GNG1106 – Fundamentals of Engineering Computation
Practice Questions - Solutions

Short Answer Questions

1)

Console

```
a = 10 b = 6
```

2)
```
double x;
updateValue(&x);
```

3)

(a) **`ITEM` is the name of the structure type, not the structure variable**
```
inventory[1].cost = 92.32;
```

(b) **`cost` is not an array an hence an index cannot be applied to it, it is `inventory` that is an array.**
```
inventory[10].cost = 92.32
```

4)
```
mPtr = &mass;
```

5) To create the file *exam.txt* if it does not exist, or erase the contents of the file *exam.txt* if it does exist.

6) First lin'\0' note that the last character is a nul character.

7)

Console

```
First: x=50 y=70
Now: x=70 y=50
```

8)

```
FILE *fPtrSource, *fPtrDest;
char name[80];
char line[100];
printf("Give name of a source file: ");
scanf("%s",name); // this works only with file names with no spaces
fflush(stdin);
fPtrSource = fopen(name, "r");  // open for read
if(fPtrSource == NULL)
{
   printf("File does not exist\n");
}
else
{
   fPtrDest = fopen("Copy.txt","w");  // assume no error
   while(feof(fPtrSource) != 0)
   {
      fgets(line,99,fptrSource)
      fputs(line, fPtrDest);
   }
   printf("File Copied\n");
   fclose(fPtrSource);
   fclose(fPtrDest);
}
```

9) (b) the address zero that does not reference anything

10) (a) Reserves 112 bytes for `var`

11) (b)**`var.str`** gives the address of where the array `str` is stored

12) (c)The function call `func(var)` passes the complete contents of `var` to `func.`

Computer Algorithms

1) Bubble sort double values descending order incorporating sub-arrays.

```
/*-----------------------------------------------
Function: bubbleSortCharReverse
Parameters:
   array - pointer to an array of characters
   n - number of characters in the array
Description:  A function that sorts the array
   of characters.  The function is efficient.
-----------------------------------------------*/
void bubbleSortCharReverse(int n, char array[])
{
    int cntr; // counter of itterations
    int ix;   // for indexing into the array
    char temp;

    for(cntr = 0; cntr < n - 1; cntr = cntr + 1)
    {
        // Sort from 0 to n - cntr, only sub-array
        for(ix = 0; ix < n - cntr - 1; ix = ix + 1)
        {
            if(array[ix] < array[ix+1])
            {
               temp = array[ix];
               array[ix] = array[ix+1];
               array[ix+1] = temp;
            }
        }
    }
}
```

2) Replacement sort with strings.

```
/*-------------------------------------------------
Function: sort_repl_string
Parameters:
    num - number of rows in the array, i.e. number of strings
    num_char - max number of chars in rows (size of column),
               i.e. max size of string
    strArr - reference to a 2D array that contains strings to sort.
Description: Sorts the referenced array of strings
             using the replacement sort algorithm.
-------------------------------------------------*/
void sort_repl_string(int num, int num_char, char strArr[][num_char])
{
   int ix, pass, largestPos; // "largest string"
   char hold[num_char]; // For copying strings
   /*Replacement sort algorithm. */
   for(pass = 1; pass < num; pass=pass+1)
   {
      largestPos=0;
      for(ix = 1; ix <= (num - pass); ix=ix+1)
         if(strcmp(strArr[ix],strArr[largestPos]) > 0) largestPos=ix;
      strcpy(hold,strArr[num-pass]);
      strcpy(strArr[num-pass],strArr[largestPos]);
      strcpy(strArr[largestPos],hold);
   }
}
```

3) Bubble sort with real values (double).

```
#define EPSILON 0.001
/*-----------------------------------------------
Function: bubbleSortReaValues
Parameters:
   n - number of values in the array
   array - pointer to an array of real values
Description:  A function that sorts the array
   of real values. Considered equal if values have
   difference less than EPSILON.
-----------------------------------------------*/
void bubbleSortRealValues(int n, double array[])
{
    int cntr; // counter of itterations
    int ix;   // for indexing into the array
    double temp; // For exchanging values
    for(cntr = 0; cntr < n - 1; cntr = cntr + 1)
    {
        // Pass through the array
        for(ix = 0; ix < n-1; ix = ix + 1)
        {
            if((array[ix] > array[ix+1]) &&
               (fabs(array[ix] - array[ix+1])> EPSILON)) // not considered equal
            {
               temp = array[ix];
               array[ix] = array[ix+1];
               array[ix+1] = temp;
            }
        }

    }
}
```

