#1) I hope you found Mathcad to be a “cool” tool and one that you need to add to your collection. You should also strongly consider Mathematica (more expensive, steeper learning curve, but also more powerful). I look forward to seeing what interesting things students may have done in their 20 minutes.

#2) The set-up is \( F(z) = e^{-\lambda} \sum_{k=0}^{\infty} \frac{(\lambda z)^k}{k!} \). And following from our known expression for \( e^x \), we get \( F(z) = e^{-\lambda(1-z)} \).

#3) Recall that a sum of random variables is a convolution of pdf's which is a multiplication of the \( z \) transforms of the pdf's. Recall also that multiplying exponentials is done by adding their exponents. So, a product of multiple \( F(z) \)'s for a Poisson process results is a Poisson process. For example, two Poisson distributions with the same rate parameter, \( \lambda \), sum to \( F(z) = e^{-2\lambda(1-z)} \), which is the same transform and hence the same distribution (recall that each distribution has a unique transform). This sum of rv's results in the same distribution is a unique property of the Poisson process.

#4) We expect that the system delay will increase with utilization increasing. However, with exponential smoothing the next predicted value can never be greater than the current value (i.e., with \( \alpha = 1 \)). Thus, exponential smoothing is not a suitable technique for predicting future values of a steadily increasing series.

#5) First we need to parse-out the packet length values from trace.txt. I used awk to do this (you can also write a C program to do this). My awk program (split.awk) was a one liner:

```c
{ print ($6) }
```

And, the invocation was

```
awk -f split.awk < trace.txt > pkt.txt
```

I removed the first line of the file pkt.txt (to eliminate the text “bytes”). Then, used summary1.c and summary2.c to get:

```
---------------------------------------------- summary1.c -----
Total of 500000 values
Minimum = 28.000000 (position = 264255)
Maximum = 4470.000000 (position = 160161)
Sum = 362646682.000000
Mean = 725.293364
Variance = 440843.384368
Std Dev = 663.960379
CoV = 0.915437
---------------------------------------------------------------
---------------------------------------------- summary2.c -----
Total of 500000 values
Median = 500.000000
1% value = 40.000000
2% value = 40.000000
5% value = 40.000000
95% value = 1500.000000
98% value = 1500.000000
99% value = 1500.000000
```

```
To get a histogram I used hist.c and put the results (for bin size of 10 and 500 bins) into Excel. The histogram is shown in Figure 1. Packet lengths are distributed bimodally with peaks at 40, 1420, and 1500 bytes. This very likely corresponds to large packets for bulk data transfers and small packets for ACK and other control messages. Note that a maximum length Ethernet packet is 1500 bytes.

![Figure 1 - Histogram of packet lengths from trace.txt](image)

**#6** We need to time a function call. Clearly, one function call is too short in duration for our system clock (of granularity 10 milliseconds), so we need to embed the call within a loop. The test program looks something like:

```c
// Measurement program for assignment #2, problem #3
// - Derivative of timeit.c from tools page
#include <stdio.h> // Needed for printf()
#include <sys/time.h> // Needed for ftime() and timeb structure
double add(double x, double y);

void main(void)
{
    struct timeb start, stop; // Start and stop times structures
double elapsed; // Elapsed time in seconds
double sum, x, y; // Doubles for addends and sum
int i; // Loop counter

    // Initialize x and y
    x = 123456.0;
y = 654321.0;

    // Start timing
ftime(&start);

    for (i=0; i<1000000000; i++) {
        // sum = x + y; // line #1
        // sum = add(x,y); // line #2
    }

    // Stop timing, compute elapsed time, and output it
ftime(&stop);
elapsed = ((double) stop.time + ((double) stop.millitm * 0.001)) -
        ((double) start.time + ((double) start.millitm * 0.001));
printf("Elapsed time = %f sec \n", elapsed);
}

double add(double x, double y)
{
    return(x + y);
}
```
A loop value of 1 billion iterations results in execution times of several seconds on my old 900-Mhz P4 Windows2000 PC. An execution time of several seconds is sufficient to minimize start and stop overhead and clock granularity. To baseline the code I ran it with both line #1 and line #2 commented out and got (for 5 runs):

Elapsed time = 3.264000 sec
Elapsed time = 3.265000 sec
Elapsed time = 3.265000 sec
Elapsed time = 3.264000 sec
Elapsed time = 3.265000 sec

For line #1 executed (inline) but line #2 commented out:

Elapsed time = 15.202000 sec
Elapsed time = 15.192000 sec
Elapsed time = 15.202000 sec
Elapsed time = 15.192000 sec
Elapsed time = 15.202000 sec

And, for line #2 executed (function) but line #1 commented out:

Elapsed time = 24.966000 sec
Elapsed time = 24.966000 sec
Elapsed time = 24.965000 sec
Elapsed time = 24.966000 sec
Elapsed time = 24.966000 sec

We can see that run times have little variance. Subtracting the loop overhead from the inline time run time and dividing by 1 billion we get an execution time of about 11.6 nanoseconds. Subtracting the loop overhead from the function time and dividing by 1 billion we get 21.7 nanoseconds. Thus, it requires an additional about 10 nanoseconds for a function call, or about 47% additional time for the simple two-value addition. Here is the assembly listing from the bcc32 compiler showing just the loop overhead:

```assembly
; for (i=0; i<1000000000; i++)
; xor eax, eax
@3:
    inc eax
    cmp eax,1000000000
    jl short @3
    ;
    ; // sum = x + y;    // line #1
    ; // sum = add(x,y); // line #2
    ;}
```

The inline version (the main loop only) is:

```assembly
; for (i=0; i<1000000000; i++)
; xor eax, eax
;
; 
; sum = x + y;    // line #1
;
?live1@80: ; EAX = i
@2:
    fld qword ptr [ebp-48]
    fadd qword ptr [ebp-56]
    fstp st(0)
    inc eax
    cmp eax,1000000000
    jl short @2
    ;
    ; // sum = add(x,y); // line #2
    ;}
```
And here is the function version:

```assembly
; for (i=0; i<1000000000; i++)
;
xor ebx,ebx
;
{
;   // sum = x + y;   // line #1
;   sum = add(x, y);  // line #2
;
?live1@80: ; EBX = i
@2:  
    push    dword ptr [ebp-52]  
    push    dword ptr [ebp-56]  
    push    dword ptr [ebp-44]  
    push    dword ptr [ebp-48]  
    call    _add
    add     esp,16
    fstp    st(0)
    inc     ebx
    cmp     ebx,1000000000
    jl      short @2
;
; }
```

This shows that it takes several assembler instructions for each line of C code. This also shows that the function is not being inlined.

#7) Extra Credit: I don’t have an answer. I look forward to seeing what the students find.

---