\ 0.33 & 0.25 & 0.20 \end{bmatrix} = \begin{bmatrix} 1.0\times10^{0} & 5.0\times10^{-1} & 3.3\times10^{-1} \\ 5.0\times10^{-1} & 3.3\times10^{-1} & 2.5\times10^{-1} \\ 3.3\times10^{-1} & 2.5\times10^{-1} & 2.0\times10^{-1} \end{bmatrix}$$

$$\begin{bmatrix} 1.0\times10^{1} \\ 0.0 \\ 0.0 \end{bmatrix}$$

(c) Using two decimal-digit chopped arithmetic:

$$\begin{bmatrix} 1.0\times10^{0} & 5.0\times10^{-1} & 3.3\times10^{-1} \\ 5.0\times10^{-1} & 3.3\times10^{-1} & 2.5\times10^{-1} \\ 3.3\times10^{-1} & 2.5\times10^{-1} & 2.0\times10^{-1} \end{bmatrix}$$

$$\begin{bmatrix} 1.0\times10^{1} \\ 0.0 \\ 0.0 \end{bmatrix}$$

elimination on first column:

$$M_1A = \begin{bmatrix} 1.0\times10^{0} & 0.0 & 0.0 \\ -5.0\times10^{-1} & 1.0\times10^{0} & 0.0 \\ -3.3\times10^{-1} & 0.0 & 1.0\times10^{0} \end{bmatrix} \begin{bmatrix} 1.0\times10^{0} & 5.0\times10^{-1} & 3.3\times10^{-1} \\ 5.0\times10^{-1} & 3.3\times10^{-1} & 2.5\times10^{-1} \\ 3.3\times10^{-1} & 2.5\times10^{-1} & 2.0\times10^{-1} \end{bmatrix}$$

$$= \begin{bmatrix} 1.0\times10^{0} & 5.0\times10^{-1} & 3.3\times10^{-1} \\ 0.0 & 8.0\times10^{-2} & 9.0\times10^{-2} \\ 0.0 & 9.0\times10^{-2} & 1.0\times10^{-1} \end{bmatrix}$$

$$M_1b = \begin{bmatrix} 1.0\times10^{0} & 0.0 & 0.0 \\ -5.0\times10^{-1} & 1.0\times10^{0} & 0.0 \\ -3.3\times10^{-1} & 0.0 & 1.0\times10^{0} \end{bmatrix} \begin{bmatrix} 1.0\times10^{1} \\ 0.0 \\ 0.0 \end{bmatrix} = \begin{bmatrix} 1.0\times10^{1} \\ -5.0\times10^{-1} \\ -3.3\times10^{-1} \end{bmatrix}$$

elimination on second column:

$$M_2M_1A = \begin{bmatrix} 1.0\times10^{0} & 0.0 & 0.0 \\ 0.0 & 1.0\times10^{0} & 0.0 \\ 0.0 & -1.1\times10^{0}\ (\frac{9}{8}) & 1.0\times10^{0} \end{bmatrix} \begin{bmatrix} 1.0\times10^{0} & 5.0\times10^{-1} & 3.3\times10^{-1} \\ 0.0 & 8.0\times10^{-2} & 9.0\times10^{-2} \\ 0.0 & 9.0\times10^{-2} & 1.0\times10^{-1} \end{bmatrix}$$

$$= \begin{bmatrix} 1.0\times10^{0} & 5.0\times10^{-1} & 3.3\times10^{-1} \\ 0.0 & 8.0\times10^{-2} & 9.0\times10^{-2} \\ 0.0 & 0.0 & 1.0\times10^{-3} \end{bmatrix}$$

$$M_2M_1b = \begin{bmatrix} 1.0\times10^{0} & 0.0 & 0.0 \\ 0.0 & 1.0\times10^{0} & 0.0 \\ 0.0 & -1.1\times10^{0}\ (\frac{9}{8}) & 1.0\times10^{0} \end{bmatrix} \begin{bmatrix} 1.0\times10^{1} \\ -5.0\times10^{-1} \\ -3.3\times10^{-1} \end{bmatrix} = \begin{bmatrix} 1.0\times10^{1} \\ -5.0\times10^{-1} \\ 2.2\times10^{-1} \end{bmatrix}$$

