1 Cell Arrays (12 points)

(2 points each) Record the output of the following MATLAB commands by completing every blank output line. If the line would return an error, write "error." If the line would start an infinite loop/recursion, write "infinite loop."

```
>> clear;
>> A = {[true, false], ones(2,2), 'EASY QUESTION', [5 10; 15 20;
    25 30]};

>> class(A(2,1))

ans = cell ✓

>> class(A{1,1})

ans = logical ✓

>> size(A(1,2))

ans = 1  1 ✓

>> size(A{4})

ans = 3  2 ✓

>> A{2,2}(1,2)+A{1,2}(2,2)

ans = 11 ✓

>> A{1} + A{2}

ans = Error. ✓
```

2 Structures (10 points)

(2 points each) Record the output of the following MATLAB commands by completing every blank output line. If the line would return an error, write "error." If the line would start an infinite loop/recursion, write "infinite loop."

```
>> clear;
>> cars = struct('make',{'Audi','BMW'},'model',{'A7','M5'});
>> trucks = struct('make','Ford','model','F150','year',2004);

>> class(trucks(1).year)
```

ans = double

```
>> size(cars)
```

ans = 1 2

```
>> cars(2).make
```

ans = 'BMW'

```
>> [cars, trucks]
```

ans = Error

```
>> cars(2).year = 2010;
>> candt = [cars,trucks];
>> candt(1).year
```

ans = ∅ []

8

3 Logicals (16 points)

(2 points each) Record the output of the following MATLAB commands by completing every blank output line. If the line would return an error, write "error." If the line would start an infinite loop/recursion, write "infinite loop."

```
>> clear;
>> true  || false
ans = 1
```

```
>> 2~=1 && 1==str2num('1')
ans = 1
```

```
>> x = 4;
>> x < 1 && x >= 0
ans = 0
```

```
>> x = [1 -3 0; 1 4 7; -8 1 3];
>> logical(mod(x,2))
ans =
     1     1     0
     1     0     1
     0     1     1
```

```
>> x = [1 0 0 1 1];
>> y = [0 0 0 0 0];
>> any(x) || any(y)
ans = 1 0 0 1 1
```

```
>> false & 0/0
ans = Error
```

```
>> false && 0/0
ans = 0
```

```
>> (false & true) == (~false) | (~true)
ans = 0
```

4 Loops (16 points)

4.1 Given a partial sum

$$S_n = \sum_{k=1}^{n} \frac{1}{k^2},$$

Euler showed that

$$\lim_{n\to\infty} S_n = \sum_{k=1}^{\infty} \frac{1}{k^2} = \frac{\pi^2}{6}.$$

(4 points) The script below computes the number of terms, **n**, necessary to attain a solution to a tolerance of 10^{-3}, i.e. $|S_n - \pi^2/6| \leq 10^{-3}$. **Circle the line that has a programming mistake:**

```
1  clear;
2  n = 0;
3  d = 1e-3;
4  e = d/2;
5
6  while d <= e
7      Sn = sum((1:n).^-2);
8      e = abs(Sn-pi^2/6);
9      n = n+1;
10 end
```

4.2 (4 points each) Record the output of the following MATLAB commands by completing every blank output line. If the line would return an error, write "error." If the line would start an infinite loop/recursion, write "infinite loop."

```
>> clear;
>> x=1; y=2; z=3;
>> for k=1:3
>>     x=2*x;
>>     while x<5
>>         y=y+1;
>>         z=z*x;
>>         x=y;
>>     end
>> end
```

```
>> x
x = 20
>> y
y = 5
>> z
z = 72
```

5 If-Else and Switch (20 points)

Consider the following function, `mysteryFunction`:

```
function y = mysteryFunction(x)
    y = '';
    if x > 1
        y = [y, 'a'];
    elseif x > 10
        y = [y, 'b'];
    end
    if x < 16
        remain = rem(x, 8);
        switch remain
            case 1
            case {3, 5, 7}
                y = num2str(remain);
            otherwise
                y = remain;
            end
        end
    end
```

Record the output of the following MATLAB commands by completing every blank output line. If the line would return an error, write "error." If the line would start an infinite loop/recursion, write "infinite loop."

