Homework #3

The learning goals for this assignment are

- Practice handling arrays in C
- Implementation of a finite difference scheme in C
- Practice using MPI
- Practice writing files in C in binary format and reading them back into Python for plotting.

Problems;

- Problem #1 - Memory allocation
- Problem #2 - Finite difference scheme
- Problem #3 - Evaluate and plot a two dimensional shifted sinc function
- Problem #4 - Fix an unreliable code
- Problem #5 - Fix another unreliable code
- Problem #6 - Computing a maximum value (MPI)
- Problem #7 - Plot a circle
- Problem #8 - Plot a spiral (COMPUT/ME 571)

Note

At the start of each problem, you'll see a line magic command `%reset -f`. This is essentially equivalent to selecting `restart` from the Kernel menu.

Problem #1 - Memory allocation

Write a function that allocates memory and returns a pointer to the allocated memory. Your allocation should include space for extra values at the beginning and end of the array. This will be useful for implementing finite difference schemes.

Example

Suppose we want to allocate memory for an array $\mathbf{x}$ of $N+1$ equally spaced points in an interval $[a,b]$. In addition, we want to include m extra values at the beginning of the array and m values and the end of the array. The entries of $\mathbf{x}$ will then be given by

$$x_i = a + hi, \quad i = -m, -m+1, \ldots, 0, 1, \ldots, N, N+1, \ldots, N+m \tag{1}$$

where $h = (b-a)/N$. With the above, we will have $x_0 = a$ and $x_N = b$.

If we set $N = 4$ and $m = 1$ and call our function as

```
double *x = allocate_1d(N+1,m);
```

we should be able to index entries as

```
x[-1], x[0], x[1], x[2], x[3], x[4], x[5]
```

The total memory allocated in this case is $N + 1 + 2m = 7$.

To do

- Write the function `allocate_1d` with signature

  ```
  double  *y allocate_1d(int n, int m)
  ```

- Use your function to allocate memory for an array $\mathbf{x}$ of equally spaced points on an interval $[a,b]$. Use values $N = 8$ and $m = 1$ so that you allocate memory for $N + 1 + 2 = 11$ double values.

- Using values $a = 0$ and $b = 1$, assign values to your array $\mathbf{x}$ so that your entries in your array are

  ```
       i     x[i]
  --------------
      -1   -0.125
       0    0.000
       1    0.125
       2    0.250
       3    0.375
       4    0.500
       5    0.625
       6    0.750
       7    0.875
       8    1.000
       9    1.125
  ```

- Print out values of your array in a table as above.

- Write a function that de-allocates the memory.

Tips

- Call `malloc` in the your function `allocate_1d`.
- Return a pointer to a value other than the address at the start of the array.
- The functions `malloc` and `free` should always appear in pairs. For example.

  ```
  double *array = (double *) malloc(5*sizeof(double));

  /* ... use array ... */

  free(array)
  ```

  For this problem, you should include a second function `free_1d` that frees any memory allocated in your function `allocate_1d`.
- Much of the code is written below. You need to supply code for the functions that allocate and de-allocate arrays.
- Remove the `#if 0` and `#endif` to test your code.

In [1]:
```
%reset -f
```

In [2]:
```
%%file prob1.c

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

double* allocate_1d(int n, int m)
{
    /* Use malloc to allocate memory */
    double *y = (double*) malloc((n+2*m)*sizeof(double));
    return y;
}


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

}

int main(int argv, char**argc)
{
    int N = 8;
    int m = 1;
    double a = 0;
    double b = 1;
```

```
    double *x = allocate_1d(N+1,m);

    double h = (b-a)/N;

    printf("%5s %8s\n","i","x[i]");
    printf("--------------\n");
    for(int i = -m; i < N+m+1; i++)
    {
        x[i] = a + i*h;
        printf("%5d %8.4f\n",i,x[i]);

    }

    free_1d(&x,0);


    return 0;
}
```

```
Overwriting prob1.c
```

In [3]:
```
%%bash

rm -rf prob1

gcc -o prob1 prob1.c

./prob1
```

```
    i     x[i]
--------------
   -1  -0.1250
    0   0.0000
    1   0.1250
    2   0.2500
    3   0.3750
    4   0.5000
    5   0.6250
    6   0.7500
    7   0.8750
    8   1.0000
    9   1.1250
```

Problem #2 - Finite difference approximation

To approximate a derivative of a function $f(x)$ *numerically*, we borrow an idea from Calculus I and approximate a derivative using a *secant* approach.

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} \tag{2}$$

for a small value of h.

In this problem, you will approximate the derivative using this centered *finite difference formula*.

To Do

- Create an array x of equally spaced points $x_i = a + hi$ for $i = -1, 0.1, \ldots, N, N+1$ on the interval $[-1.1]$.