Hence, the solution is

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3.7\times10^{1} \\ -2.3\times10^{2} \\ 2.2\times10^{2} \end{bmatrix}$$

(d) Using exact arithmetic:

$$\begin{bmatrix} 1 & 0.5 & 0.33 \\ 0.5 & 0.33 & 0.25 \\ 0.33 & 0.25 & 0.2 \end{bmatrix}$$

$$\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

elimination on first column:

$$M_1A = \begin{bmatrix} 1 & 0 & 0 \\ -5.0 & 1 & 0 \\ -3.3 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0.5 & 0.33 \\ 0.5 & 0.33 & 0.25 \\ 0.33 & 0.25 & 0.2 \end{bmatrix} = \begin{bmatrix} 1.0 & 0.5.0 & 0.33 \\ 0 & 0.08 & 0.085 \\ 0 & 0.085 & 0.0911 \end{bmatrix}$$

$$M_1b = \begin{bmatrix} 1 & 0 & 0 \\ -5.0 & 1 & 0 \\ -3.3 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ -0.5 \\ -0.33 \end{bmatrix}$$

elimination on second column:

$$M_2M_1A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & \frac{9}{8} & 1 \end{bmatrix} \begin{bmatrix} 1.0 & 0.5.0 & 0.33 \\ 0 & 0.08 & 0.085 \\ 0 & 0.085 & 0.0911 \end{bmatrix} = \begin{bmatrix} 1 & 0.5 & 0.33 \\ 0 & 0.08 & 0.085 \\ 0 & 0.0 & 0.0007875 \end{bmatrix}$$

$$M_2M_1b = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & \frac{9}{8} & 1 \end{bmatrix} \begin{bmatrix} 1 \\ -0.5 \\ -0.33 \end{bmatrix} = \begin{bmatrix} 1 \\ -0.5 \\ 0.20125 \end{bmatrix}$$

Then, we get result,

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 500/9 \\ -2500/9 \\ 2300/9 \end{bmatrix}$$

(e) From the previous calculation we can see that Precision (t) in floating point arithmetic has a great effect of the accuracy of a algorithm.
In c and d, we are using the same algorithm, but since in c we only have 2 digits of precision, then results is different from d.
Also, In a and d even both are using exact arithmetic but since in d the matrix is from two decimal-digit chopping, then we lost a lot in accuracy compare with a.

2. In order to compute:

$$x = B^{-1}(C^{-1} + A)(2A + I)\omega$$

We can rewrite it as:

$$Bx = (C^{-1} + A)(2A + I)\omega$$

$$CBx = C(C^{-1} + A)(2A + I)\omega$$

$$CBx = (I + CA)(2A + I)\omega$$

Clearly, CB is a $n \times n$ matrix and $(I + CA)(2A + I)\omega$ is $n - vector$, so just solve the linear equation for x.

1 calculate $\omega' = I\omega + 2A\omega$, which takes $2n^2 + n^2 + 2n^2 + n = 5n^2 + n$;

2 calculate $b = I\omega' + CA\omega'$, which takes $2n^2 + 2n^2 + 2n^2 + n = 6n^2 + n$;

3 use Gauss elimination to factor C and B into $P_B B = L_B U_B$ and $P_C C = L_C U_C$, which takes $2(\frac{2}{3}n^3 + \Theta(n^2)) = \frac{4}{3}n^3 + \Theta(n^2)$;

4 use forward solve to solve $L_C y_1 = P_C b$ for y_1, which takes $n^2 + \Theta(n)$;

5 use backward solve to solve $U_C y_2 = y_1$ for y_2, which takes $n^2 + \Theta(n)$;

6 use forward solve to solve $L_B y_3 = P_B y_2$ for y_3, which takes $n^2 + \Theta(n)$;

7 use backward solve to solve $U_B x = y_3$ for x, which takes $n^2 + \Theta(n)$.

Hence, this algorithm cost $\frac{4}{3}n^3 + \Theta(n^2)$ flops.

