CS1112-206/211/212–Fall 2009

Section 3 Solution

Friday, September 18

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 π

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:

```
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=0.00099999974999898100
```

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. 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_n` and prints out the error associate with this `rho_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 The One-Million-Digit $n!$

In this problem we do not know the final value of n; otherwise we would not have to solve this problem! As in the last section, we will keep the number of digits we have so far of $k!$. We keep incrementing k until we find that the number of digits of $k!$ is at least one million. The point to note here is that we need not calculate the *actual* value of $k!$; we just need to determine the number of digits of $k!$.

First, we know that 1! has 1 digit. That was easy. Now, suppose we know the number of digits of $k!$, which is $\text{floor}(\log_{10}(k!)) + 1$. In particular, we know $\log_{10}(k!)$ as well. How do we find the number of digits of $(k+1)!$? Observe that $(k+1)! = (k+1) \cdot k!$. Hence, the number of digits of $(k+1)!$ is

$$\begin{aligned} \text{floor}(\log_{10}[(k+1)!]) + 1 &= \text{floor}(\log_{10}[(k+1) \cdot k!]) + 1 \\ &= \text{floor}(\log_{10}(k+1) + \log_{10}(k!)) + 1. \end{aligned}$$

Because we already know $\log_{10}(k!)$, we can easily calculate the above quantity by computing $\log_{10}(k+1)$, sum them up, and take the floor.

From the above discussion, we see that we need to keep track of $\log_{10}(k!)$, which is the sum of $\log_{10}(i)$ from $i = 1, 2, \ldots, k$. This results in the following program:

```
n=1;
sum_logs=0;

while floor(sum_logs)+1<1000000
    n=n+1;
    sum_logs=sum_logs+log10(n);
end

fprintf('%d! has at least one million digits.\n',n);
```

It turns out that 205022! has at least one million digits.