<hr style=

Homework #4

In this assignment, you will implement a routines that manage "collective communication"

- Problem #1 - Computing statistics (serial)
- Problem #2 - Computing statistics (parallel)
- Problem #3 - Compute derivative and do convergence study. Write results to a file.

In [1]:
```
from numpy import *
from matplotlib.pyplot import *
```

Problem #1 (Statistics in serial)

In this problem, you will assume that you have a stream of data that is coming in continuously. As the data comes in, you want to compute the mean, variance and standard deviation of the data you have seen so far.

To compute the statistics, you will use "incrememtal algorithnms" to update the mean and a sum-of-squared-differences as each new data point comes in.

Mathematical formulas

Formulas for the mean, variance and standard deviation are given by

$$\begin{aligned}
\overline{x}_n &\equiv \frac{1}{n}\sum_{i=1}^{n} x_i, &\qquad& \text{(Mean)} \\
S^2 &\equiv \frac{1}{n}\sum_{i=1}^{n} (x_i - \overline{x}_n)^2, && \text{(Sample variance)} \\
S &\equiv \sqrt{\frac{1}{n}\sum_{i=1}^{n} (x_i - \overline{x}_n)^2}, && \text{(Standard deviation)}
\end{aligned} \tag{1}$$

Incremental formulas

It can be shown that the mean can be incrementally updated using

$$\overline{x}_n = \overline{x}_{n-1} + \frac{x_n - \overline{x}_{n-1}}{n}, \qquad n = 2, \ldots, \tag{2}$$

We define the sum of squared differences as

$$M_n \equiv \sum_{i=1}^{n}(x_i - \overline{x}_n)^2 \tag{3}$$

Then M_n can be incrementally updated using

$$M_n = M_{n-1} + (x_n - \overline{x}_n)(x_n - \overline{x}_{n-1}) \tag{4}$$

The resulting algorithms is essentially the serial *Welford's Algorithm* for computing these statistics.

To Do

- Show that the above formulas are correct. Include your derivations in Latex in this notebook.
- Use Python code below to create a "large" data set of N random numbers. Write the file to a binary file to be read in C.
- Write a function that implements the formulas for incrementally computing the mean and sum-of-squared-differences M_N formulas. Your function should have the signature

  ```
  void compute_stats(FILE *file, int n, double *mean, double
  *sum2)
  ```

  where `file` is a file handle to the location in the binary data file containing `n` data items to be read. The returns are `mean` and `sum2`. Assume that `n` data items cannot all fit into memory, so that you can only read one item at a time from the data file.
- Compute the variance and the standard deviation M_N.
- Compare your results to those obtained from Python. Your Python and C results should be essentially the same.

For this problem, set $N = 2^{20}$. For this problem, you only need to run on a single processor.

Tips

- Use `fread` to read a single value from the data set.

```
double x;
fread(&x,sizeof(double),1,file);
```

- You do not need to allocate any memory for this problem.

Derivations of incremental formulas

TODO : Include derivations of formulas here. You may do these directly in the notebook using LaTex (easiest!) or scan in hand-written a derivation and include it here using HTML :

Question 1

(a)

$$\bar{x}_n = \frac{1}{n}\sum_{i=1}^{n} x_i$$

$$n\bar{x}_n = \sum_{i=1}^{n} x_i$$

$$= x_n + \sum_{i=1}^{n-1} x_i$$

But $\sum_{i=1}^{n-1} x_i = (n-1)\bar{x}_{n-1}$

$$n\bar{x}_n = x_n + (n-1)\bar{x}_{n-1}$$

$$\bar{x}_n = \frac{x_n + (n-1)\bar{x}_{n-1}}{n} = \bar{x}_{n-1} + \frac{(x_n - \bar{x}_{n-1})}{n}$$