4) Searching strings – binary.

```
/*-------------------------------------------
Function: binarySearchString
Parameters:
    key - the value to search in the array
    num - number of rows in the array, i.e. number of strings
    num_char - max number of chars in rows, i.e. max size of string
    strArr - reference to a 2D array that contains strings to sort.
Return Value: Position (index) of key in array
              or -1 if key not found.
Description: Returns the position of the key
             found (i.e. row where key is found)
             or -1 if the key is not found in the array.
-------------------------------------------*/
int binarySearchString(char *key, int num, int num_char, char arr[][num_char])
{
   // All variables are indexes into the array
   int middle;    // Middle of subarray
   int pos = -1;  // Position of key if found
   int low = 0;   // lower bound of subarray
   int high = num-1; // upper bound of subarray
   // Loop to search the array
   while ( (low <= high) && (pos == -1))
   {
      middle = (low + high) / 2;
      if(strcmp(arr[middle],key) == 0)
         pos = middle;               /* Found key. */
      else if (strcmp(key, arr[middle]) < 0)
         high = middle - 1;           /* Search bottom half.  */
      else
         low = middle + 1;            /* Search top half. */
   }
   return(pos);
}
```

5) Searching values integers – linear, for all values in the array.

```
/*-----------------------------------------
Function: linearSearch
Parameters:
    key - the value to search in the array
    num - the number of elements in both arrays (arr and found)
    arr - reference to an array of integer
          values
    found - reference to array for storing the indexes of elements
            that match the key
Return Value: Number of times the key was found.
Description: Searches for all occurences of the key in the
             array.
-----------------------------------------*/
int linearSearchAll(int key, int num, int arr[], int found[])
{
   int ix;
   int ixFound;
   ixFound = 0;  // found no matches to key yet
   // Check all elements in the array
   for( ix = 0; ix < num; ix = ix + 1)
   {
      if (arr[ix] == key)
      {
          found[ixFound] = ix;
          ixFound = ixFound + 1;
      }
   }
   return(ixFound);
}
```

6) Recursive function to sum 1 to N integers.

```
/*-----------------------------------------------------
Function: sumN
Parameters:
    integer:  integer value
Description: Recursive function for computing the
             sum from 1 to N (positive integer).
-----------------------------------------------------*/
int sumN(int integer)
{
    int sum;
    if(integer == 1) // Simplest case.
        sum = 1;
    else // Recursive call.
        sum = integer + sumN(integer - 1);
    return(sum);
}
```

7) Recursive function to solve x^b.

```
/*-------------------------------------------------------
Function: powerRecursive
Parameters:
    x:  real value
    b:  positive integer value
Description: Recursive function for computing x
             to the power of b.
-------------------------------------------------------*/
double powerRecursive(double x, int b)
{
    double power;
    if(b == 1) // Simplest case.
        power = x;
    else // Recursive call.
        power = x * powerRecursive(x, b -1);
    return(power);
}
```

8) Recursive function to reverse the characters in a string.

```
/*-------------------------------------------------------
Function: reverseString
Parameters:
    n:  number of chars to reverse
    str: reference to string to reverse
Description: Recursive function for reversing the
             characters in a string.
-------------------------------------------------------*/
void reverseString(int n, char str[])
{
    char save;
    // Do nothing if <= 1, thus check only
    // for complex case
    if(n > 1) // Simplest case 1 or 0 characters
    {
       // First reverse first and last char
       save = str[0];
       str[0] = str[n-1];
       str[n-1] = save;
       // Now reverse rest of string
       reverseString(n-2, &str[1]);
    }
    return;
}
```

Numerical Methods

1)

```
/*-------------------------------------------------------
Function: nthroot
Parameters
    n - real value for finding the nth root
    c - real value to find nth root of c
Returns:  The nth root of c
Description: Finds the nth root of c by finding the root
             of the function f(x)= x^n - c using the bisection
             root finding method with the interval from 0
             to c.   EPSILON is set to 10e-6 and termination
             of the root finding is done when the search interval
             has been reduced within ESPSILON.
-------------------------------------------------------------*/
#define EPSILON 10e-7
double nthroot(double n, double c)
{
    double func_xL, func_xU;  // values of function at border of interval
    double xR,func_xR; // Root estimate
    double xL, xU;  // Search interval
    // Some initialisation
    xU = c;
    xL = 0;  // Start interval search between 0 and c
    func_xL = pow(xL,n) - c;  // Initilise f(xL) and f(xU)
    func_xU = pow(xU,n) - c;
    while((xU-xL) > EPSILON)
    {
        xR = (xU+xL)/2;
        func_xR = pow(xR,n) - c;
        if( (func_xL*func_xR) < 0.0 )  // zero in lower half
        {
            xU = xR;
            func_xU = pow(xU,n) - c; // recalculate f(xU)
        }
        else // zero in upper half
        {
            xL = xR;
            func_xL = pow(xL,n) - c; // recalculate f(xL)
        }
    }
    return(xR);
}
```