3. (a) Gauss elimination without pivoting:

```python
def noGauss(A,b):
    """ Gauss elimination with no pivoting

    A: nxn matirx with form [[row(1)],[row(2)]...[row(n)]]
    b: n-vector with form [b(1),b(2)...b(n)]

    return x: the solution for the linear system eqution Ax = b.
    """
    n = len(A)
    #iteration times
    iteration = n - 1
    for i in range(0, iteration):
        # elimination on kth row
        for k in range(i+1, n):
            m = -A[k][i]/A[i][i]
            # Set all entrys right below (in the current colnum) A[i][i] to 0
            A[k][i] = 0.0
            # update b
            b[k] += m*b[i]
            for j in range(i+1, n):
                A[k][j] += m * A[i][j]
    # Solve equation Ax=b for an upper triangular matrix A
    x = [0.0 for i in range(n)]
    for i in range(n-1, -1, -1):
        x[i] = b[i]/A[i][i]
        for k in range(i-1, -1, -1):
            b[k] -= A[k][i] * x[i]
    return x
```

Gauss elimination with partial pivoting:

```python
def partialGauss(A,b):
    """ Gauss elimination with partial pivoting

    A: nxn matirx with form [[row(1)],[row(2)]...[row(n)]]
    b: n-vector with form [b(1),b(2)...b(n)]

    return x: the solution for the linear system eqution Ax = b.
    """
    n = len(A)
    iteration = n - 1
    for i in range(0, iteration):
    # Search for maximum in this column
        pivot = abs(A[i][i])
        maxRow = i
        for k in range(i+1, n):
            if abs(A[k][i]) > pivot:
                pivot = abs(A[k][i])
                maxRow = k
        # Swap maximum row with current row
        if i != maxRow:
            A[i], A[k] = A[k], A[i]
            b[i], b[k] = b[k], b[i]
        # Set all entrys right below (in the current colnum) A[i][i] to 0
        for k in range(i+1, n):
            m = -A[k][i]/A[i][i]
            A[k][i] = 0.0
            b[k] += m*b[i]
            for j in range(i+1, n):
                A[k][j] += m * A[i][j]
    # Solve equation Ax=b for an upper triangular matrix A
    x = [0.0 for i in range(n)]
    for i in range(n-1, -1, -1):
        x[i] = b[i]/A[i][i]
        for k in range(i-1, -1, -1):
            b[k] -= A[k][i] * x[i]
    return x
```

Gauss elimination complete pivoting:

```python
def completeGauss(A,b):
    """ Gauss elimination with complete pivoting

    A: nxn matirx with form [[row(1)],[row(2)]...[row(n)]]
    b: n-vector with form [b(1),b(2)...b(n)]

    return x: the solution for the linear system eqution Ax = b.
    """
    n = len(A)
    iteration = n - 1
    # r indicate the order of each of x(i)s
    x = [[r,0.0] for r in range(n)]
    for i in range(iteration):
    # Search for maximum in submatirx
        pivot = abs(A[i][i])
        maxRow = i
        maxCol = i
        for row in range(i, n):
            for col in range(i, n):
                if abs(A[row][col]) > pivot:
                    pivot = abs(A[row][col])
                    maxRow = row
                    maxCol = col
        if i != maxRow or i!= maxCol:
            # Swap maximum row with current row
            A[i], A[maxRow] = A[maxRow], A[i]
            b[i], b[maxRow] = b[maxRow], b[i]
            # swap the order of x
            x[i], x[maxCol] = x[maxCol], x[i]
            # Swap maximum col with current col for A
            for row in range(n):
                A[row][i], A[row][maxCol] = A[row][maxCol], A[row][i]
        # Set all entrys right below (in the current colnum) A[i][i] to 0
        for k in range(i+1, n):
            m = -A[k][i]/A[i][i]
            A[k][i] = 0.0
            b[k] += m*b[i]
            for j in range(i+1, n):
                A[k][j] += m * A[i][j]
    # Solve equation Ax=b for an upper triangular matrix A
    for i in range(n-1, -1, -1):
        x[i][1] = b[i]/A[i][i]
        for k in range(i-1, -1, -1):
            b[k] -= A[k][i] * x[i][1]
    # reorder x(i)s
    result = [ 0.0 for i in range(n)]
    for i in range(n):
        result[x[i][0]] = x[i][1]
    return result
```

