1 Multiples of $k$

We basically need to print out $k, 2k, 3k, \ldots, nk$, such that $nk \leq 1000$ and $(n+1)k > 1000$. That is, we start at $k$, stepping by $k$ at a time until the value exceeds 1000. Hence, what goes in the blank is $k:k:1000$.

2 Approximate $\pi$

We will do the approximation using $R_n$ here. Using $T_n$ is similar and is left as an exercise.

Because we need to calculate $R_n$ for $n = 100, 200, \ldots, 1000$, we need a for loop that keeps the current value of $n$. For each $n$, we compute $R_n$ by calculating each term $r_k$ in the summation and accumulate the results using a variable `sum` that serves as the running sum of current $R_n$. That is, the algorithm would look like the following:

```python
for n=100:100:1000
    sum=0;
    for k=1:n
        r_k=(-1)^(k+1)/(2*k-1);
        sum=sum+r_k;
    end
    rho_n=4*sum;
    error=abs(pi-rho_n);
    fprintf('n=%d, error=%.20f\n',n,error);
end
```

This program outputs

```
n=100, error=0.00999975003123942940
n=200, error=0.00499996875097696860
n=300, error=0.00333332407420172670
n=400, error=0.00249999609377882240
n=500, error=0.00199999800000805190
n=600, error=0.00166666550926208860
n=700, error=0.00142857069970858670
n=800, error=0.00124999951171744780
n=900, error=0.00111111076817493880
n=1000, error=0.0009999974999989100
```

Note that we need to reset `sum` for every iteration. Otherwise, old values from the previous calculation would be carried over, and the later iterations would not calculate the correct values of $R_n$. 
One might observe that the program above repeatedly calculates \( r_1, r_2, \ldots, r_{100} \) for every iteration, making the program inefficient. We can simply eliminate these repetitions by keeping calculating \( r_k \) for each \( k \). Once the value of \( k \) reaches a value of \( n \), we know that \( \sum \) must equal \( R_n \) at that point. We then can calculate \( \rho_0, n \) and prints out the error associate with this \( \rho_0, n \). This eliminates the outer for loop, but we have to change the criteria for the other for loop accordingly. The resulting program is presented in the solution on the course webpage.

### 3 Approximate Square Root (Again!)

In Project 1, if we want to do the averaging three times, we need to come up with the following code:

```matlab
%input
A=input('Enter a positive value: ');  
N=input('Enter a positive integer: '); 

%initialize sides
L=A;  
W=1;  
L=(L+W)/2;  
W=A/L;  
L=(L+W)/2;  
W=A/L;  
L=(L+W)/2;  
W=A/L;  
fprintf('sqrt(%f) is approximately %f\n',A,L); 
```

Now, having learned for loop, we can now compact the code above to the following:

```matlab
%input
A=input('Enter a positive value: ');  
N=input('Enter a positive integer: '); 

%initialize sides
L=A;  
W=1;  
for i=1:3
    L=(L+W)/2;
    W=A/L;
end
fprintf('sqrt(%f) is approximately %f\n',A,L); 
```
Now, if we want to do the averaging $N$ times, we simply change 3 to $N$:

```matlab
A=input('Enter a positive value: ');
N=input('Enter a positive integer: ');

%initialize sides
L=A;
W=1;

for i=1:N
    L=(L+W)/2;
    W=A/L;
end

fprintf('sqrt(%f) is approximately %f
',A,L);
```

Last but not least, we need to be a little careful here about the number of times we do the averaging. If the input is $N$, do we average the side $N$ times or $N - 1$ times? Well, when $i = 1$, the second rectangle is generated, i.e., the value of $L$ is the average of the length and width of the first rectangle. Therefore, when $i = N$, the value of $L$ is the average of the length and width of the $N^{th}$ rectangle, which is the final square root value we want to print out, resulting in the code above.

One could do the loop only $N - 1$ times and print out the average of $L$ and $W$ at the end. This, however, can be incorporated into the `for` loop. That is, these two programs are equivalent.