HW1

1. Machine precision

(1).

Code:

```
#include <stdio.h>

// function to find the machine precision for single precision
float machine_precision_single() {
    float m = 1.0f;
    while (1.0f + m != 1.0f) {
        m /= 2.0f;
    }
    return m * 2.0f;
}

// function to find the machine precision for double precision
double machine_precision_double() {
    double m = 1.0;
    while (1.0 + m != 1.0) {
        m /= 2.0;
    }
    return m * 2.0;
}

int main() {
    float single_precision = machine_precision_single();
    double double_precision = machine_precision_double();

    printf("Machine Precision for Single Precision(float): %.10e\n",
single_precision);
    printf("Machine Precision for Double Precision(double): %.10e\n",
double_precision);

    return 0;
}
```

Outputs:

```
Machine Precision for Single Precision(float): 1.1920928955e-07
Machine Precision for Double Precision(double): 2.2204460493e-16
```

(2).

Code:

```
#include <stdio.h>

// function to find the smallest positive single
float min_single() {
    float f_min = 1.0f;
    while (f_min /2.0f != 0.0f) {
        f_min /= 2.0f;
    }
    return f_min;
}

// function to find the smallest positive double
double min_double() {
    double d_min = 1.0;
    while (d_min /2.0 != 0.0) {
        d_min /= 2.0;
    }
    return d_min;
}

int main() {
    float f_min = min_single();
    double d_min = min_double();

    printf("Smallest Positive Number for Single Precision(float):
%.10e\n", f_min);
    printf("Smallest Positive Number for Double Precision(double):
%.10e\n", d_min);

    return 0;
}
```

Outputs:

```
Smallest Positive Number for Single Precision(float): 1.4012984643e-45
Smallest Positive Number for Double Precision(double): 4.9406564584e-324
```

Findings:

The experimentally determined smallest positive number with the magnitude of 1e-45 (in FLOAT) or 1e-324 (in DOUBLE) is much more smaller than the smallest normalized numbers which is with the magnitude of 1e-38 or 1e-308 for each type, because it includes denormalized numbers.

2. Numerical derivative on non-uniform grid

Suppose we have three consecutive grid points x_{i-1}, x_i, x_{i+1} with grid spacing $h_i = x_i - x_{i-1}$ and $h_{i+1} = x_{i+1} - x_i$. Firsr we use Taylor expansions around x_i for $f(x_{i+1})$ and $f(x_{i-1})$: $$ f(x_{i+1})=f(x_i)+h_{i+1}f'(x_i)+\frac{h^2_{i+1}}2f''(x_i)+O(h^3_{i+1}) \
f(x_{i-1})=f(x_i)-h_i f'(x_i)+\frac{h^2_i}2f''(x_i)-O(h^3_i) $$ then we substract the two expansions to cancel second terms to get $f'(x_i)$ or cancel first terms to get $f''(x_i)$: $$
f'(x_i)\approx\frac{h_{i+1}^2f(x_{i-1})+(h_i^2-h_{i+1}^2)f(x_i)-h_i^2f(x_{i+1})}{h_i h_{i+1}(h_i+h_{i+1})} \
f''(x_i)\approx\frac2{h_{i+1}(h_i+h_{i+1})}f(x_{i+1})+\frac2{h_i(h_i+h_{i+1})}f(x_{i-1})-\frac2{h_ih_{i+1}}f(x_i)
$$ when $h_i=h_{i+1}=h$, they will reduce to central differencing formula when x_{i-1}, x_i, x_{i+1} are evenly spaced: $$ f'(x_i) \approx \frac{f(x_{i+1})-f(x_{i-1})}{2h}=\frac{f(x_i+h)-f(x_i-h)}{2h} \
f''(x_i)\approx\frac{f(x_{i+1})-2f(x_i)+f(x_{i-1})}{h^2}=\frac{f(x_i+h)-2f(x_i)+f(x_i-h)}{h^2} $$