(b)

$$M_n \equiv \sum_{i=1}^{n} (x_i - \bar{x}_n)^2$$

$$M_n = \sum_{i=1}^{n} (x_i^2 - 2\bar{x}_n x_i + \bar{x}_n^2)$$

$$= \sum_{i=1}^{n} x_i^2 - 2\bar{x}_n \sum_{i=1}^{n} x_i + n\bar{x}^2$$

$$= \sum_{i=1}^{n} x_i^2 - 2n\bar{x}_n^2 + n\bar{x}^2$$

$$M_n = \sum_{i=1}^{n} x_i^2 - n\bar{x}_n^2$$

Intuitively, we can generate an expression for M_{n-1}, so that;

$$M_n - M_{n-1} = \left[\sum_{i=1}^{n} x_i^2 - n\bar{x}_n^2\right] - \left[\sum_{i=1}^{n-1} x_i^2 - (n-1)\bar{x}_{n-1}^2\right]$$

$$= \sum_{i=1}^{n} x_i^2 - n\bar{x}_n^2 - \sum_{i=1}^{n-1} x_i^2 + (n-1)\bar{x}_{n-1}^2$$

$$= \left[\sum_{i=1}^{n} x_i^2 - \sum_{i=1}^{n-1} x_i^2\right] - n\bar{x}_n^2 + (n-1)\bar{x}_{n-1}^2$$

$$= x_n^2 - n\bar{x}_n^2 + (n-1)\bar{x}_{n-1}^2$$

$$M_n - M_{n-1} = x_n^2 + n(\bar{x}_{n-1}^2 - \bar{x}_n^2) - \bar{x}_{n-1}^2$$

$$M_n - M_{n-1} = x_n^2 + n(\bar{x}_{n-1} - \bar{x}_n)(\bar{x}_{n-1} + \bar{x}_n) - \bar{x}_{n-1}^2 \quad - \textcircled{1}$$

From part a)

$$\bar{x}_n = \bar{x}_{n-1} + \frac{(x_n - \bar{x}_{n-1})}{n}$$

$$\bar{x}_{n-1} - \bar{x}_n = \frac{\bar{x}_{n-1} - x_n}{n} \quad - \textcircled{2}$$

Substituting $\textcircled{2}$ into $\textcircled{1}$

$$M_n - M_{n-1} = x_n^2 + n\left[\frac{\bar{x}_{n-1} - x_n}{n}\right](\bar{x}_{n-1} + \bar{x}_n) - \bar{x}_{n-1}^2$$

$$= x_n^2 + (\bar{x}_{n-1} - x_n)(\bar{x}_{n-1} + \bar{x}_n) - \bar{x}_{n-1}^2$$

$$= x_n^2 + \bar{x}_{n-1}^2 + \bar{x}_n\bar{x}_{n-1} - x_n\bar{x}_{n-1} - x_n\bar{x}_n - \bar{x}_{n-1}^2$$

$$= x_n^2 + \bar{x}_n\bar{x}_{n-1} - x_n\bar{x}_{n-1} - \bar{x}_n x_n$$

$$M_n - M_{n-1} = (x_n - \bar{x}_n)(x_n - \bar{x}_{n-1})$$

$$M_n = M_{n-1} + (x_n - \bar{x}_n)(x_n - \bar{x}_{n-1}).$$

If you include scanned work, be sure it shows up in your final PDF. You may need to test different browsers and methods of export to PDF to really see the image show up.

(prob1) Generate data

Create data array of length N and save it to binary file `X.dat`. Store the value of N as meta data in the same file.

In [2]:
```
set_printoptions(formatter={'float' : "{:24.16}".format})
N = 1 << 20;     # 2**20

X = random.rand(N)

fout = open("X.dat","wb")
array([N],dtype=int32).tofile(fout)
X.tofile(fout)
fout.close()
```

(prob1) Compute statistics

Complete the routine `compute_stats` below to compute mean and sum-of-squared-differences.