(3 points each)

```
>> clear;
>> mysteryFunction(9)
ans = '9'

>> mysteryFunction(3)
ans = '3'

>> mysteryFunction(1)
ans = '1'

>> mysteryFunction(0)
ans = 0

>> mysteryFunction(-1)
ans = '7'
```

(1 point each)

```
>> clear;
>> class(mysteryFunction(9))
ans = char

>> class(mysteryFunction(3))
ans = char

>> class(mysteryFunction(1))
ans = char

>> class(mysteryFunction(0))
ans = double

>> class(mysteryFunction(-1))
ans = char
```

6 Recursion (18 points)

6.1 Consider the following function, `multiplyByRecursion`:

```
function y = multiplyByRecursion(a, b)
    if a == 1
        y = b;
    elseif b == 1
        y = a;
    else
        % ... <------- one line of code goes here
    end
```

`mutliplyByRecursion(a, b)` calculates the product of `a` and `b` using only + or − operators, where `a` and `b` are both positive integers.

(5 points) There is one incomplete line in the code given above. Given choices a – e, which line(s) make(s) the code work properly? **Circle your answer(s).** No partial credit.

(a) `y = multiplyByRecursion(a, b - 1) + a`

(b) `y = multiplyByRecursion(a, b - 1) + b`

(c) `y = multiplyByRecursion(a - 1, b) + a`

(d) `y = multiplyByRecursion(a - 1, b) + b`

(e) `y = multiplyByRecursion(a - 1, b - 1) + a + b - 1`

(5 points) Assume `a = 100, b = 200`. Which **correct** solution(s) from the previous part make(s) the most calls to `multiplyByRecursion`? **Circle your answer(s).** No partial credit.

(a) `y = multiplyByRecursion(a, b - 1) + a`

(b) `y = multiplyByRecursion(a, b - 1) + b`

(c) `y = multiplyByRecursion(a - 1, b) + a`

(d) `y = multiplyByRecursion(a - 1, b) + b`

(e) `y = multiplyByRecursion(a - 1, b - 1) + a + b - 1`

6.2 Consider following function, `sumDownBy2`:

```
function y = sumDownBy2(n)
    if n == 1
        y = 1;
    else
        y = n + sumDownBy2(n-2);
    end
```

9 + sum(7)

(4 points each) Record the output of the following MATLAB commands by completing every blank output line. If the line would return an error, write "error." If the line would start an infinite loop/recursion, write "infinite loop."

```
>> clear;
>> sumDownBy2(9)
```

ans = 25

```
>> sumDownBy2(8)
```

ans = infinite loop

9 + sum(7)
↳ 7 + (5)
↳ 5 + (3)
↳ 3 + (1)
↳ 1

8 (6)
↳ 6 (

7 Linear Algebra (8 points)

Consider the following system of linear equations:

$$\begin{aligned} 3x_2 - 4x_1 + 5x_3 &= 1 \\ 2x_1 - 3x_2 + 5x_2 &= 4 \\ -x_2 + 4x_3 - 2x_1 &= 6, \end{aligned}$$

where the solution x follows the order:

$$\mathrm{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}.$$

(4 points each) Complete the MATLAB code below to solve the system.

```
>> clear;
>> A = [ -4, 3, 5; ...
         2, 2, 0; ...
        -2, -1, 4 ];

>> b = [ 1; 4; 6 ];

>> x = A\b;
```

Function Help Pages

```
>>help mod
```
mod: Modulus after division.

> mod(x,y) returns x - floor(x./y).*y if y~=0, carefully computed to avoid rounding error. If y is not an integer and the quotient x./y is within roundoff error of an integer, then n is that integer. The inputs x and y must be real and have compatible sizes. In the simplest cases, they can be the same size or one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are either the same or one of them is 1. The statement "x and y are congruent mod m" means mod(x,m) == mod(y,m).

By convention:

> mod(x,0) is x.
> mod(x,x) is 0.
> mod(x,y), for x~=y and y~=0, has the same sign as y.

Note: rem(x,y), for x~=y and y~=0, has the same sign as x. mod(x,y) and rem(x,y) are equal if x and y have the same sign, but differ by y if x and y have different signs.

```
>>help rem
```
rem: Remainder after division.

> rem(x,y) returns x - fix(x./y).*y if y~=0, carefully computed to void rounding error. If y is not an integer and the quotient x./y is within roundoff error of an integer, then n is that integer. The inputs x and y must be real and have compatible sizes. In the simplest cases, they can be the same size or one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are either the same or one of them is 1.

By convention:

> rem(x,0) is NaN.
> rem(x,x), for x~=0, is 0.
> rem(x,y), for x~=y and y~=0, has the same sign as x.

Note: mod(x,y), for x~=y and y~=0, has the same sign as y. rem(x,y) and mod(x,y) are equal if x and y have the same sign, but differ by y if x and y have different signs.

```
>> help floor
```
floor: Round towards minus infinity.

> floor(x) rounds the elements of x to the nearest integers towards minus infinity.

```
>> help fix
```
fix: Round towards zero.

> fix(x) rounds the elements of x to the nearest integers towards zero.