- Create an array y containing values $y_i = f(x_i)$ for

$$f(x) = \cos(x)e^{(-10x^2)}, \quad x \in [-1, 1] \tag{3}$$

- Create an array g containing approximations to $f'(x_i)$, using a *centered difference formula*.

- Write out a table of values for $N = 8$ for values i, x_i, $f(x_i)$, your approximation $g_i = f'(x_i)$ and error $|f'(x_i) - g_i|$.

Your table should $N + 1$ rows corresponding to equally spaced points in $[-1, 1]$.

The results of your table should look like :

```
 i           x         f(x)        f'(x)       err(x)
----------------------------------------------------
 0  -1.00000000   0.00052880   0.00527766   4.7489e-03
 1  -0.75000000   0.04204160   0.14402367   1.0198e-01
 2  -0.50000000   0.75971728   1.03196513   2.7225e-01
 3  -0.25000000   2.72553303   1.85592727   8.6961e-01
 4   0.00000000  -0.00000000   0.00000000   0.0000e+00
 5   0.25000000  -2.72553303  -1.85592727   8.6961e-01
 6   0.50000000  -0.75971728  -1.03196513   2.7225e-01
 7   0.75000000  -0.04204160  -0.14402367   1.0198e-01
 8   1.00000000  -0.00052880  -0.00527766   4.7489e-03
```

Tips

- Use your functions `allocate_1d` and `free_1d` to allocate and de-allocate arrays for x, $f(x)$ and $g(x)$.

- Use the Sympy code below to compute the true derivative $f'(x)$. You will need this to compute your error.

In [4]:
```
%reset -f
```

Sympy

Sympy is a *symbolic* module available in Python. Using Sympy, we can compute the derivative of $f(x)$ analytially. In your code, you can use this analytic form to write a function that computes the true derivative.

In [5]:
```python
import sympy as sp

x_sp = sp.symbols('x')

f_sp = sp.cos(x_sp)*sp.exp(-10*x_sp**2)
print("f(x) = ")
display(f_sp)

dfdx_sp = f_sp.diff().simplify()
print("f'(x) = ")
display(dfdx_sp)
```

```
f(x) =
```

$$e^{-10x^2}\cos(x)$$

```
f'(x) =
```

$$-(20x\cos(x)+\sin(x))\,e^{-10x^2}$$

In [6]:
```c
%%file prob2.c

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

#include <math.h>

double f(double x)
{
    return cos(x)*exp(-10*x*x);
}

double fp(double x)
{
    /* Put true derivative here */
    return -exp(-10*x*x)*(sin(x)+20*x*cos(x));
}



double* allocate_1d(int n, int m)
{
    /* Use malloc to allocate memory */
    double *y = (double*) malloc((n+2*m)*sizeof(double));
    return y;
}


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

}
```

```c
int main(int argv, char**argc)
{
    int N = 8;
    int m = 1;
    double a = -1;
    double b = 1;
    double g;

    double *x = allocate_1d(N+1,m);
    double h = (b-a)/N;

    for(int i = -m; i < N+m+1; i++)
    {
        x[i] = a + i*h;
    }
    printf("%6s %8s %12s %12s %12s\n","i","x","f'(x)","g(x)","err(x)");
    printf("------------------------------------------------------------------\n'
    for(int i = 0; i <= N+m+1; i++)
    {
        g = (f(x[i+1])-f(x[i-1]))/(2*h);
        if (i >= 0 && i < N+m)
            printf("%5d %12.8f %12.8f %12.8f %12.4e\n",i,x[i],fp(x[i]),g,fabs(g

    }

    free_1d(&x,0);
    return 0;
}
```

```
Overwriting prob2.c
```

In [7]:

```bash
%%bash

rm -rf prob2

gcc -o prob2 prob2.c

./prob2
```

```
     i        x        f'(x)         g(x)       err(x)
-----------------------------------------------------------------
    0  -1.00000000   0.00052880   0.00527766   4.7489e-03
    1  -0.75000000   0.04204160   0.14402367   1.0198e-01
    2  -0.50000000   0.75971728   1.03196513   2.7225e-01
    3  -0.25000000   2.72553303   1.85592727   8.6961e-01
    4   0.00000000  -0.00000000   0.00000000   0.0000e+00
    5   0.25000000  -2.72553303  -1.85592727   8.6961e-01
    6   0.50000000  -0.75971728  -1.03196513   2.7225e-01
    7   0.75000000  -0.04204160  -0.14402367   1.0198e-01
    8   1.00000000  -0.00052880  -0.00527766   4.7489e-03
```

Problem #3 - Evaluate and plot a sinc function

In this problem, you will compute the values of a function on a two dimensional grid, write the solution to a file, load the solution back into Python, and plot the solution.

