02635 Fall 2016 — Module 3 (solutions)

Exercises — Part I

1. Exercises 5-1, 5-2, and 5-4 in "Beginning C":

   Exercise 5-1

```
#include <stdio.h>

int main(void) {

  double arr[5], repr[5], rsum = 0.0;

  // Prompt user to enter numbers
  printf("Enter five nonzero numbers:\n");
  for (int i=0;i<5;i++) {
    printf("%2d> ",i+1);
    scanf("%lf",arr+i);
  }

  // Compute and sum reciprocals
  for (int i=0;i<5;i++) {
    repr[i] = 1/arr[i];
    rsum += repr[i];
  }
  printf("Sum of reciprocals: %.4e\n", rsum);

  return 0;
}
```

   Exercise 5-2

```c
#include <stdio.h>

int main(void) {

  double data[100], val = 0.0, k = 1.0;
  int j;

  // Compute array with terms
  for (int i=0;i<100;i++) {
    j = 2 + 2*i;  // 2,4,6,...,200
    data[i] = 1.0/(j*(j+1)*(j+2));
  }

  // Sum terms with alternating sign
  for (int i=0;i<100;i++) {
    val += data[i]*k;
    k *= -1;
  }

  // Multiply by 4, add 3, and print
  val = 4*val + 3;
  printf("Value: %.16lf\n",val);

  return 0;
}
```

Exercise 5-4

```c
#include <stdio.h>

int main(void) {

  // Allocate and fill array
  double data[11][5];
  for (int i=0;i<11;i++) {
    data[i][0] = 2.0 + 0.1*i;           // x
    data[i][1] = 1/data[i][0];          // 1/x
    data[i][2] = data[i][0]*data[i][0]; // x^2
    data[i][3] = data[i][0]*data[i][2]; // x^3
    data[i][4] = data[i][0]*data[i][3]; // x^4
  }

  // Print table head
  printf("%8s %8s %8s %8s %8s\n","x","1/x","x^2","x^3","x^4");
  for (int i=0;i<11;i++) {
    // Print table row
    for (int j=0;j<5;j++)
      printf("%8.2f ",data[i][j]);
    printf("\n");
  }
  return 0;
}
```

2. See quiz answers [here](#).

3. See quiz answers [here](#).

4. The program computes the average of the 10 numbers. The while loop is equivalent to a for loop:

```c
#include <stdio.h>

int main(void) {

    int arr[10] = {19,74,13,67,44,80,7,36,9,77};
    double val = 0.0;

    // Compute average
    for (int i=0;i<10;i++) val += arr[i];
    val /= 10;

    printf("Value: %.2f\n",val);

    return 0;
}
```

5. Copying arrays with memcpy:

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

#define N 10

int main(void) {

  double arr1[N], arr2[N];

  // Prompt user to enter N numbers
  printf("Input %d numbers:\n",N);
  for (int i=0;i<N;i++) {
    printf("%2d>",i+1);
    scanf("%lf",arr1+i);
  }

  // Copy data from arr1 to arr2
  memcpy(arr2,arr1,N*sizeof(double));

  // Print values stored in arr2
  for (int i=0;i<N;i++)
    printf("%2d: %.4f\n",i+1,arr2[i]);

  return 0;
}
```

Exercises — Part II

1. We have

$$
\begin{aligned}
D(t) &= \sqrt{\left(\frac{dx(t)}{dt}\right)^2 + \left(\frac{dy(t)}{dt}\right)^2} \\
&= \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}
\end{aligned}
$$

   which is the distance between the points (x_1, y_1) and (x_2, y_2). Thus, $D(t)$ is a constant and does not depend on t.

2. Program to approximate integral of $f(x, y) = \cos(1 - xy)$ over a line segment:

   `line_int.c`

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

int main(void) {

int n;               // number of subintervals
double x1,y1,x2,y2; // points on line segment
double val;          // storage for result
double t, dist;

// Prompt user to enter parameters
printf("Enter number of subintervals: ");
scanf("%d",&n);

printf("Enter x1,y1: ");
scanf("%lf,%lf",&x1,&y1);
printf("Enter x2,y2: ");
scanf("%lf,%lf",&x2,&y2);

printf("Computing integral over line segment "
    "between (%.2f,%.2f) and (%.2f,%.2f):\n",x1,y1,x2,y2);

// Compute distance between the two points
dist = sqrt((x2-x1)*(x2-x1) + (y2-y1)*(y2-y1));

// Apply repeated trapezoidal rule
val = (cos(1-x1*y1) + cos(1-x2*y2))/2.0;
for (int i=1;i<n;i++) {
    t = (double)i/n;
    val += cos(1-(x1+t*(x2-x1))*(y1+t*(y2-y1)));
}
val *= dist/n;

printf("Value: %.4e\n",val);

return 0;
}
```

Compiling the program:

```
$ cc line_int.c -Wall -lm -o line_int
```

Test cases:

To test the program, we will define two simple test cases.