2) (a) The equation can reorganized as follows:

$$\ln(o_{sf}) + 139.34411 - \frac{1.575701\times10^{5}}{T_a} + \frac{6.642308\times10^{7}}{T_a^2} - \frac{1.2438\times10^{10}}{T_a^3} + \frac{8.621949\times10^{11}}{T_a^4} = 0$$

Thus, the problem consists of finding the root of the following function to provide the temperature for a given saturation concentration of oxygen.

$$f(T_a) = \ln(o_{sf}) + 139.34411 - \frac{1.575701\times10^{5}}{T_a} + \frac{6.642308\times10^{7}}{T_a^2} - \frac{1.2438\times10^{10}}{T_a^3} + \frac{8.621949\times10^{11}}{T_a^4}$$

(b)

```
/*-------------------------------------------------------------
Function: getSatTemp
Parameters:
    osf - saturation temperature of dissolved oxgen in fresh water
Returns:  The freshwater temperature
Description: Finds the temperature using a root finding method
             root finding method with the interval from 0
             to 40 degrees C.
-------------------------------------------------------------*/
#define EPSILON 10e-7
#define TEMP_L 0.0 // Degrees C
#define TEMP_U 40.0 // Degrees C
#define TCONV 273.15 // To converte between C and K
double getSatTemp(double osf)
{
    double func_tL, func_tU;  // values of function at border of interval
    double tR,func_tR; // Root estimate
    double tL, tU;  // Temperature Search interval
    // Some initialisation
    tU = TEMP_U + TCONV;
    tL = TEMP_L + TCONV;  // Start interval search between 0 and c
    func_tL = f_ta(tL, osf);  // Initilise f(xL) and f(xU)
    func_tU = f_ta(tU, osf);
    tR=-9999; // Indicates an error
    if(func_tL*func_tU < 0)  // Ensures concentration is in range
    {
        while( fabs(func_tU*func_tL) > EPSILON)
        {
            tR = (tU+tL)/2;
            func_tR = f_ta(tR, osf);
            if( (func_tL*func_tR) < 0.0 )  // zero in lower half
            {
                tU = tR;
                func_tU = f_ta(tU, osf); // recalculate f(xU)
            }
            else // zero in upper half
            {
                tL = tR;
                func_tL = f_ta(tL, osf); // recalculate f(xU)
            }
        }
        tR = tR-TCONV;  // converte to degrees C
    }
    return(tR);
}

/*-------------------------------------------------------------
Function: f_ta
Parameters:
    ta - temperature in degrees Kelvin
    osf - saturation concentration of oxygen in mg/L
Description:  Calculates value of f(ta) for given temperature
              and saturation concentration.
-------------------------------------------------------------*/
#define A 139.34411
#define B 1.575701e5
#define C 6.642308e7
#define D 1.2438e10
#define E 8.621949e11
```

```
double f_ta(double ta, double osf)
{
    double fta;
    //fta = A - B/ta + C/(ta*ta) - D/pow(ta,3) + E/pow(ta,4);
    //fta = exp(-fta);
    fta = log(osf)+ A - B/ta + C/(ta*ta) - D/pow(ta,3) + E/pow(ta,4);
    return(fta);
}
```