In [3]:
```
%%file prob1.c

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include <math.h>

void compute_stats(FILE *file, int n, double *mean, double *sum2)
{
    /* # TODO : Implement incremental algorithm for mean and sum2.
       # Use only 'fread' to read 1 item at time from `file`.
    */
    int i = 0;
    double dm, value;
    *mean = 0;
    *sum2 = 0;

    do {
        fread(&value, sizeof(double), 1, file);
        dm = value - *mean;
        *mean +=   dm/(double)(i + 1);
        *sum2 += dm * (value - *mean);
        i++;
    } while (i <n);

}


int main(int argc, char** argv)
{
    FILE *file=fopen("X.dat","r");
    int N;
    fread(&N,sizeof(int),1,file);
    double mean, sum2;
    compute_stats(file,N,&mean, &sum2);
    fclose(file);

    printf("Mean  (C)       = %24.16f\n",mean);
    printf("Sum2  (C)       = %24.16f\n",sum2);
    printf("STD   (C)       = %24.16f\n",sqrt(sum2/N));

    return 0;
}
```

```
Overwriting prob1.c
```

(prob1) Compile and run code

In [4]:
```
%%bash

rm -rf prob1

gcc -o prob1 prob1.c

./prob1
```

```
Mean  (C)      =      0.5004424265985566
Sum2  (C)      =  87276.1699317541060736
STD   (C)      =      0.2885013720907083
```

(prob1) Verify results

In [5]:
```
print(f"Mean (Python) is {X.mean():24.16f}")

sum2 = sum((X-X.mean())**2)
print(f"Sum2 (Python) is {sum2:24.16f}")
print(f"STD  (Python) is {X.std():24.16f}")
```

```
Mean (Python) is       0.5004424265985371
Sum2 (Python) is   87276.1699317570892163
STD  (Python) is       0.2885013720907132
```

In [6]:
```
# relative error: (mean_c - mean_python)/mean_python 10-15
```

Problem #2 (Statistics in parallel)

For this problem, you again have a data set that is too large to fit into memory. This time, you want to read values from the data set one at a time. Each processor will read from the same file, starting at a location determined by the processor rank.

As in problem #1, you will use *incremental formulas* to compute the mean $\overline{x}_N$ and the sum-of-squared-differences M_N of N data values. But this time, each processor will compute the mean $\overline{x}_n$ and M_n of a chunk of n data values. Then, rank 0 will gather these mean values and combine them incrementally to form the mean, variance and standard deviation of the entire set.

Mathematical formulas

Let $\overline{X}_A$ and $\overline{X}_B$ be the mean of two data sets X_A and X_B, respectively. Then the mean of the combined data set $X_X = X_A \cup X_B$ is given by

$$\overline{X}_X = \overline{X}_A + \left(\overline{X}_B - \overline{X}_A\right)\frac{N_B}{N_X} \tag{5}$$

where N_A and N_B are the number of data elements in X_A and X_B, respectively.

Let M_A and M_B be the sum-of-squared differences of two data sets X_A and X_B. Then the sum-of-squared-differences M_X of the two combined data sets X_X is given by

$$M_X = M_A + M_B + \left(\overline{X}_B - \overline{X}_A\right)^2 \frac{N_A N_B}{N_X} \tag{6}$$

The goal of this problem is to compute the mean, variance and standard deviation of a large data set using multiple processors. Each process will compute the mean and sum-of-squared-differences for a chunk of data, read in one at a time from a file. The rank 0 will collect these results and use the incremental update for sets to form the true mean, variance and standard deviation of the full data set.

To do

- Verify the two incremental formulas above.
- Create a large data set
- Implement the incremental formula in C. Use MPI to compute statistics on multiple processors.

Question 2

a)

$$\bar{X}_x = \frac{1}{N_x} \sum_{i=1}^{N_x} x_i$$

Since $X_x = X_A \cup X_B$, $N_x = N_A + N_B$

$$\bar{X}_x = \frac{1}{N_x}\left[\sum_{i=1}^{N_A} x_i + \sum_{i=1}^{N_B} x_i \right]$$

$$\bar{X}_x = \frac{N_A}{N_x} \times \frac{1}{N_A} \sum_{i=1}^{N_A} x_i + \frac{N_B}{N_x} \times \frac{1}{N_B} \sum_{i=1}^{N_B} x_i$$

$$\bar{X}_x = \frac{N_A}{N_x} \bar{X}_A + \frac{N_B}{N_x} \bar{X}_B \quad - \textcircled{1}$$