(b) The function below generates random linear system equation $Ax = b$, with correct solution x that $x_i = (-1)^{i+1}$

```
def generator(n):
    A = [[random.random() for i in range(n)] for j in range(n)]
    b = [0.0 for i in range(n)]
    for row in range(n):
        for col in range(n):
            b[row] = b[row] + float(pow(-1,col+2)*A[row][col])
    return A,b
```

Then we randomly generates matrix of size from 1 to 9, the following graph shows the solutions using no pivoting, partial pivoting, complete pivoting.

dimension	NO PIVOTING relative error	ROW PIVOTING relative error	COMPLETE PIVOTING relative error
Random matrices:			
66	1.0292e-13	9.7372e-14	2.4855e-15
56	6.0984e-13	8.8765e-14	3.2462e-15
98	2.8458e-12	1.8105e-13	1.7243e-14
12	5.4707e-15	6.0783e-15	9.8887e-16
129	5.0734e-12	1.5656e-12	1.1226e-14

Observing the results, we can get that generally complete pivoting gives the best accuracy. Partial pivoting and no pivoting has close performance, but generally partial pivoting is better

(c) If the matrix A has the form

$$\begin{bmatrix} a_1 & 0 & 0 & \dots & 0 & b_1 \\ -a_1 & a_2 & 0 & \dots & 0 & b_2 \\ -a_1 & -a_2 & a_3 & \dots & 0 & b_3 \\ -a_1 & -a_2 & -a_3 & \dots & 0 & b_4 \\ & & \dots & & & \\ & & & \dots & & \\ -a_1 & -a_2 & -a_3 & \dots & a_{n-1} & b_{n-1} \\ -a_1 & -a_2 & -a_3 & \dots & -a_{n-1} & b_n \end{bmatrix}$$

where $a_1, a_2, \dots a_{n_1}$ and $b_1, b_2 \dots b_n$ are all positive and b_i $(0 < i \le n)$ are all bigger than 1 and a_i $(0 < i < n)$ are relatively small. Then as size of matrix A grows bigger complete pivoting is significantly more accurate.

Now, suppose $a_1 = a_2 = \dots = a_{n_1} = 1$ and $b_1 = b_2 = \dots = b_n = 1$, and $Ax = b$ where exact solution is $x_i = (-1)^{i+1}$.

$$\begin{bmatrix} 1 & 0 & 0 & \dots & 0 & 1 \\ -1 & 1 & 0 & \dots & 0 & 1 \\ -1 & -1 & 1 & \dots & 0 & 1 \\ -1 & -1 & -1 & \dots & 0 & 1 \\ & & \dots & & & \\ & & & \dots & & \\ -1 & -1 & -1 & \dots & 1 & 1 \\ -1 & -1 & -1 & \dots & -1 & 1 \end{bmatrix}$$

No we use no pivoting, partial pivoting and complete pivoting compute $Ax = b$.

dimension	NO PIVOTING relative error	ROW PIVOTING relative error	COMPLETE PIVOTING relative error
problem 2.7:			
60	3.163e-01	3.163e-01	4.438e-16
71	4.747e-01	4.747e-04	2.363e-15

4. (a) We know that

$$\frac{\|\hat{x}-x\|_\infty}{\|x\|_\infty} \approx cond(A)\epsilon_{mach}$$

Then,

$$\frac{\|\hat{x}-x\|_\infty}{\|x\|_\infty} \approx 10^{-8}$$

Since $\|x\|_\infty = 1.234567890123456789 \times 10^1$, then $\|\hat{x}-x\|_\infty \leq 1.234567890123456789 \times 10^{-7}$.
Then $\hat{x}$ will have probably up to 10^{-5} or 10^{-6}accuracy. Then those digits

$$\begin{bmatrix} \underline{12.34567}\,8901234567890 \\ \underline{0.00123}\,4567890123456 \end{bmatrix}$$

$$\begin{bmatrix} \underline{12.345678}\,901234567890 \\ \underline{0.001234}\,567890123456 \end{bmatrix}$$

are probability agree.