3) (a)

$$f(x) = x^2 \cos(x) - 5$$

or

$$f(x) = 5 - x^2 \cos(x)$$

(b)

```
/*-------------------------------------------------------------
Function: findAllRoots
Parameters:
    start, end:  start and end of interval (x values) for searching roots
    roots - reference to array for storing roots
Returns:
    The number of roots found (value of rootIx at the end of the loop)
Description: Find the roots between the interval for the
             function f(x) = x^2 cos(x)-5 (see the function func).
-------------------------------------------------------------------*/
int findAllRoots(double start, double end, double roots[])
{
    double subinter = (end-start)/SI_RESOLUTION;
    int cntr; // for counting subintervals
    double ak, bk;  // values at the borders of the subinterval
    double root;    // for getting root values
    int rootIx;  // for indexing into roots
    rootIx = 0;
    for(cntr = 0; cntr < SI_RESOLUTION; cntr = cntr + 1)
    {
        ak = start + (cntr*subinter);
        bk = start + ((cntr + 1)*subinter);
        if(bk > end) bk = end;
        if(findRoot(ak, bk, &root))
        {
            roots[rootIx] = root;
            rootIx = rootIx + 1;
        }
    }
    return(rootIx);
}
/*-------------------------------------------------------------
Function: findRoot
Parameters:
    left, right:  values of x at the edges of the subinterval
    root - reference to save the value of the root found.
Description: Check if a root exists in the subinterval for the
             polynomial defined by coefficients.
-------------------------------------------------------------------*/
int findRoot(double left, double right, double *root)
{
    double func_left, func_right;  // values of function at border of interval
    double factor;  // for storing func_left*func_right
    func_left = func(left);
```

```
    func_right = func(right);
    factor = func_left*func_right;
    int retVal = FALSE;
    if(fabs(factor) < ALMOST_0)  // About zero
    {
        if(fabs(func_left) < ALMOST_0)
        {
            *root = left;
            retVal = TRUE;
        }
    }
    else if( factor < 0.0)
    {
        *root = (left+right)/2.0;
        retVal = TRUE;
    }
    return(retVal);
}

/*------------------------------------------------------------
Function: func
Parameters:
    x - x value function f(x)
Returns: value y of function f(x)
Description: Plots the value of the function:
             f(x) = x^2 cos(x) - 5.
------------------------------------------------------------*/
double func(double x)
{
    double fx;
    fx = x*x*fabs(cos(x)) - 5;
    return(fx);
}
```

4) (a) Applying Euler's method gives us the following difference equation :

$$t_{i+1} = t_i + \Delta t$$

$$y_{i+1} = y_i + f(y_i)\Delta t$$

$$= y_i - \left(k\sqrt{y_i}\right)\Delta t$$

and the initial conditions are t_0 =0.0, y_0 = initial depth of the water. The time step Δt is defined as 0.01 minutes.√

(b)

```
/*-------------------------------------------------------------
Function: calculateDrainTime
Parameters:
    k - Constant k that depends on shape of drain hole and
        cross-sectional area of tank.
    yInit - Initial depth of the liquid in the tank.
Return Value:
    The time it takes for all the liquid to drain from the tank.
Description: Using Euler's method, computes the time it takes
             to drain all the liquid from the cylindrical
             tank.
-------------------------------------------------------------*/
#define DELTA_T 0.01  // in minutes
double calculateDrainTime(double k, double yInit)
{
    double ti;  // Time
    double yi; // depth of the liquid
    // Initiliase the variables
    ti = 0.0;
    yi = yInit;
    while(yi > 0) // as long as liquid exists in the tank
    {
        yi = yi - k*sqrt(yi)*DELTA_T;  // computes yi+1 from yi
        ti = ti + DELTA_T;
    }
    return(ti);
}
```

5) (a) Applying Euler's method to both differential equations gives us the following difference equations :

Time: $t_{i+1} = t_i + \Delta t$

Velocity: $v_{i+1} = v_i + f(v_i)\Delta t = v_i + \left(g - \frac{c_d}{m} v_i^2 \right) \Delta t$

Distance: $x_{i+1} = x_i + f(x_i)\Delta t = x_i - v_i \Delta t$

Initial conditions: $t_0 = 0.0$, $v_0 = 0.0$, x_0 = initial height of the object.

(b)

```
/*-------------------------------------------------------------
Function: calculateDropTime
Parameters:
    height - Initial height at point of release in meters
    weight - Weight of object in kg.
    drag - drag coefficient cd in km/m.
Return Value:
    The time it takes for the object to reach the ground.
Description: Using Euler's method, computes the time it takes
             for the falling object to reach the ground.
-------------------------------------------------------------*/
#define DELTA_T 0.01  // in seconds
double calculateDropTime(double height, double weight, double drag)
{
    double ti;  // Time
    double vi; // velocity
    double xi; // distance from point of release
    // Initiliase the variables
    ti = 0.0;
    vi = 0.0;
    xi = height;
    while(xi > 0) // as long as above ground
    {
        ti = ti + DELTA_T;  // Go to i+1
        xi = xi - vi*DELTA_T;  // Computes xi from xi-1 and vi -
                               // note must come before updating vi
        vi = vi + (G-(drag/weight)*vi*vi)*DELTA_T;  // computes vi+1 from vi
    }
    return(ti);
}
```