Substituting $N_A = N_x - N_B$ into $\textcircled{1}$, we obtain;

$$\bar{X}_x = \frac{(N_x - N_B)}{N_x} \bar{X}_A + \frac{N_B}{N_x} \bar{X}_B$$

$$N_x \bar{X}_x = N_x \bar{X}_A - N_B \bar{X}_A + N_B \bar{X}_B$$

$$= N_x \bar{X}_A + (\bar{X}_B - \bar{X}_A) N_B$$

$$\bar{X}_x = \bar{X}_A + (\bar{X}_B - \bar{X}_A) \frac{N_B}{N_x}$$

(2b)

$$\bar{X}_x = \bar{X}_A + (\bar{X}_B - \bar{X}_A)\frac{N_B}{N_X} \quad \text{— (1)}$$

We want to show that:

$$M_x = M_A + M_B + (\bar{X}_B - \bar{X}_A)^2 \frac{N_A N_B}{N_X}$$

By definition, $M_X = \sum_{i=1}^{N_X} (X_{xi} - \bar{X}_x)^2$.

$$M_x = \sum_{i=1}^{N_x} (X_{xi} - \bar{X}_x)^2$$
$$= \sum_{i=1}^{N_x} (X_{xi}^2 - 2\bar{X}_x X_{xi} + \bar{X}_x^2)$$
$$= \sum_{i=1}^{N_X} X_{xi}^2 - 2\bar{X}_x \sum_{i=1}^{N_x} X_{xi} + N_X \bar{X}_x^2$$
$$= \sum_{i=1}^{N_X} X_{xi}^2 - 2N_X \bar{X}_x^2 + N_X \bar{X}_x^2$$
$$M_x = \sum_{i=1}^{N_X} X_{xi}^2 - N_X \bar{X}_x^2 \quad \text{— (2)}$$

$$M_A = \sum_{i=1}^{N_A} (X_{Ai} - \bar{X}_A)^2 = \sum_{i=1}^{N_A} X_{Ai}^2 - N_A \bar{X}_A^2$$

Similarly, $M_B = \sum_{i=1}^{N_B} X_{Bi}^2 - N_B \bar{X}_B^2$

$$M_A + M_B = \sum_{i=1}^{N_A} X_{Ai}^2 + \sum_{i=1}^{N_B} X_{Bi}^2 - N_A \bar{X}_A^2 - N_B \bar{X}_B^2$$

$$M_A + M_B + N_A \bar{X}_A^2 + N_B \bar{X}_B^2 = \sum_{i=1}^{N_A} X_{Ai}^2 + \sum_{i=1}^{N_B} X_{Bi}^2 = \sum_{i=1}^{N_X} X_{xi}^2 \quad \text{— (3)}$$

Substituting (3) for $\sum_{i=1}^{N_X} X_{xi}^2$ in (2), we obtain:

$$M_x = M_A + M_B + N_A \bar{X}_A^2 + N_B \bar{X}_B^2 - N_X \bar{X}_x^2 \quad \text{— (4)}$$

Substituting (1) in (4) we obtain;

$$M_x = M_A + M_B + N_A \bar{x}_A^2 + N_B \bar{x}_B^2 - N_x\left[\bar{x}_A + (\bar{x}_B - \bar{x}_A)\frac{N_B}{N_x}\right]^2$$

$$= M_A + M_B + N_A\bar{x}_A^2 + N_B\bar{x}_B^2 - N_x\bar{x}_A^2 - (\bar{x}_B-\bar{x}_A)^2\frac{N_B^2}{N_x} - 2\bar{x}_A(\bar{x}_B-\bar{x}_A)N_B$$

$$= M_A + M_B + N_B\bar{x}_B^2 - 2N_B\bar{x}_A\bar{x}_B - \bar{x}_A^2(N_A+N_B) + 2\bar{x}_A^2 N_B - (\bar{x}_B-\bar{x}_A)^2\frac{N_B^2}{N_x}$$

$$= M_A + M_B + N_B\bar{x}_B^2 + N_B\bar{x}_A^2 - 2N_B\bar{x}_A\bar{x}_B - (\bar{x}_B-\bar{x}_A)^2\frac{N_B^2}{N_x}$$