To Do

- Write a C program to compute the values of the two dimensional shifted *sinc* function

$$f(x,y) = \frac{\sin(r)}{r}, \qquad r = \sqrt{(x-5)^2 + (y+5)^2} \tag{4}$$

on the domain $(x,y) \in [-20, 20] \times [-20, 20]$.

- Write the values to a file using `fwrite`, along with any metadata,
- Load the data in Python
- Plot your solution.

See the practice notebook from Week #6 (Wednesday) for hints on this problem.

Tips

- To allocate memory for the function values, use a "statically defined array".

  ```
  double F[N+1][N+1];
  ```

  We will discuss more flexible array indexing in 2d later.

- The sinc function is a continuous function, even at 0. Using L'Hopitals rule, we can easily show that

$$\lim_{x \to 0} \frac{\sin(x)}{x} = 1 \tag{5}$$

From this, we can define $\text{sinc}(0) = 1$. In your code, you can do something like

```
if (x == 0)
    return 1;
else
    return sin(x)/x;
```

- Be sure that your image shows the peak of the *sinc* function located at grid point $(5, -5)$.

- Use `imshow` to plot the 2d solution.

In [8]:
```
%reset -f
```

In [9]:
```
%%file prob3.c

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

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

}

double *linspace(int a,int b,int N)
{
    double *y = (double*) malloc((N+1)*sizeof(double));
    double h = (double)(b-a)/N;

    for(int i=0;i<=N;i++)
    {
       y[i]=a+(double)i*h;
    }
    return y;
}


int main(int argv, char** argc)
{
    int N=100;

    /* Your code goes here*/
    double *y = linspace(-20,20,N);
    double *x = linspace(-20,20,N);
    double r;
    double F[N+1][N+1];

    for(int i=0;i<=N;i++)
    {
        for(int j=0;j<=N;j++)
        {
            r=sqrt((x[i]-5)*(x[i]-5) + (y[j]+5)*(y[j]+5));
            if (r ==0)
                F[j][i]= 1;
            else
                F[j][i]= sin(r)/r;
        }
    }

    FILE *file = fopen("prob3_output.dat","w");
    fwrite(&N,sizeof(int),1, file);
    fwrite(&x[0],sizeof(double),N+1, file);
    fwrite(&y[0],sizeof(double),N+1, file);
    fwrite(&F, sizeof(double), (N+1)*(N+1), file);
    fclose(file);
```

```
    free_1d(&y,0);
    free_1d(&x,0);
    return 0;
}
```

```
Overwriting prob3.c
```

In [10]:

```
%%bash

rm -rf prob3

gcc -o prob3 prob3.c

./prob3
```

In [11]:

```
import os

stats = os.stat('prob3_output.dat')
print(f"File size : {stats.st_size:d} bytes")

N = 100
print("Expected file size : ",8*(N+1)**2+ 2*(N+1)*8 +4)
```

```
File size : 83228 bytes
Expected file size :  83228
```

In [12]:

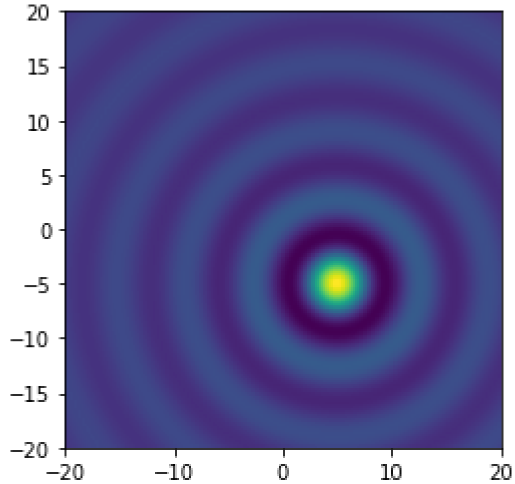
```
from matplotlib.pyplot import *
from numpy import *
```

In [13]:

```
dt = dtype([('N',np.int32), \
            ('x',(np.float64,N+1)),\
            ('y',(np.float64,N+1)),\
            ('F',(np.float64,(N+1)**2))])

# read data from file
fout = open('prob3_output.dat','rb')
N,x,y,F = fromfile(fout,dtype=dt,count=1)[0]
fout.close()
```

In [14]:

```
clf()
figure(1)
fxy=F.reshape((N+1,N+1))
e = [x[0],x[N],y[0],y[N]]
imshow(fxy,aspect=True,extent=e,origin="lower");
```



Problem #4

Why might the following code produce unreliable results?

Tip

- Think about "scope".