6) (a) Applying Euler's method the differential equation gives us the following difference equation :

Time: $t_{i+1} = t_i + \Delta t$

Population: $p_{i+1} = p_i + f(p_i)\Delta t = v_i + \left[k_{gm}\left(1 - \frac{p_i}{p_{\max}}\right) p_i \right] \Delta t$

Initial conditions: t_0 = first year, p_0 = initial population

(b)

```
/*-------------------------------------------------------------
Function: calculatePopulationEuler
Parameters:
    popInit – initial population at year y1
    kgm - growth rate
    pmax - carrying capacity
    y1, y2 - range of years to compute population increase
    n - number of points in array, i.e. number of columns
    popPoints - points of the time/population
                Row T_IX contains time values
                Row POP_IX contains population values.
Description: Computes the change in population from y1
             to y2 using Euler's method to solve differential
             equation.
-------------------------------------------------------------*/
#define T_IX 0  // Row for time values in popPoints
#define POP_IX 1 // Row for population values in popPoints
void calculatePopulationEuler(double popInit, double kgm,  double pmax,
                              double y1, double y2,
                              int n, double popPoints[][n])
{
    int i;
    double deltat;
    double f_p;  // for computing f(pi-1)
    // Some initialization
    deltat = (y2-y1)/(n-1);  // Note that for n values, there are n-1 steps
    popPoints[T_IX][0] = y1;
    popPoints[POP_IX][0] = popInit;
    for(i = 1; i < n; i = i +1)
    {
        popPoints[T_IX][i] = popPoints[T_IX][i-1] + deltat;
        f_p = kgm*(1-popPoints[POP_IX][i-i]/pmax)*popPoints[POP_IX][i-1];
        popPoints[POP_IX][i] = popPoints[POP_IX][i-1] + f_p*deltat;
    }
}
```

7)

```
/*-------------------------------------------------------------
Function: integrate
Parameters:
    integralId - one of I1, I2, I3 or I4 to identify integral
                 to solve
    x1, x2 - limits of integration
Return Value
    The result of the definite integral
Description: Solves one of four defnite integrals and returns
             the result.
-------------------------------------------------------------*/
#define NUM_STEPS 20
double integrate(int integralId, double x1, double x2)
{
   // Declarations
   double c;  // the result of the integral
   double sum;  // for computing the sum in the numerical equation
   int i;
   double h = (x2-x1)/NUM_STEPS;
   double xi; // for incrementing values of x

   // Calculate the sum
   sum =0;
   xi = x1+h; // start at initial
   for(i=1; i< NUM_STEPS; i=i+1)
   {
       sum = sum+f_x(integralId,xi);
       xi = xi + h;
   }
   c = (f_x(integralId, x1) + 2*sum + f_x(integralId, x2))*h/2;
   return(c);
}

/*-------------------------------------------------------------
Function: f_x
Parameters:
    funcId - one of I1, I2, I3 or I4 to identify function f(x)
    x - value of x
Return Value
    The result of f(x)
Description: Computes f(x) using the funcId to determine which
             function to solve.
-------------------------------------------------------------*/
double f_x(int funcId, double x)
{
    double f_x;
    if(funcId == I1)  f_x = pow((x + 1)/x, 2);
    else if(funcId == I2) f_x = pow(4*x-3, 3);
    else if(funcId == I3) f_x = x*x*exp(x);
    else f_x = pow(15.0, 2*x);
    return(f_x);
}
```

8) (a) First, let's substitute for density and cross-sectional area in the integral and simplify:

$$m = \int_0^L \rho(x) A_C(x)\,dx = \int_0^L f(x)\,dx$$

Where $f(x) = \rho(x) A_C(x)$

$\rho(x) = 4000 - 70x$

$A_c(x) = 0.01 + 5\times10^{-5}x^2$

Substituting $\rho(x)$ and A_c into $f(x)$ and simplifying gives:

$$\begin{aligned} f(x) &= \rho(x) A_C(x) \\ &= (4000 - 70x)\left(0.01 + 5\times10^{-5}x^2\right) \\ &= 40 + 0.2x^2 - 0.7x - 0.0035x^3 \\ &= 40 - 0.7x + 0.2x^2 - 0.0035x^3 \end{aligned}$$

