DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Nº1

12 de Octubre de 2010 Sistemas Complejos en Máquinas Paralelas

Integrante	LU	Correo electrónico
Allekotte, Kevin	490/08	kevinalle@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Discretización del Sistema

Queremos resolver por el método de *Diferencias Finitas* la ecuación transitoria de calor en 1D usando el método alfa para la discretización temporal.

La ecuacion es:

$$\frac{\partial u}{\partial t} = K\frac{\partial^2 u}{\partial x^2}$$

1.1. Adimensionalización del Sistema

Planteamos la ecuación para x' y t' con:

$$x' = \frac{x}{x_0} \qquad t' = \frac{t}{t_0}$$

Y obtenemos como resultado

$$\frac{\partial u}{\partial(t't_0)} = K\frac{\partial^2 u}{\partial(x'x_0)^2}$$

Entonces el sistema adimensionalizado es

$$\frac{\partial u}{\partial t'} = \left(\frac{t_0 K}{x_0^2}\right)\frac{\partial^2 u}{\partial x'^2} = K'\frac{\partial^2 u}{\partial x'^2}$$

1.2. Discretización

$$u(x,t) \longrightarrow u(x_i,t_n) = \mathbf{u}_{i,n}$$

1.2.1. Condiciones Iniciales y de Borde

$\mathbf{u}_i^0 = F(i)$ $\quad$ $\mathbf{u}_0^n =$ Condicion de borde Izquierdo $\quad$ $\mathbf{u}_1^n =$ Condicion de borde Derecho

1.2.2. $\alpha = 0$

$$\frac{\mathbf{u}_i^{n+1} - \mathbf{u}_i^n}{k} = K'\frac{\mathbf{u}_{i+1}^n - 2\mathbf{u}_i^n + \mathbf{u}_{i-1}^n}{h^2}$$

$$\mathbf{u}_i^{n+1} = r\mathbf{u}_{i+1}^n + (1-2r)\mathbf{u}_i^n + r\mathbf{u}_{i-1}^n \qquad r = \frac{kK'}{h^2}$$

1.2.3. $\alpha = 1$

$$\frac{\mathbf{u}_i^{n+1} - \mathbf{u}_i^n}{k} = K'\frac{\mathbf{u}_{i+1}^{n+1} - 2\mathbf{u}_i^{n+1} + \mathbf{u}_{i-1}^{n+1}}{h^2}$$

$$\mathbf{u}_i^{n+1} = \frac{r}{1+2r}(\mathbf{u}_{i+1}^{n+1} + \mathbf{u}_{i-1}^{n+1}) + \frac{\mathbf{u}_i^n}{1+2r}$$

2. Resolución Numérica Serial

2.1. $\alpha = 0$

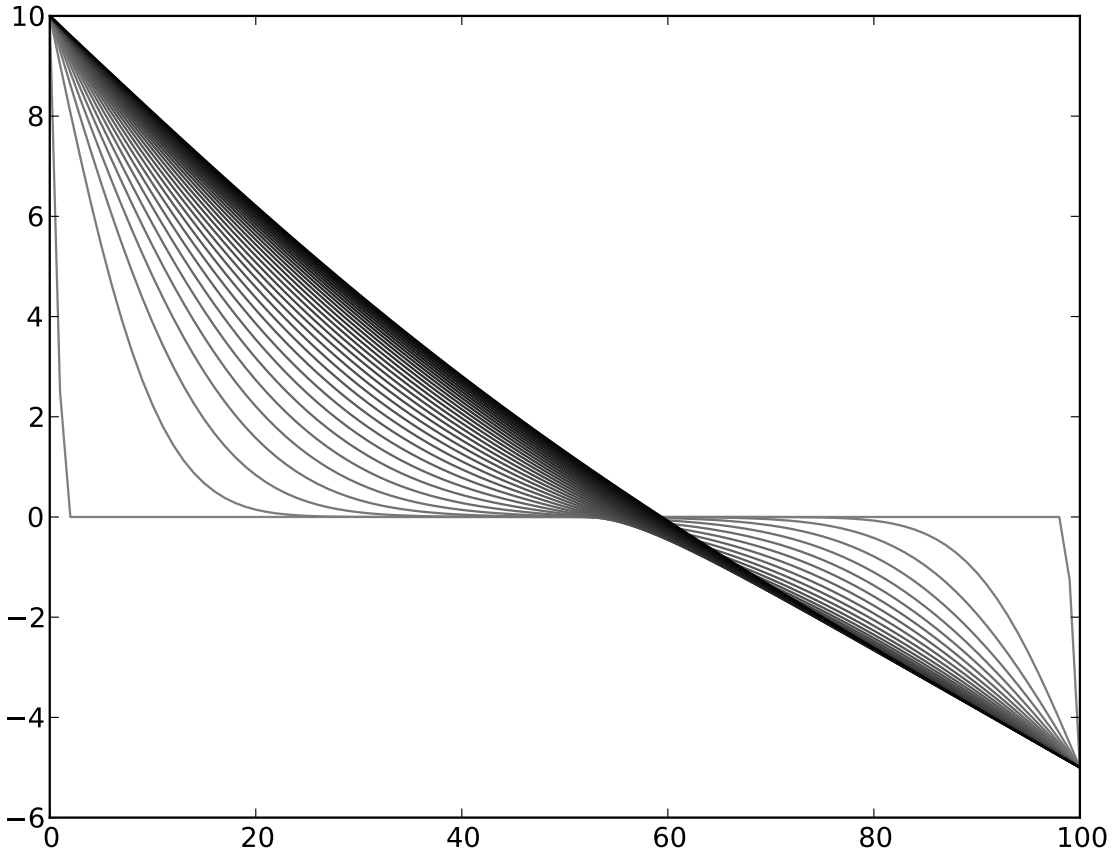
Para resolver el sistema con $\alpha = 0$ se puede hacer un vector y para cada intervalo de tiempo se cambia el valor de $\mathbf{u}_i$ por $r\mathbf{u}_{i+1} + (1-2r)\mathbf{u}_i + r\mathbf{u}_{i-1}$.