3. Numerical integration

Code:

```c
#include <stdio.h>
#include <math.h>

// constants
#define H0 67.0          // Hubble constant in km/s/Mpc
#define OMEGA_M 0.3      // matter density
#define OMEGA_LAMBDA 0.7 // dark energy density
#define C 299792.458     // speed of light in km/s

// function to calculate H0 / H(z)
double H0overHz(double z) {
    return 1/(sqrt(OMEGA_M * pow(1 + z, 3) + OMEGA_LAMBDA));
}

// composite Simpson's rule integration
double simpsons_rule(double (*func)(double), double a, double b, int n) {
    double h = (b - a) / n;
    double integral = func(a) + func(b);

    for (int i = 1; i < n; i++) {
        double x = a + i * h;
        integral += (i % 2 == 0) ? 2 * func(x) : 4 * func(x);
    }

    integral *= h / 3.0;

    return integral;
}

// adaptive Simpson's rule with step increment of 2
double adaptive_simpson(double (*func)(double), double a, double b, double
epsilon, double old_integral, int *steps) {
    int n = 4;
    *steps = n; // store the number of steps
    double integral = simpsons_rule(func, a, b, n);
```

```
    // loop until desired precision is achieved
    while (fabs((integral - old_integral)/integral) > epsilon) {
        n += 2;  // increment by 2
        *steps = n; // store the number of steps
        old_integral = integral; // update full integral to the last
estimate
        integral = simpsons_rule(func, a, b, n);
    }

    return integral;
}

int main() {
    double z_values[] = {1.0, 3.0, 8.2};
    int num_z = sizeof(z_values) / sizeof(z_values[0]);

    for (int i = 0; i < num_z; i++) {
        double z = z_values[i];
        double comoving_distance;
        double epsilon = 1e-4; // relative precision
        int steps = 2; // initial step count

        // perform adaptive integration
        double old_integral = simpsons_rule(H0overHz, 0, z, 2); // start
with 2 intervals
        comoving_distance = (C / H0) * adaptive_simpson(H0overHz, 0, z,
epsilon, old_integral, &steps);

        printf("Comoving distance for z = %.1f: %.6f Mpc, steps needed:
%d\n", z, comoving_distance, steps);
    }

    return 0;
}
```

Outputs:

```
Comoving distance for z = 1.0: 3451.731702 Mpc, steps(intervals) needed: 6
Comoving distance for z = 3.0: 6639.944764 Mpc, steps(intervals) needed:
10
Comoving distance for z = 8.2: 9401.886748 Mpc, steps(intervals) needed:
20
```

4. Hilbert Matrix

Code for (1)~(4):

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

// define whether to use float or double
// uncomment the following line to use single precision (float)
// #define USE_FLOAT

#ifdef USE_FLOAT
    typedef float real;
    #define REAL_FORMAT "%.20f"
    #define DATATYPE "FLOAT"
#else
    typedef double real;
    #define REAL_FORMAT "%.20lf"
    #define DATATYPE "DOUBLE"
#endif

// function to create the Hilbert matrix
real** create_hilbert(int n) {
    real** H = (real**)malloc(n * sizeof(real*));
    for (int i = 0; i < n; i++) {
        H[i] = (real*)malloc(n * sizeof(real));
        for (int j = 0; j < n; j++) {
            H[i][j] = 1.0 / (i + j + 1.0); // H(i,j) = 1/(i+j+1)
        }
    }
    return H;
}

// function for Cholesky decomposition
void cholesky_decomposition(real** A, real** L, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j <= i; j++) {
            real sum = 0.0;
            for (int k = 0; k < j; k++) {
                sum += L[i][k] * L[j][k];
            }
            if (i == j) {
                L[i][j] = sqrt(A[i][i] - sum); // diagonal elements
            } else {
                L[i][j] = (A[i][j] - sum) / L[j][j]; // non-diagonal
elements
            }
        }
    }
}

// function to calculate the b vector
void calculate_b(int n, real* b) {
    for (int i = 0; i < n; i++) {
        b[i] = 0.0;
        for (int j = 1; j <= n; j++) {
```

```
            b[i] += 1.0 / (i + j);
        }
    }
}

// function to solve Ax = b using Cholesky decomposition
void solve_cholesky(real** L, real* b, real* x, int n) {
    real* y = (real*)malloc(n * sizeof(real));