Note that $f(0) = 40$, applying the Trapezoidal rule, the numerical equation becomes:

$$\int_0^L f(x)\,dx = \frac{h}{2}\left[f(x_0) + \left[2\sum_{i=1}^{n-1} f(x_i) \right] + f(x_n) \right] = h\left[20 + \left[\sum_{i=1}^{n-1} f(x_i) \right] + 0.5 f(L) \right]$$

Where $f(x_i) = 40 - 0.7x_i + 0.2x_i^2 - 0.0035x_i^3$ and $h = L/n$ where n is the number of steps to divide the integration interval 0 to L. For programming and efficiency $f(x_i)$ can be computed as:

$$f(x_i) = (-0.0035x + 0.2)x - 0.7)x + 40$$

(b)

```
/*-------------------------------------------------------------
Function: findRodMass
Parameters:
    length - rod length
Return Value
    The mass of the rod.
Description: Applies the trapezoidal rule to solve for the
             mass of a rod whose density and cross-sectional
             area varies with length.
-------------------------------------------------------------*/
#define NUM_STEPS 20
double findRodMass(double length)
{
   // Declarations
   double mass;  // the result of the integral
   double sum;  // for computing the sum in the numerical equation
   int i;
   double h = length/NUM_STEPS;
   double xi; // for incrementing values of x

   // Calculate the sum
   sum =0;
   xi = h; // start at initial
   for(i=1; i< NUM_STEPS; i=i+1)
   {
       sum = sum+f_x(xi);
       xi = xi + h;
   }
   mass = (20.0 + sum +
           0.5*f_x(length))*h;
   return(mass);
}
```

```
/*----------------------------
Function: f_x
Parameters:
    x - value of x
Return Value
    The result of f(x)
Description: Computes f(x).
-----------------------------*/
double f_x(double x)
{
  double f_x;
  f_x = (((-0.0035*x+0.2)*x)-0.7)*x+40.0;
  return(f_x);
}
```

9) (a) Revise the equation to simplify the integral:

$$v_{rms} = \sqrt{\frac{1}{T}\int_0^T A^2 \sin^2(2\pi ft)dt} = \sqrt{\frac{1}{T} A^2 \int_0^T \sin^2(2\pi ft)dt}$$

Apply the Trapezoidal rule to solve for the integral and note that for the sin function that

at t=0, sin(0)=0 and

at $t = T$, $\sin(2\pi fT) = \sin(2\pi) = 0$ since $T = 1/f$:

$$\int_0^T \sin^2(2\pi ft)dt = \frac{h}{2}\left[\sin^2(0) + \left[2\sum_{i=1}^{n-1}\sin^2(2\pi ft_i)\right] + \sin^2(2\pi fT)\right] = h\sum_{i=1}^{n-1}\sin^2(2\pi ft_i)$$

The value of $h=T/n$, where n is the number of integration steps. Thus v_{rms} can be found using the following numerical equation:

$$v_{rms} = \sqrt{\frac{1}{T} A^2 h\sum_{i=1}^{n-1}\sin^2(2\pi ft_i)} = \sqrt{fA^2 h\sum_{i=1}^{n-1}\sin^2(2\pi ft_i)}$$

(b)

```
/*-------------------------------------------------------------
Function: rmsVoltage
Parameters:
    a - Signal amplitude A
    f - Signal frequency f
Return Value
    The RMS value of the AC voltage.
Description: Using the Trapezoidal rule, computes the
             RMS voltage of the AC voltage sin(2PI f t).
-------------------------------------------------------------*/
#define NUM_STEPS 20
double rmsVoltage(double a, double f)
{
   // Declarations
   double v_rms;  // the RMS voltage
   double sum;  // for computing the sum in the numerical equation
   int i;
   double period;  // the period T
   double h;      // the integration step
   double ti; // for incrementing values of t over the period
   // initialisation
   period = 1.0/f;
   h = period/NUM_STEPS;
   // Calculate the sum
   sum =0;
   ti = h; // start at i = 1
   for(i=1; i< NUM_STEPS; i=i+1)
   {
       sum = sum + pow(sin(2.0*M_PI*f*ti),2);
       ti = ti + h;
   }
   v_rms = sqrt(f*a*a*h*sum);
   return(v_rms);
}
```