In [15]:
```
%reset -f
```

In [16]:
```
%%file prob4.c

#include <stdio.h>

int main(int argc, char** argv)
{
    double *y = NULL;
    {
        double x = 11.5;
        y = &x;
    }
    printf("y[0] = %f\n",y[0]);
}
```

```
Overwriting prob4.c
```

In [17]:
```
%%bash

rm -rf prob4.o

gcc -o prob4 prob4.c

./prob4
```

```
y[0] = 11.500000
```

COMMENT

The code is attempting to access previously deallocated memory, which could lead to unreliable results. The location of a local variable x allocated on the stack inside the block separated by curly braces is given to the pointer y. The lifetime of x expires after that block does, and attempting to access it via y causes ambiguous behavior. As a result, trying to print the value of y[0] after x has left the scope may cause the program to output an unexpected value. To prevent this from occuring, pointers must refer only to valid memory locations by employing dynamic memory allocation or keeping the variable pointed to in scope.

Problem #5 - Fix a memory issue

The following code should print out two different values. It appears to work. But is it reliable?

To Do

- What is the code below doing? Why might we expect unpredictable behavior?
- Describe two fixes that would guarantee predictable behavior.
- Implement your fixes.

In [18]:
```
%reset -f
```

In [19]:
```
%%file prob5.c

#include <stdio.h>

void change_value(double **y)
{
    double x = 11.5;
    *y = &x;
}

int main(int argc, char** argv)
{
    double x = 2.3;
    double *y = &x;

    printf("Old value : y[0] = %f\n",y[0]);
    change_value(&y);
    printf("New value : y[0] = %f\n",y[0]);
}
```

```
Overwriting prob5.c
```

In [20]:
```
%%bash

rm -rf prob5.o

gcc -o prob5 prob5.c

./prob5
```

```
Old value : y[0] = 2.300000
New value : y[0] = 11.500000
```

COMMENT

The code tries to modify a pointer's value to a double passed by reference to the function change_value(). The pointer y *is given the address of a local variable x in the function, designating that* y points to memory that is deallocated when the function completes (i.e *y points to invalid memory when x goes out of scope). The pointer y points to invalid memory that could be overwritten at any time when main() attempts to dereference it, leading to unpredictable behavior.

1. Use malloc() to dynamically create memory for the double pointed to by y *to ensure that it remains available after the change value() function has finished. After that, provide* y access to the memory's assigned address.
2. Pass the variable by reference: In this instance, the change value() function can accept the address of the variable x from the main function. The value of x in the main function will be updated when we alter the value of *y in the change value() function in this fashion.

In [21]:
```
%%file prob5_fix1.c
#include <stdio.h>      // FIRST FIX //
#include <stdlib.h>

void change_value(double **y)
{
    double *x =(double*) malloc(sizeof(double));
     *x = 11.5;
    *y = x;
}

int main(int argc, char** argv)
{
    double x = 2.3;
    double *y = &x;

    printf("Old value : y[0] = %f\n",y[0]);
    change_value(&y);
    printf("New value : y[0] = %f\n",y[0]);

    free(y);
}
```

```
Overwriting prob5_fix1.c
```

In [22]:
```
%%bash

rm -rf prob5_fix1.o

gcc -o prob5_fix1 prob5_fix1.c

./prob5_fix1
```

```
Old value : y[0] = 2.300000
New value : y[0] = 11.500000
```

In [23]:
```
%%file prob5_fix2.c

#include <stdio.h>  // second fix //

void change_value(double **y)
{

    **y = 11.5;
}

int main(int argc, char** argv)
{
    double x = 2.3;
    double *y = &x;

    printf("Old value : y[0] = %f\n",y[0]);
    change_value(&y);
    printf("New value : y[0] = %f\n",y[0]);
}
```

```
Overwriting prob5_fix2.c
```

In [24]:
```
%%bash

rm -rf prob5_fix2.o

gcc -o prob5_fix2 prob5_fix1.c

./prob5_fix2
```

```
Old value : y[0] = 2.300000
New value : y[0] = 11.500000
```

Problem #6 - Computing a maximum using MPI

In the following problem, each processor will generate an random number. Your task is to compute the largest of these numbers.

To Do

- Complete the code below so that rank 0 receives a value from each of the other processes and computes a maximum of all values (including a value generated on rank

0).

- All ranks other than 0 should send their value to rank 0.
- Each rank should print out its random value.
- Report the maximum value you found.

Run your code on 16 processes, so that 16 random numbers are generated.

Question

We have only discussed in class *synchronous* send and receive calls. This means that a send and receive calls are *blocking* (only complete when the message has been sent *and* recieved). If you could send and receive *asynchronously* (.e.g. using non-blocking calls), can you think of an efficient way to organize the communication between processors?

COMMENT

Synchronous calls are less effective in facilitating communication between processors than non-blocking calls. One technique to structure communication using non-blocking calls is to overlap processing and communication. This entails beginning communication processes while carrying on with computation. Managing the overlap between communication and computation is essential to prevent overloading either process.