    // solve Ly = b
    for (int i = 0; i < n; i++) {
        real sum = 0.0;
        for (int j = 0; j < i; j++) {
            sum += L[i][j] * y[j];
        }
        y[i] = (b[i] - sum) / L[i][i];
    }

    // solve L^Tx = y
    for (int i = n - 1; i >= 0; i--) {
        real sum = 0.0;
        for (int j = i + 1; j < n; j++) {
            sum += L[j][i] * x[j];
        }
        x[i] = (y[i] - sum) / L[i][i];
    }

    free(y);
}

// function to solve the Hilbert matrix equation Ax = b
void solve_hilbert(int n, real* x) {
    real** H = create_hilbert(n);
    real* b = (real*)malloc(n * sizeof(real));
    calculate_b(n, b);

    real** L = (real**)malloc(n * sizeof(real*));
    for (int i = 0; i < n; i++) {
        L[i] = (real*)malloc(n * sizeof(real));
    }

    cholesky_decomposition(H, L, n);
    solve_cholesky(L, b, x, n);

    // free allocated memory
    free(b);
    for (int i = 0; i < n; i++) {
        free(L[i]);
    }
    free(L);
    for (int i = 0; i < n; i++) {
        free(H[i]);
    }
    free(H);
```

```
}

// function to calculate the infinity norm
real infinity_norm(real** A, int n) {
    real max_row_sum = 0.0;
    for (int i = 0; i < n; i++) {
        real row_sum = 0.0;
        for (int j = 0; j < n; j++) {
            row_sum += fabs(A[i][j]);
        }
        if (row_sum > max_row_sum) {
            max_row_sum = row_sum; // update maximum row sum
        }
    }
    return max_row_sum; // return the infinity norm
}

// function to compute the inverse of A using Cholesky decomposition
void cholesky_inverse(real** L, real** A_inv, int n) {
    // solve L * Y = I for Y
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (i == j) {
                A_inv[i][j] = 1.0 / L[i][i]; // diagonal
            }
            else if (j < i) {
                real sum = 0.0;
                for (int k = j; k < i; k++) {
                    sum += L[i][k] * A_inv[k][j];
                }
                A_inv[i][j] = -sum / L[i][i]; // lower triangle
            }
        }
    }

    // solve L^T * A_inv = Y for A_inv
    for (int i = n - 1; i >= 0; i--) {
        for (int j = 0; j < n; j++) {
            real sum = 0.0;
            for (int k = i + 1; k < n; k++) {
                sum += L[k][i] * A_inv[k][j];
            }
            A_inv[i][j] = (A_inv[i][j] - sum) / L[i][i]; // upper triangle
        }
    }
}

// function to calculate the condition number using infinity norm
real condition_number(real** A, int n) {
    real norm_A = infinity_norm(A, n); // compute the infinity norm of A

    real** A_inv = (real**)malloc(n * sizeof(real*));
    for (int i = 0; i < n; i++) {
        A_inv[i] = (real*)malloc(n * sizeof(real));
```

```
    }

    // compute the Cholesky decomposition
    real** L = (real**)malloc(n * sizeof(real*));
    for (int i = 0; i < n; i++) {
        L[i] = (real*)malloc(n * sizeof(real));
    }
    cholesky_decomposition(A, L, n); // perform Cholesky decomposition

    cholesky_inverse(L, A_inv, n); // compute the inverse of A

    real norm_A_inv = infinity_norm(A_inv, n); // compute the infinity
norm of the inverse of A

    real cond_number = norm_A * norm_A_inv; // compute the condition
number

    // free allocated memory
    for (int i = 0; i < n; i++) {
        free(A_inv[i]);
        free(L[i]);
    }
    free(A_inv);
    free(L);

    return cond_number; // return the condition number
}

// function to compute the condition number for specific n values
void compute_condition_numbers() {
    int ns[] = {3, 6, 9, 12};
    for (int i = 0; i < 4; i++) {
        int n = ns[i];
        real** H = create_hilbert(n);
        real cond_num = condition_number(H, n);
        printf("Condition number for n = %d: %.2e\n", n, cond_num);