$$= M_A + M_B + N_B(\bar{x}_A^2 + \bar{x}_B^2 - 2\bar{x}_A\bar{x}_B) - (\bar{x}_B-\bar{x}_A)^2\frac{N_B^2}{N_x}$$

$$= M_A + M_B + (\bar{x}_B-\bar{x}_A)^2 N_B - (\bar{x}_B-\bar{x}_A)^2\frac{N_B^2}{N_x}$$

$$= M_A + M_B + (\bar{x}_B-\bar{x}_A) N_B\left[1 - \frac{N_B}{N_x}\right]$$

$$M_x = M_A + M_B + (\bar{x}_B-\bar{x}_A)^2\frac{N_B N_A}{N_x}$$

(prob2) Create large data set

In [7]:
```
set_printoptions(formatter={'float' : "{:24.16}".format})
N = 1 << 20;

X = random.rand(N)
if N < 32:
    display(X)
else:
    print("N is too large to display.")

fout = open("X.dat","wb")
array([N],dtype=int32).tofile(fout)
X.tofile(fout)
fout.close()
```

```
N is too large to display.
```

(prob2) Compute statistics in C (parallel)

```
In [ ]:
```

```
In [8]: %%file prob2.c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <mpi.h>

void compute_stats(FILE *file, int n, double *mean, double *sum2)
{
    int i = 0;
    double dm, value;
    *mean = 0;
    *sum2 = 0;

    do {
        fread(&value, sizeof(double), 1, file);
        dm = value - *mean;
        *mean += dm/(i + 1);
        *sum2 += dm * (value - *mean);
        i++;
    } while (i < n);
}

int main(int argc, char** argv)
{
    MPI_Init(&argc, &argv);

    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int nprocs;
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

    FILE *file=fopen("X.dat","r");
    int N;
    fread(&N,sizeof(int),1,file);

    int N_local = N/nprocs;
    double mean_local, sum2_local;

    fseek(file, rank * N_local * sizeof(double), SEEK_CUR);
    compute_stats(file, N_local, &mean_local, &sum2_local);
    fclose(file);

    MPI_Send(&mean_local, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    MPI_Send(&sum2_local, 1, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD);
    double mean, sum2;
    mean = 0;
    sum2 = 0;

    if (rank == 0)
    {
        int NT = N_local;
        mean = mean_local;
        sum2 = sum2_local;
        for (int i = 1; i < nprocs; i++)
        {
            MPI_Recv(&mean_local, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, MPI_S
            MPI_Recv(&sum2_local, 1, MPI_DOUBLE, i, 1, MPI_COMM_WORLD, MPI_S
            double delta = mean_local - mean;
            mean += delta * N_local / ((i + 1) *N_local);
            sum2 += sum2_local +  (pow(delta,2)* NT * N_local)/((i + 1) *N_l
            NT +=N_local;
        }

        printf("Mean   (C)       = %24.16f\n", mean);
        printf("Sum2   (C)       = %24.16f\n", sum2);
        printf("STD    (C)       = %24.16f\n",sqrt(sum2/N));
    }

    MPI_Finalize();

    return 0;
}
```

```
Overwriting prob2.c
```

(prob2) Compile and run code

In [9]:
```
%%bash

rm -rf prob2

mpicc -o prob2 prob2.c

mpirun -n 8 ./prob2
```

```
Mean   (C)       =       0.5001490177052933
Sum2   (C)       =   87323.4490369770646794
STD    (C)       =       0.2885795047583510
```

In [ ]:

(prob2) Verify results

In [10]:
```
print(f"Mean (Python) is {X.mean():24.16f}")

sum2 = sum((X-X.mean())**2)
print(f"Sum2 (Python) is {sum2:24.16f}")
print(f"STD  (Python) is {X.std():24.16f}")
```

```
Mean (Python) is       0.5001490177052945
Sum2 (Python) is   87323.4490369764680509
STD  (Python) is       0.2885795047583500
```

In [ ]:

Problem #3 : Truncation error

When developing numerical codes it is often useful as a verification step of code correctness to confirm the expected order of accuracy of a numerical scheme. Below, you will confirm the second order accuracy of a finite difference scheme by numerically computing the *truncation error* of the scheme.

