3.1 What happens in the greeting program if, instead of strlen(greeting)+1, we use strlen(greeting) for the length of the message being sent by processes $1, 2, \ldots$,comm˙sz-1? What happens if we use MAX˙STRING instead? Can you explain these results?

If we forget to add 1 to strlen(..) we are not sending the null terminating character of the string in the message, thus there are no guarantees on where the string will terminate when print it out in process 0.
If we use MAX˙STRING, we are still sending the null terminator, thus printf will work as expected, however this time we are simply sending more data than is needed, including a bunch of garbage data after the string.

3.2 Modify the trapezoidal rule so that it will correctly estimate the integral even if comm˙sz doesn't evenly devide n. (You can still assume $n \geq$comm˙sz.)

```c
#include <stdio.h>
#include <string.h>
#include <mpi.h>

double f(double x) {
        return x*x + x + 3;
}

double Trap(double left, double right, int c, double base) {
        double estimate, x;

        estimate = (f(left) + f(right))/2.0;
        for (int i=1; i<c; i++) {
                x = left + i * base;
                estimate += f(x);
        }

        return (estimate * base);
}

int main(int argc, char ** argv) {
        int my_rank, comm_sz, n=512, source, local_n;
```

```
double a = 0.0, b = 1.0, local_a, local_b, h;
double total_int = 0, local_int;

MPI_Init(NULL, NULL);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

// compute the width of each trapezoid,
// as well as how mayn trapezoids
// each process needs to count
// (in order to have 1024 total)
h = (b-a)/n;
local_n = n/comm_sz;

// each process calculates it's local trapezoidal area
// we will sum all of these to obtain an estimate for the integral
local_a = a + my_rank * local_n * h;
local_b = local_a + local_n * h;

if (my_rank == comm_sz-1) {
        // pick up the remaining trapezoid computations
        int remaining = n - (comm_sz * local_n);
        local_int = Trap(local_a, local_b, local_n + remaining, h);
} else {
        local_int = Trap(local_a, local_b, local_n, h);
}

if (my_rank != 0) {
        MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
} else {
        // starting with rank 0's area...
        total_int = local_int;
        // add the area found by all other processes
        for (source=1; source<comm_sz; source++) {
                MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0,
                        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

```
                        total_int += local_int;
                }
        }

        if (my_rank == 0) {
                printf("With n = %d trapezoids, our estiamte\n", n);
                printf("of the integral from %f to %f = %.15e\n",
                        a, b, total_int);
                // Should be about 3.83333...
        }

        MPI_Finalize();
        return 0;
}
```

3.3 Determine which of the variables in the trapezoidal rule program are local and which are global.

MPI runs main n times as specified on mpiexec, thus each process has it's own local copy of all variables declared in main. Because we are not using any other values, they aren't physically shared across each process. However, due to their setup, values such as *a,b,h,local˙n* will be equivalent across all processes (and are not changed by any processes) but are not technically global.

3.4 Modify the program that just prints a line of output from each process (mpi˙output.c) so that the output is printed in process rank order: process 0s output first, then process 1s, and so on.

We could follow the strategy provided by the hello world mpi example. Where only my˙rank 0 does the printing and all others send that process strings. But if we want each process to invoke printf, we can control the timing by using MPI˙Recv to cause a process to wait until the previous process is done printing, once that process is done printing, it will invoke MPI˙Send to let the next process know it's okay to print.

```
#include <stdio.h>
#include <mpi.h>
```

```
int main(int argv, char ** argc) {
        int my_rank, comm_sz, recv;

        MPI_Init(NULL, NULL);
        MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
        MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

        // wait for the previous process to send a message before printing
        // the previous process will send it's message once it has finished
        // printing, the initial process my_rank = 0 does not have to wait
        if (my_rank > 0) {
                MPI_Recv(&recv, 1, MPI_INT, my_rank-1, 0, MPI_COMM_WORLD, MPI_STATUS_I
        }

        printf("Proc %d of %d > Does anyone have a toothpick?\n",
                my_rank, comm_sz);

        // done printing, send message to next process and let it print
        if (my_rank < comm_sz-1) {
                MPI_Send(&my_rank, 1, MPI_INT, my_rank+1, 0, MPI_COMM_WORLD);
        }

        MPI_Finalize();
        return 0;
}
```