In [25]:
```
%reset -f
```

In [26]:
```
%%file prob6.c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <mpi.h>

double random_number()
{
  return (double) rand() / (double) RAND_MAX ;
}

void random_seed()
{
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    srand(clock() + rank);
}

int main(int argc, char** argv)
{
    int rank, nprocs;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
```

```
    random_seed();

    double x = random_number();
    printf("Rank %3d : Random number is %12.8f\n",rank,x);

    if (rank == 0)
    {
          double xmax = x;

        for (int i = 1; i < nprocs; i++)
        {
            double temp;
            MPI_Recv(&temp, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGN
            xmax = fmax(xmax, temp);
        }
        printf("Maximum value is %12.8f\n",xmax);
    }
    else
    {
        MPI_Send(&x, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    }

    MPI_Finalize();
}
```

```
Overwriting prob6.c
```

In [27]:

```
%%bash

rm -rf prob6

mpicc -o prob6 prob6.c

mpirun -n 16 ./prob6
```

```
Rank   0 : Random number is   0.29399624
Rank   9 : Random number is   0.23802205
Rank   3 : Random number is   0.23733333
Rank   5 : Random number is   0.20914275
Rank  11 : Random number is   0.17947298
Rank   1 : Random number is   0.25545137
Rank   4 : Random number is   0.31187167
Rank  12 : Random number is   0.25483309
Rank   2 : Random number is   0.30791153
Rank   6 : Random number is   0.23867946
Rank   8 : Random number is   0.30821675
Rank  10 : Random number is   0.31493178
Rank  14 : Random number is   0.19172125
Rank  13 : Random number is   0.24135608
Rank   7 : Random number is   0.31703707
Rank  15 : Random number is   0.28457329
Maximum value is   0.31703707
```

Problem #7 - Plot a circle! (MPI)

In the following, you will evaluate the parametric equations for a circle, write the values to a file, load the values in Python and then plot the values to create a circle. The work of evaluating the parametric equations will be distributed to several MPI ranks.

The parametric equations $(C(\theta), S(\theta))$ for the circle are given by

$$C(\theta) = \cos(\theta) \tag{6}$$
$$S(\theta) = \sin(\theta) \tag{7}$$

for $\theta \in [0, 2\pi]$. To plot a circle, we can evaluate these equations at N equally spaced values θ_i, $i = 0, 1, 2, \ldots N$ in the interval $[0, 2\pi]$ to get points (C_i, S_i). We then plot these points to get a circle.

To Do

- Write a function that allocates memory for arrays $\mathbf{C}$ and $\mathbf{S}$, and evaluates the parametric equations at a range of values sub-intervals $[a, b] \subset [0, 2\pi]$. The signature for your function might look something like :

```
    void eval_xy(double ap, double bp, int n, double**C, double **S)
```

The function should allocate memory for arrays $\mathbf{C}$ and $\mathbf{S}$ and compute C_i and S_i for $i = 0, 1, 2, \ldots, n$.

- The work will be distributed among a total of P processes by dividing the domain $[a, b]$ into equal-sized subintervals $[a_p, b_p]$, $p = 0, 1, \ldots P - 1$.
- Each rank p should evaluate the parametric equations at N/P equally spaced values in the sub-interval $[a_p, b_p] \subset [0, 2\pi]$.
- Each rank p, $p > 0$, should send their results to rank 0.
- Rank 0 should collect all results for sub-intervals and store them in an arrays of length $N + 1$.
- Rank 0 should write out N, the interval $[a, b] = [0, 2\pi]$, and arrays θ, $\mathbf{C}$ and $\mathbf{S}$.
- Load the meta data and arrays in python and plot your circle.

Compute the above for $N = 32$. Run your code on 4 processors.

Tips

- Use your function `allocate_1d` (with $m = 0$) to allocate memory for local arrays in `eval_xy`.
- Rank 0 should also do some work and compute values on the sub-interval $[a_0, b_0]$.

- Use the MPI `tag` to make sure that the two arrays `C` and `S` get correctly received by rank 0.
- Create a Numpy `dtype` to read data back into Python.
- Set the aspect ratio of your plot to `equal`.