For the first test case, we use $n = 50$, $(x_1, y_1) = (0, 0)$, and $(x_2, y_2) = (1, 0)$. This corresponds to the definite integral

$$\int_0^1 \cos(1)\, dx = \cos(1).$$

For the second test case, we use $n = 2$, $(x_1, y_1) = (0, 1)$, and $(x_2, y_2) = (1, 1)$. This corresponds to the definite integral

$$\int_0^1 \cos(1 - x)\, dx = \sin(1) - \sin(0).$$

To avoid entering the same input again and again, we create two text files with the test case data:

`tc1.txt`

```
50
0,1
1,1
```

`tc2.txt`

```
2
0,0
1,0
```

The two test cases can now be executed from a bash shell using the following commands:

```
$ ./line_int < tc1.txt
Value: 8.4144e-01

$ ./line_int < tc2.txt
Value: 5.4030e-01
```

3. The elements of H can be computed as follows:

```
for (int i=0;i<M;i++) {
    y = yl + i*(yu-yl)/(M-1);
    for (int j=0;j<N;j++) {
        x = xl + j*(xu-xl)/(N-1);
        H[i][j] = cos(1.0-x*y);
    }
}
```

4. Given (x, y) that satisfies $x_l \leq x \leq x_u$ and $y_l \leq y \leq y_u$, we can compute the index of the nearest grid point *below* and *left* of (x, y) by solving for i and j and rounding down:

```
jdbl = floor((x-xl)/(xu-xl)*(N-1));
idbl = floor((y-yl)/(yu-yl)*(M-1));

if (idbl < 0 || idbl > M-1 || jdbl < 0 || jdbl > N-1)
    printf("Point is outside grid\n");
else {
    i = (int) idbl;
    j = (int) jdbl;
}
```

Here `idbl` and `jdbl` are of type `double`.

5. Program to approximate integral of $f(x, y) = \cos(1 - xy)$ over a line segment using linear interpolation:

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

int main(void) {

  int n,M,N;                // number of subintervals, vertical/horizontal grid points
  double xl=0.0,yl=0.0;     // upper-right corner of grid
  double xu=1.0,yu=1.0;     // lower-left corner of grid
  double x1=0.0,y1=0.0;     // point 1 on line
  double x2=0.0,y2=0.0;     // point 2 on line
  double x,y;               // a point (x,y)
  double val=0.0;           // result
  double dist=0.0;          // length of line segment

  // Prompt user to enter data
  printf("Enter number of subintervals: ");
  scanf("%d",&n);
```

```c
printf("Enter grid dimensions M,N: ");
scanf("%d,%d",&M,&N);

printf("Enter x1,y1: ");
scanf("%lf,%lf",&x1,&y1);
printf("Enter x2,y2: ");
scanf("%lf,%lf",&x2,&y2);

// Create and precompute H
double H[M][N];
for (int i=0;i<M;i++) {
  y = yl + i*(yu-yl)/(M-1);
  for (int j=0;j<N;j++) {
    x = xl + j*(xu-xl)/(N-1);
    H[i][j] = cos(1.0-x*y);
  }
}

printf("Computing integral over line segment "
  "between (%.2f,%.2f) and (%.2f,%.2f):\n",x1,y1,x2,y2);

// Compute distance between the two points
dist = sqrt((x2-x1)*(x2-x1) + (y2-y1)*(y2-y1));

// Apply repeated trapezoidal rule
int ii,jj;
double idbl,jdbl,c,s;
double x_ll,y_ll,x_ur,y_ur;  // lower-left and upper-right grid point
for (int i=0;i<=n;i++) {

  // Point on line segment
  x = x1+(x2-x1)*i/n;
  y = y1+(y2-y1)*i/n;

  // Compute index of grid point left and below (x,y)
  jdbl = floor((x-xl)/(xu-xl)*(N-1));
  idbl = floor((y-yl)/(yu-yl)*(M-1));
  if (idbl<0 || idbl>M-1 || jdbl<0 || jdbl>N-1) {
    printf("Point (%.2e,%.2e) is outside grid\n",x,y);
    continue;
  }
  else {
    ii = (int)idbl;
    jj = (int)jdbl;
```

```
    }

    // Grid point to the left and below (x,y) ["lower-left"]
    x_ll = xl+(xu-xl)*jj/(N-1);
    y_ll = yl+(xu-xl)*ii/(M-1);

    // Grid point to the right and above (x,y) ["upper-right"]
    x_ur = x_ll + (xu-xl)/(N-1);
    y_ur = y_ll + (yu-yl)/(M-1);

    // Compute scalar constant
    s = 1.0/((x_ur-x_ll)*(y_ur-y_ll));
    if (i>0 && i<n) c = s;  // scale by s
    else c = 0.5*s;         // scale by 0.5*s if i = 0 or i = n

    // Approximate f(x,y) using bilinear interpolation
    val += c*H[ii][jj]*(x_ur-x)*(y_ur-y);
    if (jj<N-1) val += c*H[ii][jj+1]*(x-x_ll)*(y_ur-y);
    if (ii<M-1) val += c*H[ii+1][jj]*(x_ur-x)*(y-y_ll);
    if (jj<N-1 && ii<M-1) val += c*H[ii+1][jj+1]*(x-x_ll)*(y-y_ll);
  }
  val *= dist/n;

  printf("Value: %.4e\n",val);

  return 0;
}
```

Compiling the program:

```
$ cc line_int_bilinear.c -Wall -lm -o line_int_bilinear
```

6. The solution example to exercise 5 is neither flexible nor particularly efficient. However, notice that H is computed row-by-row which is consistent with the row-wise storage of two-dimensional arrays in C.

   The code can be made more flexible by dividing the code into a number of separate routines, allowing H to be specified by the user.