        // free the Hilbert matrix
        for (int j = 0; j < n; j++) {
            free(H[j]);
        }
        free(H);
    }
}

// function to calculate the mean relative error
double calculate_mean_relative_error(real* x, int n) {
    double mean_relative_error = 0.0;
    for (int i = 0; i < n; i++) {
        mean_relative_error += fabs(x[i] - 1.0); // compare with expected
solution x = (1, 1, ..., 1)
    }
    return (mean_relative_error/n);
}
```

```
// function to calculate the vector norm (2-norm)
real calculate_vector_norm(real* x, int n) {
    real vector_norm = 0.0;
    for (int i = 0; i < n; i++) {
        vector_norm += pow(x[i],2);
    }
    vector_norm = sqrt(vector_norm);
    return vector_norm;
}

// function to find instability n (use 2-norm)
void find_instability_n() {
    int n = 1;
    real last_error = 0.0;

    while (1) {
        n++; // start from n = 2
        real* x = (real*)malloc(n * sizeof(real));

        // solve the Hilbert matrix equation
        solve_hilbert(n, x);

        // calculate the relative error
        real vector_norm = calculate_vector_norm(x, n);
        real relative_error = (vector_norm - sqrt(n)) / sqrt(n); //
relative error (use 2-norm)

        // check for instability
        if (relative_error > 0.5) {
            printf("Instability found at n = %d with relative error:
%.2f%%\n", n, relative_error * 100);
            printf("Last stable n = %d with relative error: %.2f%%\n", n -
1, last_error * 100);
            free(x);
            break;
        }

        last_error = relative_error; // store the last error for
comparison

        free(x);
    }
}

// main function
int main() {
    int n = 5; // set n for the second problem
    real* x = (real*)malloc(n * sizeof(real));

    // solve the Hilbert matrix equation for n = 5
    solve_hilbert(n, x);

    // print results
```

```
    printf("In "DATATYPE":\n");
    printf("Solution x for n = %d:\n", n);
    for (int i = 0; i < n; i++) {
        printf("x[%d] = " REAL_FORMAT "\n", i, x[i]);
    }

    // calculate the mean relative error
    real mean_relative_error = calculate_mean_relative_error(x, n);
    printf("Mean relative error compared to expected solution: "
REAL_FORMAT "\n", mean_relative_error);

    free(x);

    // find the instability n (solve the third problem)
    find_instability_n();

    // compute condition numbers for n = 3, 6, 9, 12 (solve fourth
problem)
    compute_condition_numbers();

    return 0;
}
```

Outputs for (1)~(3):

If use DOUBLE, the outputs are as following:

```
In DOUBLE:
Solution x for n = 5:
x[0] = 0.99999999999993183231
x[1] = 1.00000000000122302168
x[2] = 0.99999999999486055557
x[3] = 1.00000000000761612995
x[4] = 0.99999999999632793735
Mean relative error compared to expected solution: 0.000000000035437652812%
Instability(use 2-norm) found at n = 13 with relative error: 196.35%
```

If use FLOAT, the outputs are as following:

```
In FLOAT:
Solution x for n = 5:
x[0] = 1.00016248226165771484
x[1] = 0.99712842702865600586
x[2] = 1.01198256015777587891
x[3] = 0.98230671882629394531
x[4] = 1.00851345062255859375
Mean relative error compared to expected solution: 0.8244668413619995117%
Instability(use 2-norm) found at n = 7 with relative error: 111.83%
```

Outputs for (4) in DOUBLE:

```
Condition number for n = 3: 7.48e+02
Condition number for n = 6: 2.91e+07
Condition number for n = 9: 1.10e+12
Condition number for n = 12: 3.75e+16
```

As n increases, the condition number grows rapidly, consistent with the instability trend of the solution in (3), indicating that the condition number is a good measure of the solution's instability.

Answer to (5):

In fact, when using the float type to calculate the condition number in my code, the precision isn't sufficient to correctly compute the inverse of an ill-conditioned matrix for $n \geq 8$. One possible improvement is to use Singular Value Decomposition (SVD) to calculate the condition number.