  `gca().set_aspect('equal')`

In [28]:
```
%reset -f
```

In [29]:
```
%%file prob7.c

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>

double* allocate_1d(int n, int m)
{
    /* Use malloc to allocate memory */
    double *y = (double*) malloc((n+2*m)*sizeof(double));
    return y;
}


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

}

void eval_xy(double ap, double bp, int n, double **C, double **S)
{
    double xi, h;

    /* TODO : Allocate memory for S and C */

     *C = allocate_1d(n+1,0);
     *S = allocate_1d(n+1,0);
     h = (bp-ap)/n;

    for(int i=0;i<=n;i++)
    {
       xi =(double) ap+i*h;
       *(*C+i)=cos(xi);
       *(*S+i)=sin(xi);
    }

}

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);

    int N = 32;

    double a = 0, b = 2*M_PI;

    int N_local = N/nprocs;

    /* TODO : Use "rank" to determine range of values [a_local,b_local] for thi

    double h = (b - a) / N;
    double a_local = a + rank * N_local * h;
    double b_local = a_local + N_local * h;


    double *C_local, *S_local;

    /* TODO : Compute range of values for this processor on [a_local, b_local]

    eval_xy(a_local,b_local,N_local,&C_local,&S_local);

    if (rank == 0)
    {
        /* Okay to define "automatic arrays" here, since they won't be referenc
        double C[N+1], S[N+1];

        /* Copy data from rank 0 into C and S*/
        for (int i = 0; i <= N_local; i++)
        {
            C[i] = C_local[i];
            S[i] = S_local[i];
        }


        /* TODO : Receive data from each of nprocs-1 processes into correct rar
        for (int i = 1; i < nprocs; i++)
        {
         //TODO : Receive data from process i into correct range in C, S //

            MPI_Recv(&C[i * N_local], N_local+1, MPI_DOUBLE, i, 0, MPI_COMM_WOR
            MPI_Recv(&S[i * N_local], N_local+1, MPI_DOUBLE, i, 1, MPI_COMM_WOR

        }

        FILE *fout = fopen("prob7_output.dat","w");

        /* TODO : Write out meta-data and arrays */

        fwrite(&N,sizeof(int),1,fout);
        fwrite(&a,sizeof(double),1,fout);
        fwrite(&b,sizeof(double),1,fout);
        fwrite(C,sizeof(double),N+1,fout);
        fwrite(S,sizeof(double),N+1,fout);
        fclose(fout);


        /* ... */
```

```
    }
    else
    {

        /* TODO : Send data for C and S to rank 0 */
            MPI_Send(C_local, N_local+1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
            MPI_Send(S_local, N_local+1, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD);
    }

    free_1d(&C_local);
    free_1d(&S_local);

    MPI_Finalize();
    return 0;
}
```

```
Overwriting prob7.c
```

In [30]:

```
%%bash

rm -rf prob7

mpicc -o prob7 prob7.c

mpirun -n 4 ./prob7
```

In [ ]:

```

```

Open file in Python

In [31]:

```
from numpy import *
```

In [32]:

```
fout=open("prob7_output.dat","rb")

# Read value N
N = fromfile(fout,dtype=int32,count=1)[0]

fout.close()
print(f"N is {N:d}")

# TODO : Build a  Numpy `dtype`.  Use "N" read in above.
dt = dtype([('N',int32), \
            ('a',float64),\
            ('b',float64),\
            ('C',(float64,N+1)),\
           ('S',(float64,N+1))])


# TODO : Read data (include 'N' again).

# read data from file
fout = open('prob7_output.dat','rb')
N,a,b,C,S = fromfile(fout,dtype=dt,count=1)[0]
fout.close()
```

```
N is 32
```

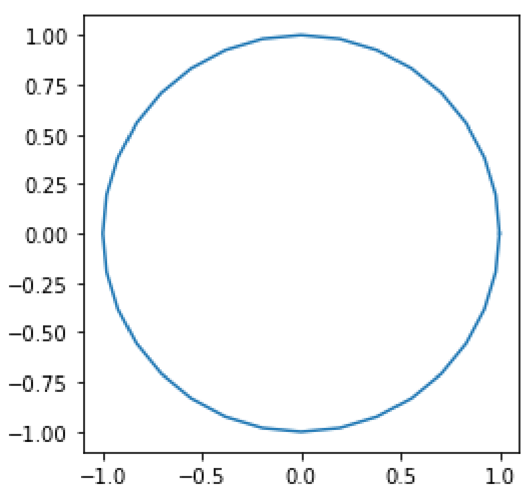
Check file size

In [33]:
```
import os

stats = os.stat("prob7_output.dat")
actual_size = stats.st_size
print(f"File size          : {actual_size:d} bytes")

fout = open("prob7_output.dat","rb")
N = fromfile(fout,dtype=int32, count=1)[0]
fout.close()

expected_size = 2*(N+1)*8 + 2*8 + 4
print(f"Expected file size : {expected_size:d} bytes")

print("")
if expected_size == actual_size:
    print("Actual size and expected size match!")
else:
    print("Something went wrong : File sizes differ")
```

```
File size          : 548 bytes
Expected file size : 548 bytes

Actual size and expected size match!
```