For this problem, you are going to compute the truncation error for the centered finite difference scheme approximation to the second derivative, given by

$$u''(x_i) \approx \frac{u_{i-1} - 2u_i + u_{i+1}}{h^2}, \qquad i = 0, 1, \ldots N \tag{7}$$

where $u_i = u(x_i)$, $x_i = a + ih$, $h = (b-a)/N$ on an interval $[a, b]$.

Truncation error

The *truncation error* $\tau(x; h)$ at a grid point x_i associated with the centered difference approximation is given by

$$\tau(x_i; h) = \left| \frac{u_{i-1} - 2u_i + u_{i+1}}{h^2} - u''(x_i) \right| = \frac{1}{12} u''''(x_i)\; h^2 + \mathcal{O}(h^4). \tag{8}$$

The underlying idea is that as h (*mesh width*) goes to zero, this trunction error goes to 0 and our finite difference approximation at x_i approaches the exact second derivative $u''(x_i)$. *This particular form of the truncation error can be easily found using Taylor series expansions learned in Calculus I.*

The centered difference scheme is referred to as a *second order scheme*, since $\tau(x; h) \sim \mathcal{O}(h^2)$ in the limit as h goes to zero.

Numerical verification of the order of accuracy

In practice, we can confirm the order accuracy of a numerical method by neglecting the higher order terms in the truncation error and defining a numerical trunction error as

$$\tau_i \equiv \tau(x_i; h) \approx Ch^p \tag{9}$$

For the second order, centered difference scheme above, the constant C is depends on the fourth derivative of $u(x)$, or

$$C \approx \frac{1}{12}u''''(x_i). \tag{10}$$

If we halve the mesh width h for the centered scheme, we expect that the trunction error is reduced by a quarter, since

$$\tau(x_i; h/2) \approx C\left(\frac{h}{2}\right)^2 = \frac{C}{4}h^2 = \frac{1}{4}\tau(x_i; h). \tag{11}$$

To confirm the second order accuracy of the truncation error, we compute a sequence of values N_m and T_m, $m = 0, 1, \ldots, M$ as follows.

- Define $N_m = N_0 2^m$.
- Define $h_m = (b-a)/N_m$ and $x_i = a + ih_m$ for $i = 0, 1, \ldots N_m$.

Then

1. Compute $\tau_i = \tau(x_i; h_m)$ at each grid point x_i. Call this vector $\mathbf{t}$, with components τ_i.

1. Compute the inf-norm $\|\cdot\|_\infty$ of the vector $\mathbf{t}$ as

$$\|\mathbf{t}\|_\infty^{N_m} = \max_{i=0,1,\ldots N_m} |\tau_i| \tag{12}$$

1. Define $T_m = \|\mathbf{t}\|_\infty^{N_m}$.

Using vectors $\mathbf{H}$ and $\mathbf{T}$ with components h_m and T_m, we seek a parameter p that fits the model

$$T_m = Ch_m^p. \tag{13}$$

Taking the log of both sides, we get the linear expression

$$\log(T_m) = \log(h_m)\, p + \log(C) \tag{14}$$

Using linear regression (e.g. `polyfit`), we can find a value p (e.g. *slope*) that best fits our linear model. For the centered difference scheme, this slope should be approximately 2.

An alternative formulation that will give us the same slope (with opposite sign) is

$$\log(T_m) = \log(N_m)\, p + \log(C) \tag{15}$$

This is more convenient form, since we can label the x-axis with integer N values rather than small h values.

To Do

Complete the code below.