(b) We know that

$$\frac{\|\hat{x}-x\|}{\|x\|} \approx cond(A)\epsilon_{mach}$$

We want the smallest component of $x = A\setminus b$ has at least six significant digits of accuracy, and we the the smallest component has exponent 10^{-2}. Hence, $\|\hat{x}-x\|$ can be at most 10^{-8}, but to ensure $\|\hat{x}-x\|$ be at most 10^{-8}, $\frac{\|\hat{x}-x\|_\infty}{\|x\|_\infty}$ can be at most 10^{-13}. Therefore, machine precision must be at most 10^{-19}.

5.

```
function [] = generator()

    diary result.out
    % format output
    disp(sprintf('iteration      reltive error      relative residual      condintion        determinant\n'))
    iteration = 1;
    [reltiveError,relativeResidual,condintion,determinant] = gauss(iteration);
    % format output
    formatSpec = '%d               %e          %e      %e      %e \n';
    disp(sprintf(formatSpec,iteration,reltiveError,relativeResidual,condintion,determinant))
    iteration = iteration + 1;
    % stop when the relative error in the computed solution is greater than 1
    while reltiveError <= 1
        [reltiveError,relativeResidual,condintion,determinant] = gauss(iteration);
        % format output
        disp(sprintf(formatSpec,iteration,reltiveError,relativeResidual,condintion,determinant))
        iteration = iteration + 1;
    end
    diary off

function [reltiveError,relativeResidual,condintion,determinant] = gauss(n)

    % intialize A, b
    A = zeros(n,n);
    b = zeros(n,1);
    rel = zeros(n,1);
    for row = 1:1:n
        for col = 1:1:n
            A(row, col) = power(col,row);
            b(row,1) = b(row,1)+power(-1,col+1)*A(row, col);
        end
        rel(row,1) = power(-1, row+1);
    end
    % calculate Ax = b
    x = A\b;
    % calcuate reltive error using 2-norm
    reltiveError = norm(rel-x, 2)/norm(rel,2);
    % calculate relative residual using 2-norm
    relativeResidual = norm(b-A*x, 2)/norm(b,2);
    % calculate conditional number for A
    condintion = cond(A,2);
    % calculate the determinant
    determinant = det(A);
```

iteration	reltive error	relative residual	condintion	determinant
1	0.000000e+00	0.000000e+00	1.000000e+00	1.000000e+00
2	0.000000e+00	0.000000e+00	1.090833e+01	2.000000e+00
3	8.308148e-16	4.364539e-17	1.412356e+02	1.200000e+01
4	7.182208e-15	4.067756e-17	2.501239e+03	2.880000e+02
5	1.900722e-13	1.908230e-17	5.689579e+04	3.456000e+04
6	1.685719e-12	1.031856e-17	1.589237e+06	2.488320e+07
7	2.124984e-10	7.897833e-17	5.284356e+07	1.254113e+11
8	4.413227e-10	2.748799e-17	2.042493e+09	5.056585e+15
9	3.762690e-08	1.045950e-16	9.005988e+10	1.834933e+21
10	1.495754e-06	1.487637e-16	4.462527e+12	6.658606e+27
11	2.122467e-05	3.521864e-17	2.454608e+14	2.657897e+35
12	1.443566e-04	2.876831e-17	1.467902e+16	1.273096e+44
13	1.407079e-01	1.411778e-16	1.032309e+18	7.941555e+53
14	2.131744e+00	1.270362e-16	1.190222e+20	6.798529e+64

Also, this the verification for generating the correct matrix when $n = 3$:

```
1    2    3
1    4    9
1    8   27

2
6
20
```

From the chart above we can see that, if $|det(A)|$ is big and condition number of A is big as well, which means A is ill condition. However, when $|det(A)|$ is closed to 0, A is also ill condition; hence, $cond(A)$ is not always grows as $|det(A)|$ grows. But from the chart we could notice that for $|det(A)| > 1$, conditional number grows as $|det(A)|$ increasing. Therefore, we know that if the determinant is too big or determinant is close to 0, then matrix might be ill condition.