Plot circle

In [34]:
```
from matplotlib.pyplot import *

figure(1)
clf()

# TODO : Plot a circle!
plot(C,S)
gca().set_aspect('equal');
```



Problem #8 - Fresnel functions (571, MPI)

Repeat problem #7, but instead of using $C(\theta) = \cos(\theta)$ and $S(\theta) = \sin(\theta)$, set C and S to the *Fresnel* functions

$$C(t) = \frac{\pi}{2}\frac{1-i}{4}\left[\mathrm{erf}\left(\frac{1+i}{\sqrt{2}}t\right) + i\mathrm{erf}\left(\frac{1-i}{\sqrt{2}}t\right)\right] \tag{8}$$

and

$$S(t) = \frac{\pi}{2}\frac{1+i}{4}\left[\mathrm{erf}\left(\frac{1+i}{\sqrt{2}}t\right) - i\mathrm{erf}\left(\frac{1-i}{\sqrt{2}}t\right)\right] \tag{9}$$

where $i = \sqrt{-1]}$.

To do

- Evaluate the Fresnel functions $(C(t), S(t))$.
- Plot the parametric equations over the interval $t \in [-20, 20]$.

Compute the Fresnel equations for $N = 1024$. Run your results on four processors.

This problem is essentially problem #7, with a different set of parametric equations

Tips

- For this, you will need to include the header `complex.h`. The complex unit is `I`.
- You will also need the complex valued error function `cerf`.

  You can install this in Linux as

  ```
  sudo apt-get install libcerf-dev
  ```

  And on a Mac (using MacPorts, for example), you can use

  ```
  sudo port install libcerf
  ```

  Read more about this library here.

- To compile the code, you may need to explicitly include an include path and a library path at the compile line.

  ```
  mpicc -o prob8 prob8.c -I<include-dir> -L<lib-dir>  -lcerf
  ```

where `<include-dir>` is the include directory containing `cerf.h` and `<lib-dir>` is the directory containing the library flie `libcerf.a`.

Example

If you use the MacPorts installation on OSX, the compile line will look something like.

```
mpicc -o prob8 prob8.c -I/opt/local/include -L/opt/local/lib
-lcerf
```

Test cerf installation

Run the code below to test your installation of the library file `libcerf`.

In [35]:
```
%%file test_cerf.c

#include <stdio.h>
#include <complex.h>
#include <math.h>
#include <cerf.h>

int main(int argc, char** argv)
{
    complex double z0 = I*I;  /* should be -1 */
    printf("I*I = %20.16f + %20.16f\n",creal(z0),cimag(z0));

    /* Complex valued error function cerf */
    complex double z1 = cerf((1 + I)/sqrt(2.0));
    printf("z   = %20.16f + %20.16f\n",creal(z1),cimag(z1));

    return 0;
}
```

```
Overwriting test_cerf.c
```

In [36]:
```
%%bash

rm -rf test_cerf

gcc -o test_cerf test_cerf.c -I/opt/local/include -L/opt/local/lib -lcerf

./test_cerf
```

```
I*I =  -1.0000000000000000 +   0.0000000000000000
z   =   0.9692642119442157 +   0.4741476366409944
```

You should get :

```
I*I =  -1.0000000000000000 +   0.0000000000000000
z   =   0.9692642119442157 +   0.4741476366409944
```

You can also check your results against the result from the complex-valued Scipy special function `erf`.

In [37]:
```
import scipy
from  scipy.special import erf
from numpy import *

z = erf((1 + 1j)/sqrt(2.))
print(f"z = {z.real:20.16f} + {z.imag:20.16f}")
```

```
z =    0.9692642119442157 +    0.4741476366409944
```