In [11]:
```
%reset -f
```

In [12]:
```
%%file prob3.c

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include <mpi.h>

#include <math.h>
double* allocate_1d(int n, int m)
{
    /* Use malloc to allocate memory */
    double *y = (double*) malloc((n+2*m)*sizeof(double));
    return &y[m];
}


void free_1d(double **x, int m)
{
    /* Use free to free memory;  Set value of pointer to NULL after freeing
    if (*x != NULL)
    {
        free(*x+m);
        *x = NULL;
    }

}


double utrue(double x)
{
    return cos(x);
```

```
}

double upp(double x)
{
    return -cos(x);
}

/* Evaluate true solution */
void compute_solution(int n, double a, double b, double *u)
{
    int rank, nprocs;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

    int i1 = 0;
    int i2 = (rank == nprocs-1) ? n+1 : n;

    double dx = (b-a)/n;
    for(int i = i1; i < i2; i++)
    {
        double x = a + dx*i;
        u[i] = utrue(x);

    }
}


void compute_rhs(int n, double a, double b, double *f)
{
    double dx = (b-a)/n;
    for(int i = 0; i < n+1; i++)
    {
        double x = a + (dx*i);
        f[i] = upp(x);
    }
}

void apply_Laplacian(int n, double a, double b, double *u, double *L)
{
    int rank, nprocs;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

    /* # TODO : Evaluate the Laplacian L.  You will need to exchange data at
       # each processor region. */
    double h = (b - a)/n;
    int i2 = (rank == nprocs-1) ? n: n-1;
    int i1 = 0;

    if (rank > 0)
    {
        int tag = 0;
        int sender = rank - 1;
        MPI_Recv(&u[-1],1,MPI_DOUBLE,sender,tag,MPI_COMM_WORLD, MPI_STATUS_I
        tag = 1;
        int dest = rank - 1;
        MPI_Send(&u[0],1,MPI_DOUBLE,dest,tag,MPI_COMM_WORLD);
    }
    if (rank < nprocs - 1)
    {
        int tag = 0;
        int dest = rank + 1;
        MPI_Send(&u[n-1],1,MPI_DOUBLE,dest,tag,MPI_COMM_WORLD);

        tag = 1;
        int sender = rank + 1;
        MPI_Recv(&u[n],1,MPI_DOUBLE,sender,tag,MPI_COMM_WORLD, MPI_STATUS_IG
    }

    if (rank == 0)
    {
        u[-1] = 2*u[0] - u[1];


    }

    if (rank == nprocs -1)
    {
        u[n+1] = 2*u[n] - u[n - 1];
    }



    for (int j = i1; j <= i2; j++)
    {
        L[j] = (u[j - 1] - 2*u[j] + u[j+1])/pow(h,2);

    }

}


double compute_grid_error(int n, double a, double b, double *u, double *v)
{
    int rank, nprocs;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

    // Compute local maximum error
    double local_max_error = 0.0;
    int i1 = (rank == 0) ? 1: 0;
    int i2 = (rank == nprocs - 1) ? n : n-1;

    for (int i = i1; i < i2; i++) {
        double error = fabs(u[i] - v[i]);
        if (error > local_max_error) {
            local_max_error = error;
        }
    }

    // Find global maximum error
    double global_max_error;
    MPI_Allreduce(&local_max_error, &global_max_error, 1, MPI_DOUBLE, MPI_MA

    return global_max_error;
}


int main(int argc, char** argv)
{
    MPI_Init(&argc, &argv);

    int rank, nprocs;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

    if (argc < 2)
    {
        printf("User must supply N as a command line argument.\n");
        exit(1);
    }

    int N = atoi(argv[1]);
    double a = 0, b = 2*M_PI;
    int N_local = N/nprocs;

    double h = (b - a) / N;

    double a_local = a + rank * N_local * h;
    double b_local = a_local + N_local * h;

    FILE *file=fopen("prob3.dat","r");
    fread(&a,sizeof(double),1,file);
    fread(&b,sizeof(double),1,file);
    fclose(file);

    /* # TODO : Determine interval [a_p, b_p] for this processor */
    double *u_local = allocate_1d(N_local+1,0);
    double *L_local = allocate_1d(N_local+1,0);
    double *v_local = allocate_1d(N_local+1,0);

    /* # TODO : Create array of u values (use 'utrue') */
    compute_solution(N_local, a_local, b_local, u_local);
    apply_Laplacian(N_local,a_local, b_local, u_local, L_local);
    compute_rhs(N_local, a_local,b_local,v_local);

    double g;
    g = compute_grid_error(N_local,a_local, b_local, L_local, v_local);
    double g_all;
```

```
    MPI_Reduce(&g, &g_all, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);

    if (rank == 0)
    {
        /* # TODO  : Use MPI_Reduce to compute norm of truncation error for

        FILE *fout = fopen("prob3.out","w");
        fwrite(&g_all,sizeof(double),1,fout);
        fclose(fout);
    }
    free_1d(&u_local,0);
    free_1d(&L_local,0);
    free_1d(&v_local,0);

    MPI_Finalize();
}
```

```
Overwriting prob3.c
```