2.1.1. Código

```
#include<iostream>
#include<cstdlib>
#include<math.h>
using namespace std;

#define forn(i,n) for(int i=0;i<n;i++)
#define forsn(i,s,n) for(int i=(int)(s);i<n;i++)

//Condiciones Iniciales y de Borde
double F(double x){return 0;}
double LeftBorder(double t){return 10.;}
double RightBorder(double t){return -5.;}

int main(int argc,char**argv){
   double dt=.000025;
   double dx=.01;
   double K=1.; // Coeficiente de difusion termica ⇝
      ⇝adimensionalizado
   double L=1.; // Longitud
   double r=dt*K/(dx*dx);

   int output_t_samples=30; //cantidad de lineas de salida
   int output_x_samples=100; //resolucion del vector de salida

   double tf=.1;  //tiempo final
   int times=tf/dt+1; //cantidad de intervalos de tiempo
   int samples=L/dx+2; //cantidad de muestras espaciales
   double*u=new double[samples];
   double*u2=new double[samples];
   forn(i,samples) u[i]=F(i/samples);
   u[0]=LeftBorder(0.);
   u[samples-1]=RightBorder(0.);

   forn(t,times+1){
      //Calculamos los nuevos datos en u2 usando u
      u2[0]=LeftBorder(tf*t/times);
      u2[samples-1]=RightBorder(tf*t/times);
      forsn(i,1,samples-1){
         u2[i]=r*u[i+1]+(1-2*r)*u[i]+r*u[i-1];
      }
      //intercambiamos u con u2
      double*temp=u; u=u2; u2=temp;
```

```
        //exportar el resultado cada algunas iteraciones
        if(t%(times/output_t_samples)==0){
            clog << "t=" << t*dt << ":\t";
            for(int i=0;i<samples;i+=samples/output_x_samples) cout⇝
                ⇝ << u[i] << "␣";
            cout << endl;
        }
    }
    return 0;
}
```

2.1.2. Resultado



Las líneas más oscuras representan el estado a mayor tiempo.

El resultado es el esperado :)

2.2. $\alpha = 1$

Con $\alpha = 1$ el programa es muy parecido, pero en vez de usar el valor de la iteración anterior, se usa el de la iteración actual. Para eso en necesario usar la técnica de Gauss-Seidel que converge a la solución.

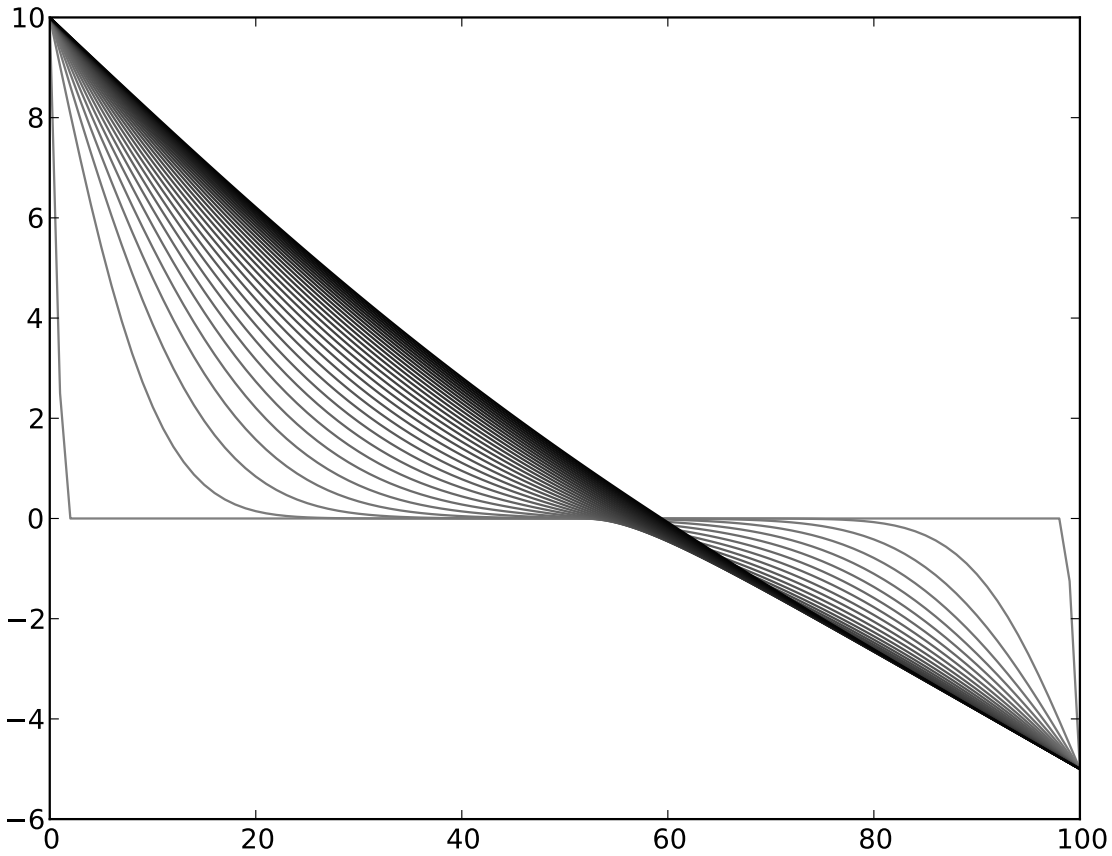
2.2.1. Código

La única parte relevante del código es la siguiente:

```
forn(iter,20){
   forsn(i,1,samples-1){
      u2[i]=(r/(1+2*r))*(u2[i+1]+u2[i-1])+u[i]/(1+2*r);
   }
}
```

(Observar que la cantidad de iteriaciones está fija y no se chequea la convergencia)

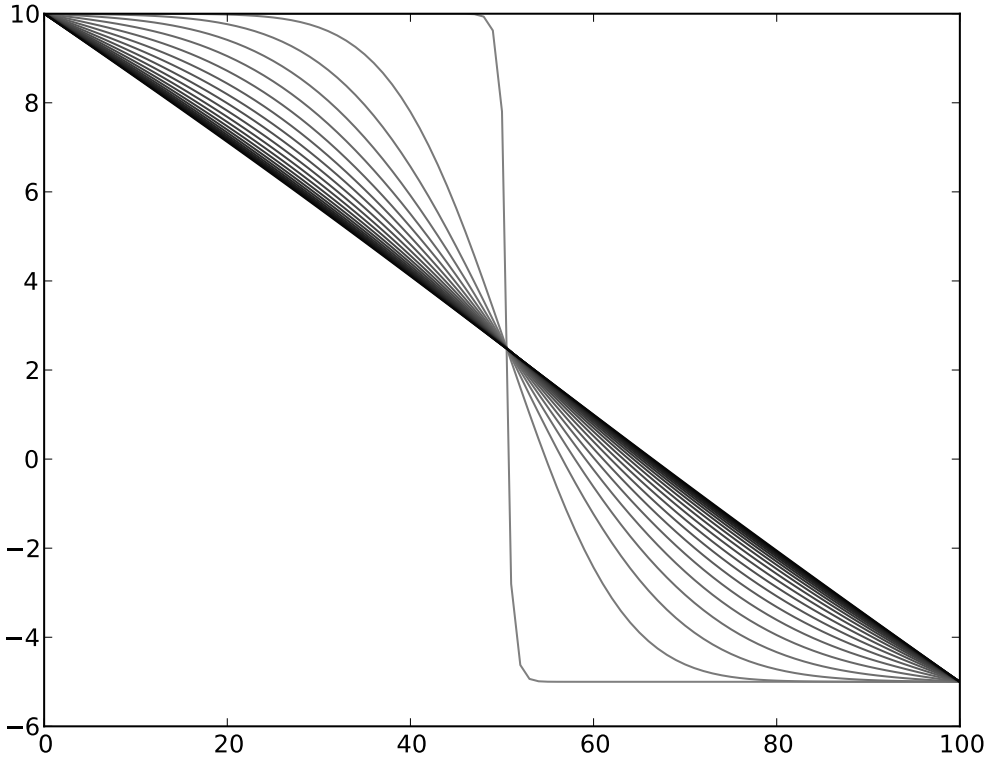
2.2.2. Resultado



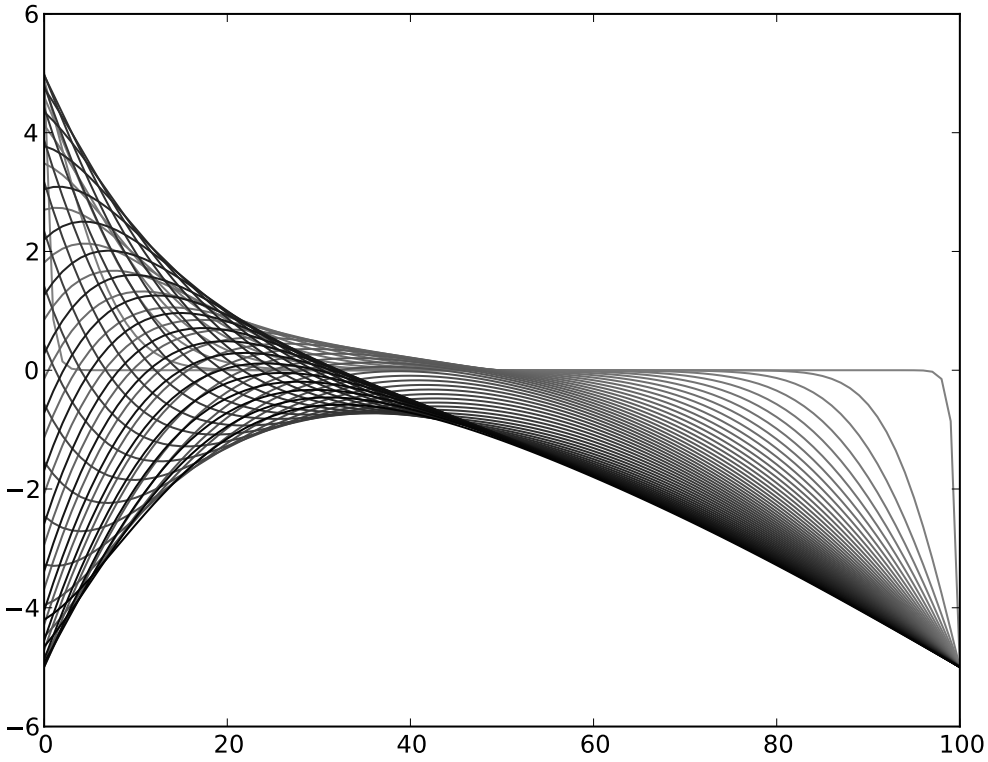
Nuevamente el resultado es el esperado.

3. Condiciones Iniciales y de Borde

$$F(x) = \begin{cases} 10 & 0 < x \leq 0{,}5 \\ -5 & 0{,}5 < x < 1 \end{cases}$$



$\mathbf{u}_0 = 5cos(100t)$



4. Implementación en paralelo: MPI

Procedemos a implementar la solución en una máquina paralela. La idea es dividir el vector en porciones aproximadamente iguales, y resolver cada pedazo en un core.

Utilizamos la librería MPI que nos brinda operaciones de ejecución en paralelo y comunicación entre los procesos. El proceso root (0) no va a hacer cálculos sino que va a coordinar a los otros procesos, recibir los resultados, y comunicarse con el usuario.

Una vez dividido el vector, cada core ejecuta una algoritmo muy parecido al anterior, pero ahora tenemos el problema de los nodos compartidos. Los procesos vecinos van a necesitar información del borde del adyacente. Esto se puede resolver con envío de mensajes sincrónicos (bloqueantes). Antes de cada iteración, los procesos envían y reciben la información necesaria de sus vecinos.

Cuando tienen resultados, éstos son enviados al root para que imprima la salida.

4.1. Código

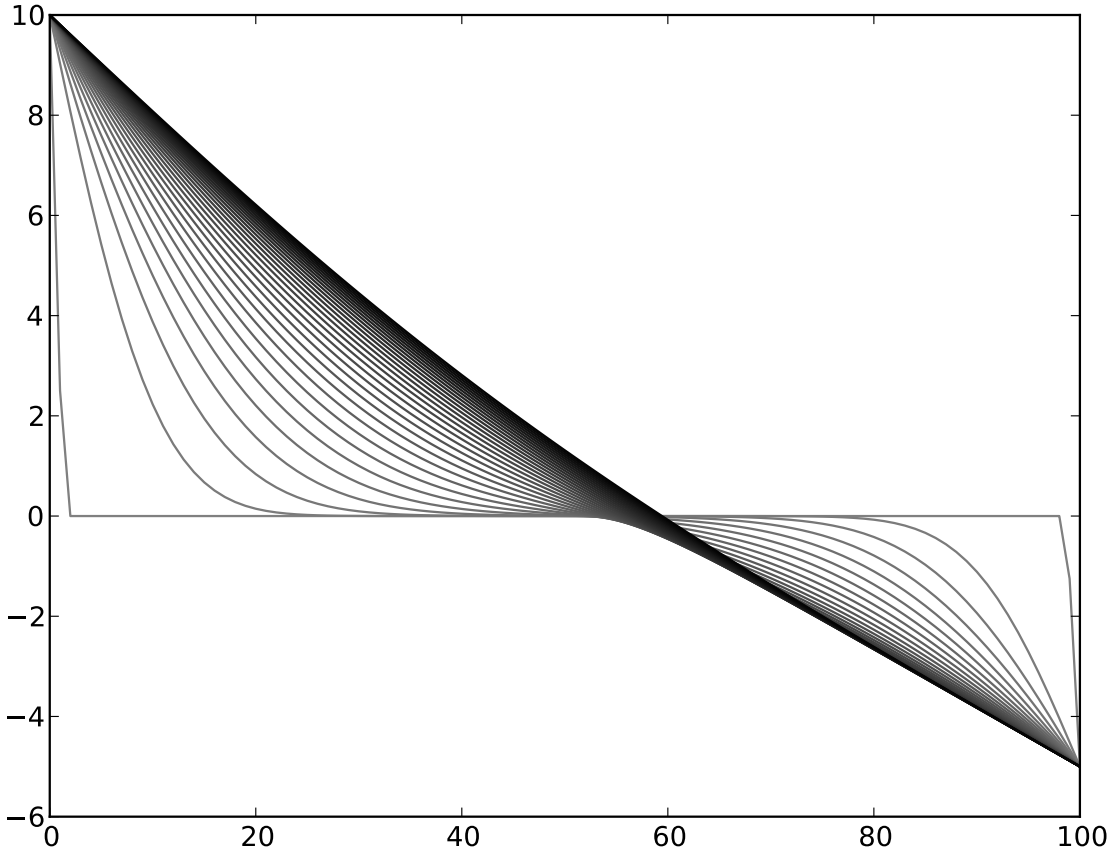
```
#include<iostream>
#include<mpi.h>
#include<cstdlib>
#include<math.h>
using namespace std;

#define forn(i,n) for(int i=0;i<n;i++)
#define forsn(i,s,n) for(int i=(int)(s);i<n;i++)

double F(double x){return x<.5?10.:-5.; /*return 0;*/}
double LeftBorder(double t){return 10.;}
double RightBorder(double t){return -5.;}

int main(int argc, char**argv){
   double dt=.000025;
   double dx=.01;
   double K=1.; // Coeficiente de difusion termica ⇝
      ⇝adimensionalizado
   double L=1.; // Longitud
   double r=dt*K/(dx*dx);

   int output_t_samples=20;
   int output_x_samples=100;

   double tf=.1;

   int times=tf/dt+1;
   int samples=L/dx+2;
```

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

if(rank!=0){
   // creo mis arrays
   int my_start=(rank-1)*samples/(np-1);
   int my_end=rank==np-1?samples:rank*samples/(np-1);
   int my_size=my_end-my_start;
   clog<<"worker␣"<<rank<<"␣computando␣desde␣"<<my_start<<"␣"
      ⇝hasta␣"<<my_end<<"␣("<<my_size<<")"<<endl;
   double*u=new double[my_size+2];
   double*u2=new double[my_size+2];
   forsn(i,my_start,my_end+2) u[i-my_start]=F((double)i/
      ⇝samples);

   u[0]=rank==1?LeftBorder(0.):F((double)(my_start-1)/samples
      ⇝);
   u[my_size+1]=rank==np-1?RightBorder(0.):F((double)(my_end
      ⇝+1)/samples);

   forn(i,my_size+2) u2[i]=u[i];

   MPI_Status s;
   if(rank!=1) MPI_Send(u+1,1,MPI_DOUBLE,rank-1,1,
      ⇝MPI_COMM_WORLD);
   if(rank!=np-1) MPI_Send(u+my_size,1,MPI_DOUBLE,rank+1,1,
      ⇝MPI_COMM_WORLD);
   forn(t,times+1){
      if(rank!=1) MPI_Recv(u2,1,MPI_DOUBLE,rank-1,1,
         ⇝MPI_COMM_WORLD,&s);
      else u2[0]=LeftBorder(tf*t/times);
      if(rank!=np-1) MPI_Recv(u2+my_size+1,1,MPI_DOUBLE,rank
         ⇝+1,1,MPI_COMM_WORLD,&s);
      else u2[my_size+1]=RightBorder(tf*t/times);
      forn(iter,20){
         forsn(i,1,my_size+1){
            //u2[i]=r*u[i+1]+(1-2*r)*u[i]+r*u[i-1];
            u2[i]=(r/(1+2*r))*(u2[i+1]+u2[i-1])+u[i]/(1+2*r);
         }
      }
      double*temp=u; u=u2; u2=temp;
```

```
            if(rank!=1) MPI_Send(u+1,1,MPI_DOUBLE,rank-1,1,⇝
               ⇝MPI_COMM_WORLD);
            if(rank!=np-1) MPI_Send(u+my_size,1,MPI_DOUBLE,rank⇝
               ⇝+1,1,MPI_COMM_WORLD);

            if(t%(times/output_t_samples)==0){
               clog<<"worker␣"<<rank<<"␣enviando␣t="<<t<<endl;
               MPI_Send(u+1,my_size,MPI_DOUBLE,0,0,MPI_COMM_WORLD);
            }
         }
      }else{
         //recibir resultados
         double*u=new double[samples];
         MPI_Status s;
         forn(t,(times+1)/(times/output_t_samples)+1){
            forsn(i,1,np) MPI_Recv(u+(i-1)*samples/(np-1), (i==np⇝
               ⇝-1?samples:i*samples/(np-1))-((i-1)*samples/(np-1)⇝
               ⇝), MPI_DOUBLE,i,0,MPI_COMM_WORLD,&s);
            for(int i=0;i<samples;i+=samples/output_x_samples) cout⇝
               ⇝ << u[i] << "␣";
            cout<<endl;
         }
         clog<<"root␣ok"<<endl;
      }
      MPI_Finalize();
      return 0;
}
```

4.2. Resultado

Nuevamente, los resultados son los mismos que el programa en serie



5. Compilación y ejecución

Los programas seriales se compilan con `g++` y se corren directamente (sin parámetros ni entrada, los valores están hardcodeados). Ejemplo: `g++ difusion.cpp -o difusion && ./difusion`

El programa con `MPI` tiene que ser compilado y ejecutado con `MPI`. Ejemplo: `mpiCC difusion_mpi.cpp -o mpi && mpiexec -np 5 mpi`

Todas las salidas son por `stdout`

Además hay un script hecho en python para visualizar los resultados que usa la librería `matplotlib`. Ejemplo: `./difusion | ./draw.py`
`mpiexec -np 5 mpi | ./draw.py`

6. To-Do

Cosas que quedaron por hacer:

- Medir los tiempos de ejecución y la performance del programa paralelo.

- Modificar los programas para que acepten los valores seleccionados por el usuario en la línea de comandos
- No está implementado el caso $\alpha = 0{,}5$
- El programa paralelo usa un nodo fantasma. Hay que modificarlo para que pueda usar 2,5, y 10 nodos fantasma.