Code for problem #8

In [38]:
```
%reset -f
```

In [39]:
```
%%file prob8.c

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>
#include <complex.h>
#include <cerf.h>

double* allocate_1d(int n, int m)
{
    /* Use malloc to allocate memory */
    double *y = (double*) malloc((n+2*m)*sizeof(double));
    return y;
}


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

}

void eval_xy(double ap, double bp, int n, double **C, double **S)
{
    double xi, h;
    /* TODO : Allocate memory for S and C */

     *C = allocate_1d(n+1,0);
     *S = allocate_1d(n+1,0);
     h = (bp-ap)/n;

    for(int i=0;i<=n;i++)
    {
       xi =(double) ap+i*h;
       *(*C+i)=(M_PI*(1-I))/8*(cerf(((1+I)/sqrt(2))*xi)+I*cerf(((1-I)/sqrt(2))*
       *(*S+i)=(M_PI*(1+I))/8*(cerf(((1+I)/sqrt(2))*xi)-I*cerf(((1-I)/sqrt(2))*
    }

}

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);
```

```
    int N = 1024;

    double a = -20.0, b = 20.0;

    int N_local = N/nprocs;

    /* TODO : Use "rank" to determine range of values [a_local,b_local] for thi

    double h = (b - a) / N;
    double a_local = a + rank * N_local * h;
    double b_local = a_local + N_local * h;


    double *C_local, *S_local;

    /* TODO : Compute range of values for this processor on [a_local, b_local]

    eval_xy(a_local,b_local,N_local,&C_local,&S_local);

    if (rank == 0)
    {
        /* Okay to define "automatic arrays" here, since they won't be referenc
        double C[N+1], S[N+1];

        /* Copy data from rank 0 into C and S*/
        for (int i = 0; i <= N_local; i++)
        {
            C[i] = C_local[i];
            S[i] = S_local[i];
        }


        /* TODO : Receive data from each of nprocs-1 processes into correct rar
        for (int i = 1; i < nprocs; i++)
        {
         //TODO : Receive data from process i into correct range in C, S //

            MPI_Recv(&C[i * N_local], N_local+1, MPI_DOUBLE, i, 0, MPI_COMM_WOR
            MPI_Recv(&S[i * N_local], N_local+1, MPI_DOUBLE, i, 1, MPI_COMM_WOR

        }

        FILE *fout = fopen("prob8_output.dat","w");

        /* TODO : Write out meta-data and arrays */

        fwrite(&N,sizeof(int),1,fout);
        fwrite(&a,sizeof(double),1,fout);
        fwrite(&b,sizeof(double),1,fout);
        fwrite(C,sizeof(double),N+1,fout);
        fwrite(S,sizeof(double),N+1,fout);
        fclose(fout);


        /* ... */

    }
    else
    {
```

```
        /* TODO : Send data for C and S to rank 0 */
            MPI_Send(C_local, N_local+1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
            MPI_Send(S_local, N_local+1, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD);
    }

    free_1d(&C_local);
    free_1d(&S_local);

    MPI_Finalize();
    return 0;
}
```

```
Overwriting prob8.c
```

In [40]:
```
%%bash

rm -rf prob8

mpicc -o prob8 prob8.c -I/opt/local/include -L/opt/local/lib -lcerf

mpirun -n 4 ./prob8
```

Open file in Python

In [41]:
```
from numpy import *
```

In [42]:
```
fout=open("prob8_output.dat","rb")

# Read value N
N = fromfile(fout,dtype=int32,count=1)[0]

fout.close()
print(f"N is {N:d}")

# TODO : Build a  Numpy `dtype`.  Use "N" read in above.
dt = dtype([('N',int32), \
            ('a',float64),\
            ('b',float64),\
            ('C',(float64,N+1)),\
           ('S',(float64,N+1))])


# TODO : Read data (include 'N' again).

# read data from file
fout = open('prob8_output.dat','rb')
N,a,b,C,S = fromfile(fout,dtype=dt,count=1)[0]
fout.close()
```

```
N is 1024
```

Check file size

In [43]:
```
# Check file size
import os

stats = os.stat("prob8_output.dat")
actual_size = stats.st_size
```

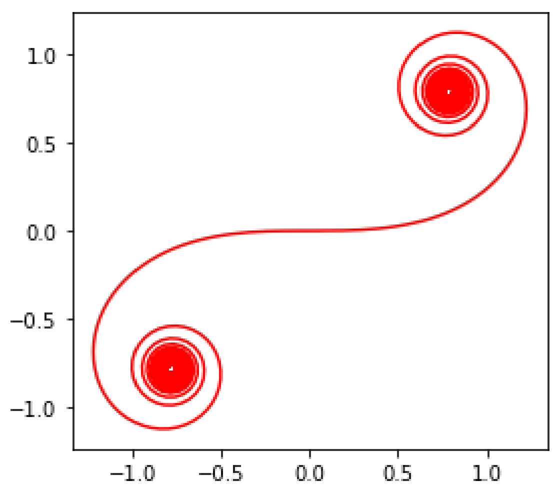
```
print(f"File size           : {actual_size:d} bytes")

fout = open("prob8_output.dat","rb")
N = fromfile(fout,dtype=int32, count=1)[0]
fout.close()

expected_size = 2*(N+1)*8 + 2*8 + 4
print(f"Expected file size : {expected_size:d} bytes")

print("")
if expected_size == actual_size:
    print("Actual size and expected size match!")
else:
    print("Something went wrong : File sizes differ")
```

```
File size           : 16420 bytes
Expected file size : 16420 bytes

Actual size and expected size match!
```

Plot the Euler Spiral!

In [44]:
```
from matplotlib.pyplot import *
```

In [45]:
```
figure(2)
clf()

# TODO : Plot a spiral
plot(C,S,"r")
gca().set_aspect('equal');
```



In [ ]:
```
```