In [13]:

```
%%bash

rm -rf prob3

mpicc -o prob3 prob3.c
```

In [14]:

```
from numpy import *
```

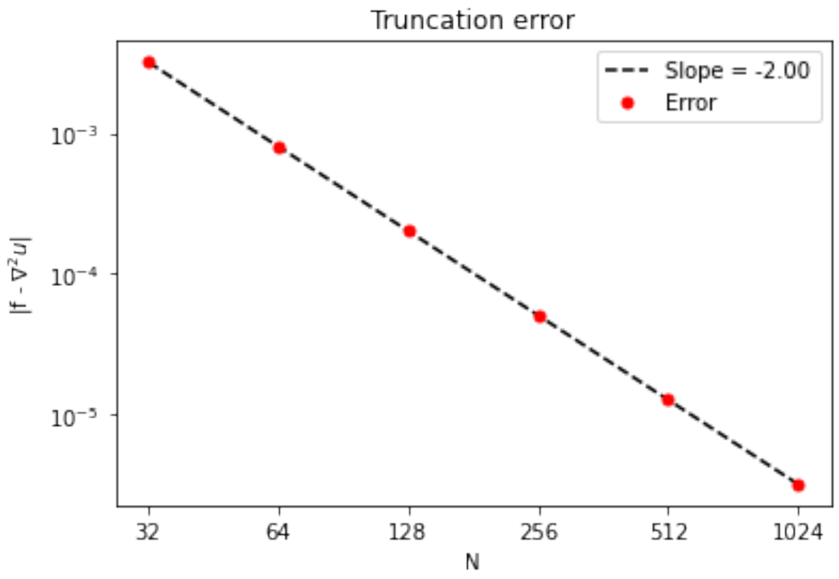
In [18]:
```
import subprocess
import shlex
import os

a = 0
b = 2*pi

# mpirun command
shell_cmd = 'mpirun -n 4 ./prob3 {N:d}'.format

fout = open("prob3.dat","wb")
array([a,b],dtype=float64).tofile(fout)
fout.close()

err = []
Nv = []
M = 5
N0 = 32
for p in range(M+1):
    N = N0*(2**p);
    cmd = shell_cmd(N=N)
    arg_list = shlex.split(cmd)

    # Run output
    output = subprocess.run(arg_list)

    fout = open("prob3.out","rb")
    g = fromfile(fout,dtype=float64,count=1)[0]
    fout.close()

    print(f"{N:8d} {g:16.4e}")

    err.append(g)
    Nv.append(N)

err = array(err)
Nv = array(Nv)
```

```
      32       3.2086e-03
      64       8.0293e-04
     128       2.0078e-04
     256       5.0198e-05
     512       1.2550e-05
    1024       3.1375e-06
```

Create plot of error

In [19]:
```
from matplotlib.pyplot import *
```

In [20]:
```
figure(1)
clf()

# TODO : Get best fit line to (N,err)

# Plot best fit line (loglog)
ps = polyfit(log(Nv),log(err),1)
label = f"Slope = {ps[0]:0.2f}"
loglog(Nv,exp(polyval(ps,log(Nv))),'k--',label=label)

# Plot actual data points (loglog)
loglog(Nv,err,'r.',ms=10,label='Error')


# Make nice tick marks
pstr = ([f'{N:d}' for N in Nv])
xticks(Nv,pstr)

minorticks_off();  # Needed to suppress minor tick mark labels

legend();

title("Truncation error")
xlabel("N")
ylabel(r"|f - $\nabla^2 u|$");
```



In [ ]: