Introduction to the Numerical Integration of PDEs.

B. Piette*
Department of Mathematical Sciences,
University of Durham,
Durham DH1 3LE, UK

Abstract

In these notes, we describe the design of a small C++ program which solves numerically the sine-Gordon equation. The program is build progressively to make it multipurpose and easy to modify to solve any system of Partial Differential Equation.

*e-mail address: B.M.A.G.Piette@durham.ac.uk

Contents

1 Introduction

The aim of these notes is to build progressively a computer program that will solve a system of Partial Differential Equations. The program will solve the sine-Gordon equation, but it will be written in such a way that it can easily be modified to solve any system of PDEs.

The prerequisites to read these notes are the following:

- Some knowledge on the numerical integration of Ordinary Numerical Equations [1] and in particular the 4th Order Runge Kutta method.
- Basic knowledge of C++ [2].
- Some knowledge on the sine-Gordon equation and non-linear differential equations in general[3].

The reader should first try to understand what a partial differential equation is and how it can be solved numerically. This is very briefly discussed in section 1.

In the next section, we describe how one solves PDE numerically. We start by sketching a few lines of code to solve a PDE with only one function.

We then design a fully working program, using a couple of classes. This first program is very limited but it outlines the structure of the more comprehensive application described in the 3rd and 4th section.

In section 3 we build a program that can solve any single PDE.

In the 4th section we develop the program from section 3 to make it solve any system of PDEs as well as to output a large class of derived data.

In the 5th section, we describe how to make modifications to the program `sineGordon` to solve other equations or output other type of data.

In the 6th section we develop a program to solve a static equation, or boundary value problem, using the relaxation method.

In section 7, we list some problems that one can try to solve. These suggest to solve the sine-Gordon equation for different type of initial conditions as well as solving other PDEs.

In the appendices we give the full listing of the programs that are described in section 4 to 6.

Note: all the programs listed in these notes have been tested using the GNU C++ compiler. One should be able to compile these programs using any other ANSI compliant compiler.

2 Discretisation of the Equation

The sine-Gordon equation is the following partial differential equation:

$$\frac{\partial^2 f}{\partial t^2} - \frac{\partial^2 f}{\partial x^2} = -\sin(f) \qquad (1)$$

where the function $f(x,t)$ is smooth for all $-\infty < x < \infty$. This equation describes a time evolution, or in other words, given any initial condition $f(x,t_0)$ and $\frac{\partial f}{\partial t}(x,t_0)$, one can use the equation to predicts $f(x,t)$ for all $t > t_0$.

The sine-Gordon equation has an Hamiltonian given by the expression

$$H = \int dx (\frac{1}{2}\frac{\partial f^2}{\partial t} + \frac{1}{2}\frac{\partial f^2}{\partial x} + 1 - \cos(f)). \tag{2}$$

2.1 Time integration

To solve the equation (1) numerically, it is convenient to rewrite it as a pair of first order differential equations:

$$\begin{aligned} \frac{\partial f}{\partial t} &= g \\ \frac{\partial g}{\partial t} &= \frac{\partial^2 f}{\partial x^2} - \sin(f) \end{aligned} \tag{3}$$

Both equations are now of the form

$$\frac{\partial f}{\partial t} = F(f(x,t)) \tag{4}$$

which formally looks like an ordinary differential equation. Expending $f(t)$ as a Taylor series around t_0 we can write

$$f(t_0 + dt) = f(t_0) + dtF(f(x,t_0)). \tag{5}$$

If we take dt small enough, we can use (5) to evaluated $f(t_0 + dt)$ and then apply (5) one more time to get $f(t_0 + 2dt)$ and so on for any $t > t_0$.

This is the so called Euler method. Its main advantage is to be very simple. Its main problem is that it is inaccurate and unstable. One should use a method like the 4th order Runge-Kutta method instead. The program `sineGordon` is written in such a way that it can use either, but we will stick to the Euler method for now as its simplicity makes it an ideal pedagogical choice.

2.2 Spacial discretisation

The problem to solve (3) using (5) is that the expression $F(f(x,t_0))$ is not an ordinary function, but a differential equation. We must thus find a way to evaluate the derivatives with respect to x.

To do this we discretize the real axis, that is, split it in a finite number of segments and evaluate the function $f(x)$ on a finite number of points. For this we consider a finite interval $[x_{min}, x_{max}]$ and split it in $N-1$ segments. We will then evaluate the function $f(x)$ at the points $-x_{min}, -x_{min}+$

$dx, -x_{min}+2dx, ..., x_{max}-2dx, x_{max}-dx, x_{max}$ where $dx = (x_{max}-x_{min})/N$. The criteria for choosing x_{min} and x_{max} is to make sure that the function $f(x)$ is virtually 0 , modulo 2π for the sine-Gordon equation, everywhere outside the range $[x_{min},x_{max}]$. At the same time, the interval must not be too large, otherwise too many points will be required and the integration will take too much time.

Having done this, we have a set of points that we will label x_0, x_1 ... x_{N-1} and a set of sampling values for the functions f which we label $f_i = f(x_i)$ for $i = 0$ to $N-1$.

To evaluate the derivatives, we use the Taylor expansion around x_i, and it is easy to show that the first and second derivatives can be approximated as follow:

$$\begin{aligned} \frac{\partial f}{\partial x}(x_i) &\approx \tfrac{f_{i+1}-f_{i-1}}{2dx} \\ \frac{\partial^2 f}{\partial x^2}(x_i) &\approx \tfrac{f_{i+1}+f_{i-1}-2f_i}{dx^2} \end{aligned} \tag{6}$$

To illustrate what we have learnt so far more concretely, we consider the following first order PDE of 1 function $f(x,t)$ which we refer to as the diffusion sine-Gordon equation:

$$\frac{\partial f}{\partial t} = \frac{\partial^2 f}{\partial x^2} - \sin(f). \tag{7}$$

This equation is very similar to (1), the difference being that we now have to solve a single 1st order equation instead of a pair of 1st order equations.

2.3 An example: the diffusion sine-Gordon equation

To solve the time evolution equation (7) we consider an initial value $f(x,0)$ at an initial time that we choose to be $t = 0$. We thus assume that we know f_i for all values of i. If we store these values in an array `double f[N]`, we can then integrate equation (7) from $t = 0$ to $t = dt$ using a few lines of code that will look as follow

```
// The variable that we need
int const N=101;
double f[N],fxx,temp[N],pi,dx,xmin,xmax,dt;
int i;

// Set some parameters
pi = atan(1.0)*4;
xmin = -10.0;
xmax = 10.0;
dx = (xmax-xmin)/(N-1);
dt = dx*dx*0.25;

// Compute an inital value
```

```
for(i=0; i < N; ++i)
{ f[i] =  exp(-(i-(N+1)/2)*(i-(N+1)/2)); // assume xmin = -xmax
}

// Evaluates the equation for all i
for(i=0; i < N; ++i)
{ fxx = (f[i+1]+f[i-1]+2*f[i])/(2*dx);
  temp[i] = fxx-sin(f[i]) ; // Compute the equation right hand side
}

// Perform a single Euler integration setp.
for(i=0; i < N; ++i)
{ f[i] += dt*temp[i];
}
```

Notice that we have to use a temporary variable, `temp`, to store the equation values before we perform the actual integration step. This is required to ensure that when we evaluate the derivatives at a point x_i, we don't use the value `f[i-1]` after it has been modified.

Several criticisms can be made about the code above:

- The evaluation of the initial condition is a bit clumsy. Usually one will use an explicit function of x, not i. So, for clarity, We should compute x from i and then evaluate the initial condition using x.
- The integration loop is incorrect. The reason is that near the edge of the interval of integration, we evaluate the derivative of f using values that do not exist: `f[-1]` and `f[N]`. We must thus treat the first and last value of `i` separately. In this case, it makes sense to use fixed boundary conditions (i.e. `f[0]` and `f[N-1]` do not change) and set `temp[0]=temp[N-1] = 0.0`.
- The equation we try to solve is embedded inside the integration loop. This is bad practise and it makes it difficult to reuse the code. We will address the issue more comprehensively in the next section, but for now we will simply evaluate the equation using a small subroutine. The same is true for the initial value. We will thus use a subroutine to evaluate that as well.
- Having integrated the equation, it would be useful to produce some output like, for example, a graphic of the function `f(x)` after integration. We will thus add one subroutine to perform this task: output the pairs of values `x_i,f(x_i)` in a file that can then be read by gnuplot to produce a graphic plot.
- It would also be useful to integrate the equation for several steps up to a given value of `t` rather than performing a single integration step.

Our improved program sketch is then the following

```
int const N=101;
double f[N],fxx,temp[N],pi,dx,xmin,xmax,dt,t,tmax,x;
int i;

// Evaluate the equation for the given value of f and fxx
double Equation(double f, double fxx)
{ return(fxx-sin(f));
}

// Compute the intial condition
double Initial_value(double x)
{ return(exp(-x*x));
}

// Perform a single Euler integration step.
void Euler_1_step()
{ // Evaluates the equation for all i
  temp[0] = temp[N] = 0.0;
  for(i=1; i < (N-1); ++i)
  { fxx = (f[i+1]+f[i-1]+2*f[i])/(2*dx);
    temp[i] = Equation(f[i],fxx) ;
  }

  // Euler integration setp.
  for(i=0; i < N; ++i)
  { f[i] += dt*temp[i];
  }
}

// Output the function values as pairs  x_i f_i
void graph(char *file_name)
{ ofstream ofs(file_name));

  for(i=0; i < N; ++i)
  { ofs << xmin+i*dx << " " << f[i] <<"\n";
  }
}

// Parameters
pi = atan(1.0)*4;
xmin = -10.0;
xmax = 10.0;
dx = (xmax-xmin)/(N-1);
dt = dx*dx*0.25;
tmax = 5.0;

// Compute the inital value
for(i=0; i < N; ++i)
{ x = xmin+i*dx;
  f[i] = Initial_value(x); // inital value
}

t = 0.0;
while(t < tmax)
```

```
{ Euler_1_step();
  t += dt;
}

graph("test_x_f.txt");
```

Our program is starting to take shape, but it must still be improved substantially.

- So far, we have been using global variables to store the function `f` and other parameters. This is dangerous, one should rather use classes where, for all practical purposes, the parameters of the class look like global variable for member functions.

- The program only produces one graphic and in a predetermined format. We should try to produce graphics at regular interval during the integration to monitor the time evolution of the solution. This is what we are interested in after all.

- Our program so far can only solve a single first order differential equation. We should write our program in such a way that it can solve any system of any number of differential equations.

- The program parameters are all entered by hand in the code. One should not have to recompile the program each time a parameter is changed. These parameters should thus be read from a file, including the number of sampling points for the grid.

- It is often desirable to output more data than the functions themselves. In particular, one could produce graphic data for the energy density, or, for solitonic equations, monitor the time evolution of the position of the solitons.

- As we said above, the Euler method is very simple but quite inaccurate. One should use the 4th order Runge-Kutta method instead. We should write our program in such a way that we can easily implement any other integration method as well.

- We should disentangle what is specific to the equation we want to solve from the general method that we implement to solve them. The program should be made of 2 or more files; the main file will contain the code specific to the equation being solved while the others will be (class) libraries which can be reused without modification to solve any other equation. This will substantially improve the reliability of the programs as once the library classes have been thoroughly checked, they can safely be used for any other equation.

2.4 Choosing the time integration step.

In our program so far, we have chosen dt to be $dx^2/4$. The reason for this is the following: for the integration method to work at all, the time step dt must be sufficiently small. As a matter of fact, each integration method has its own lower bound. If the time step is larger than that value the integration simply does not work. Moreover the bound on dt differ depending on the type of equation that one solves. The Euler and the Runge-Kutta method have essentially the same bound.

For a diffusion equation of the type

$$\frac{\partial f}{\partial t} = \frac{\partial f^2}{\partial x^2} + ... \tag{8}$$

we must take $dt < dx^2$ while for a wave equation of the type

$$\frac{\partial f^2}{\partial t^2} = \frac{\partial f^2}{\partial x^2} + ... \tag{9}$$

the condition is $dt < dx$. This last constraint is easy to justify for the Euler method. It you think about it, it says that any signal must be able to propagate at the speed of light, equal to 1 in our dimensionless units. Indeed, at every integration step, a signal propagates from one point to the next meaning that its speed of propagation is at most equal to dx/dt.

For the Runge-Kutta method one might be tempted to take $dt < dx/4$ as each integration is made of four successive steps. Actually this is wrong; each Runge-Kutta sub-step is an approximation of the same single step dt and the bound on dt is the same as for the Euler method.

In practise, one should take a value of dt smaller that the upper bound given above. A good choice is to take $dt = 0.5dx$ or $dt = 0.25dx$. Taking smaller dt is usually unnecessary, especially when using the Runge-Kutta method. If you find yourself having to decrease the value of dt to get a reasonable accuracy, say to improve the conservation of energy, then you should decrease the lattice interval dx as well, and keep the ratio between dx and dt the same. The reason for this is that the numerical evaluation of the spacial derivatives is of second order only while the time integration, for the 4th order Runge-Kutta method, is of the 4th order.

3 Designing the Application Using Classes.

In this section we design a program that can be used to solve any reasonable 1st order differential equation. By this we mean that to solve a different equation, the user will only have to make a minimum amount of modification to the program we are about to write. The only major changes will be ,the equation itself and the initial condition subroutines.

3.1 A virtual class to solve a single PDE.

First of all, let us investigate how the use of classes can help in writing a program that is reliable and reusable. We start by constructing a virtual class called `pde` that will contain all the parameters necessary to solve a general first order single differential equation.

We also construct a class `grf_pde` which will be used to output the function values into a file. An instance of the `grf_pde` class will then be passed to the instance of the `pde` that is being solved. The reason to use a class for this is to make it possible to output data in different formats. Should one want to do this, all that is required is to create a class derived from `grf_pde` to output data in a different file format, and pass an instance of this class to `pde`.

We will then derive the class `pde_euler` from `pde`; it will implement the Euler method. Similarly, a class `pde_RK4` will be derived to implement the 4th order Runge-Kutta method.

Notice that, following [4], we add an underscore suffix to every non-function member of the class.

Here follows the header file `pde.h` in which the class `pde` is declared.

```
#ifndef PDE_H
#define PDE_H
#include <grf_pde.h>
#include <cmath>
#include <fstream>
#include <cstdio>

class pde {
public:
         // constructors and destructors
  pde(double xmin,double xmax,int nx);
  virtual ~pde();

         // Method to Override for specific equation
  virtual void Eqn(double &F)=0;           // Equations to solve
  virtual void Init(double x,double &F)=0; // Initial value
  void init_grf(grf_pde *grfp) { grfp_ = grfp; };
  void set_dt(double dt) { dt_ = dt;}

  double xmin() { return(Xmin_);}
  double xmax() { return(Xmax_);}
  double dx() { return(dx_);}
  int    nx() { return(nx_);}

  virtual void set_neighbour_fcts(double *f,int i);
  virtual void eval_derivatives();

         // integration method overriden by sub-classes
  virtual void initial_value();
  void write_grf();       // Output one graph
```

```
  virtual double integrate_1step() {return(0.0);}; // to override

         // Calls integrate_1step() several times
  virtual double integrate_until(double t);
  virtual double integrate_grf_until(double tmax,double grf_dt);

protected:
  double dt_;
  double t_;
  double Xmin_;
  double Xmax_;
  double dx_;

  int nx_;

  double *f_,*fl_,*fll_,*fr_,*frr_; // fcts pointers
  double x_;                 // local value of x = Xmin_+i_*dx_
  int i_;                    // local value of grid index

         // allocate arrays for the fcts and densities
  double *Fcts_;     // A vector of size nx

  double Force_;       // The evaluated equation
  double fx_,fxx_,fxxx_,fxxxx_;

  // Graphic variables
  grf_pde *grfp_;
  int grf_index_;
};
#endif
```

Source file `pde.h`

The methods of this class are used in the program `difSineGordon`.

Let us analyse the members of this class. Most methods are used in the program `difSineGordon`, so it will be useful to look at the listing of that file below while reading the descriptions that follow:

`pde(double xmin,double xmax,int nx)` : The constructor for the class `pde` takes 3 arguments: the boundary of the lattice range, `xmin` and `xmax`, and the number of sampling points. The main task of the constructor is to allocate an array of size `nx` for the function value; this array pointer is stored in the member `Fcts_`.

`Eqn(double &F)` : This method will be overridden in the main program to correspond to the equation one wants to solve. It evaluates the right hand side of the equation at a single point and return the result in the reference `F`.

The method can use the following members which are guaranteed to be correctly initialised:

- `f_`: the value of the function at the local point.
- `fl_`: the value of the function one point to the left.
- `fll_`: the value of the function two points to the left.
- `fr_`: the value of the function one point to the right.
- `frr_`: the value of the function two points to the right.
- `x_`: the value of `x` at the local point.
- `i_`: the index value of the local point. One should try to avoid using it.

These can be used to evaluate the derivatives but one should rather call the method `eval_derivatives()` instead.

The motivation to use a reference to `F` rather than returning the value come from our aim to generalise this to a system of PDE. Then, the parameter will be a pointer to an array in which the values must be returned.

`set_neighbour_functions(double *f,int i)` : This method sets the variable `f_`, `fl_`, `fll_`, `fr_`, `frr_`, `i_` and `x_` used by the method `Eqn()`. It is called by the integration method before each call to `Eqn()`.

`eval_derivatives()` : This method should only be called from `Eqn` to evaluate the first and second derivatives of the function and store them respectively in `fx_` and `fxx_`. It only uses the members `f_`, `fl_` and `fr_`. This method can be overridden to evaluate higher derivatives as well. It can also use `fll_`, `frr_`, `x_` and `i_`

`Init(double x,double &F)` : This method will be overridden in the main program to correspond to the initial condition one wants to solve. It is called by the method `initial_value()`. It computes the initial value of the function. The initial value must be copied into `F`.

`initial_value()` : This method sets the in ital value for the functions. It calls the method `Init()` for each point on the lattice. It can be overridden to set the initial condition differently, say by loading data from a file.

`init_grf(grf_pde *grfp)` : After creating a instance of `pde`, one can create an instance of a `grf_pde` for that `pde`. One must then pass the `grf_pde` pointer to the `pde` object, using the method `init_grf`, so that it can create the graphic files.

`integrate_grf_until(double tmax,double grf_dt)` : Integrate the pde up to `tmax` and output graphic files at a time interval given by `grf_dt`.

Most of the methods are defined in the file `pde.cc`:

```
#include <pde.h>
#include <iostream>

using namespace std;

// pde constructor
// xmin, xmax : lattice range
// nx : number of lattice points
//******************************
pde::pde(double xmin,double xmax,int nx):
  t_(0.0),Xmin_(xmin),Xmax_(xmax),nx_(nx)
{ dx_ = (Xmax_-Xmin_)/(nx_-1);
  dt_ = dx_*dx_*0.25;
  grf_index_ = 0;

  try
  { Fcts_  = new double[nx_];
  }
  catch(...) { throw((string)"No more memory!"); }
}

// Destructor
//************************
pde::~pde()
{ delete [] Fcts_;
}

// Set f_, fl_, fll_, fr_, frr_, x_ and i_.
// f : fct array.
// i : index position of the grid
//****************************************
void pde::set_neighbour_fcts(double *f,int i)
{
  x_ = Xmin_+i*dx_;
  i_ = i;

  f_   = &f[i];
  fl_  = &f[i-1];
  fll_ = &f[i-2];
  fr_  = &f[i+1];
  frr_ = &f[i+2];

  if(i==0)           { fl_ =  0; fll_ = 0; }
  else if(i==1)      { fll_ = 0; }
  else if(i==(nx_-2)){ frr_ = 0; }
  else if(i==(nx_-1)){ fr_ =  0; frr_ = 0; }

}

// Evaluate the derivatives of the function.
// f_, fl_, fr_, point respectively to current lattice point,
//               the point on left and the one on the right.
// The result is put into fx_ and fxx_.
// The method is called by the integration routine before calling
```

```
//   the equation method.
//*********************************************************
void pde::eval_derivatives()
{ if(!fl_) // left edge of the grid
  { fx_ = (fr_[0]-f_[0])/dx_;
    fxx_ = (f_[0]+fr_[1]-2*fr_[0])/(dx_*dx_);
    return;
  }
  if(!fr_) // right edge of the grid
  { fx_ = (f_[0]-fl_[0])/dx_;
    fxx_ = (fl_[-1]+f_[0]-2*fl_[0])/(dx_*dx_);
    return;
  }

  fx_ = (fr_[0]-fl_[0])/(2*dx_);
  fxx_ = (fl_[0]+fr_[0]-2*f_[0])/(dx_*dx_);
}

// Initialise the Fcts_
// Calls Init() for every value x_i
//*********************************
void pde::initial_value()
{ int i;
  double x;

  x = Xmin_;

  for(i=0; i < nx_; i++)
  { Init(x,Fcts_[i]);
     x += dx_;
  }
}

// Create a graphic file for the selected fcts and densities
// Throw a string on error
//*********************************************************
void pde::write_grf()
{ grfp_->write_1_grf(grf_index_,Fcts_,t_);

  grf_index_++;
}

// Perform n integration steps on the grid
// By calling  integrate_1step several times.
// Return : the updated time
//*********************************************
double pde::integrate_until(double t)
{
  while(integrate_1step() < t);

  return(t_);
}

// Perform n integration step on the grid with graphics
```

```
// Output graphs every grf_dt
// Update the time t_
// tmax : integration bound.
// grf_dt : time interval between graphics
// Return : the updated time
//*************************************************************
double pde::integrate_grf_until(double tmax,double grf_dt)
{ int i;
  string grf_name;
  double grf_t;

  grf_t = t_ + grf_dt;

  while(t_ < tmax)
  { if(t_ > grf_t) { write_grf(); grf_t += grf_dt; }
    integrate_1step();
    i++;
  }

  if(t_ > grf_t) write_grf();

  return(t_);
}
```

Source file `pde.cc`

3.2 A class to generate PDE graphic files.

We now design the class `grf_pde` which will be used to output data in a file. The class contains only a few members:

`grf_pde(pde *pde,string file_name)` : The constructor only has two parameters. The first one, `pde` is used to get the PDE lattice parameters `xmin`, `xmax`, `dx` and `nx`. The second parameter is the prefix name for the file into which the graphics data will be output.

`write_1_grf(int index,double *p,double t)` : This method generates a graphic file for the data pointer to by `p`, that is `f[i] := p[i]`. The first argument, `index` is the file index. Each graphic file must have a different index number, if only to make sure their filenames differ. This index is incremented by the integration method that calls `write_1_grf()`. The 3rd argument is the current integration time. It is written as a comment at the top of the graphic file.

The name for the output file is created using the method `make_file_name` and the suffix is always `txt`.

The method is called by the `pde` integration method.

`make_file_name(string name,int index,string suffix)` : This method generates a name using a prefix, `name`, an index value and a suffix. The three components are concatenated in that order, but with an underscore separator between the name and the index and a dot before the suffix.

```
#ifndef GRF_PDE_H
#define GRF_PDE_H
#include<string>

class pde;
class grf_pde {
public:
  grf_pde(pde *pde,std::string file_name);

  virtual ~grf_pde() {};

  virtual void write_1_grf(int index,double *p,double t);
  std::string make_file_name(std::string name,int index,std::string suffix);

protected:
  int nx_;
  double xmin_;
  double xmax_;
  double dx_;

  std::string file_name_;
};
#endif
```

Source file `grf_pde.h`

```
#include <pde.h>
#include <grf_pde.h>
#include <iostream>

using namespace std;


// Constructor
// Get the parameters from the pde
// file_name the name for the output file
//***************************************
grf_pde::grf_pde(pde *pde,string file_name) : file_name_(file_name)
{ nx_    = pde->nx();
  xmin_  = pde->xmin();
  xmax_  = pde->xmax();
  dx_    = pde->dx();
};

// Ouput a single txt 1D graphic file
// findex : file index
// p : pointer to the data array
// t : value of t for the snapshot
// Output format:
```

```
// The file should be a .m file. It can be execute by matlab
//    # t = t
//    ti Fi
// The file can be plotted using the following gnuplot command:
//    gnuplot> plot 'FILENAME' with lines
// Throw a string on error
//***************************************************************
void grf_pde::write_1_grf(int findex,double *p, double t)
{ int ix,index;
  double x;
  string fname = make_file_name(file_name_,findex,"txt");

  ofstream ofs(fname.c_str());

  if(!ofs)
  { throw((string)" Error: Can't open file : "+fname);
  }

  ofs <<"# t = "<<t<<";\n";
  index = 0 ;
  x = xmin_;
  for(ix = 0; ix < nx_; ix += 1)
  { ofs << x << " " << p[index] << "\n";
    ++index;
    x += dx_;
  }
}

// Create a file name of the form:
//    NAME_DDD.XXX
// where NAME is the string name
//       DDD is the index
//       XXX is the suffix
// Return the nameas a string
//******************************
string grf_pde::make_file_name(string name,int index,string suffix)
{ char Buff[40];

  sprintf(Buff,"%d",index);

  return(name+"_"+Buff+"."+suffix);
}
```

Source file `grf_pde.cc`

3.3 A class to implement the Euler method.

We now derive the class `pde_euler` from the class `pde`. This class will implement the Euler method as the integration procedure.

It overrides only 3 methods. The constructor allocates an extra array, while the destructor deletes it. The method `integrate_1step` performs a single integration step. This method must call the method `pde::set_neighbour_points` before calling Eqn().

```
#ifndef PDE_EULER_H
#define PDE_EULER_H
#include <pde.h>

class pde_euler : public pde {
public:
  pde_euler(double xmin,double xmax,int nx);

  ~pde_euler();
  virtual double integrate_1step();

protected:
  double *tmp1_;        // Same size as Fcts_
};
#endif
```

Source file pde_euler_.h

```
#include <pde_euler.h>
#include <iostream>

using namespace std;

// Constructor for Euler pde
// Allocated 2 extra arrays
//*****************************
pde_euler::pde_euler(double xmin,double xmax,int nx)
   : pde(xmin,xmax,nx)
{ tmp1_ = new double[nx_];
}

// Destructor
//***********
pde_euler::~pde_euler()
{ delete [] tmp1_;
}

// Perform a single euler integration step on the grid
// Update the time t_
// pde : pde parameters
// Eqn(Force) : computes the right hand side of
//     the equation.
//       uses f_,fl_,fr_: local fct pointer
//       Force : Reference to a double for the result.
//       i_    : lattice index
// Return : the updated time
//****************************************************
double pde_euler::integrate_1step()
{ int i;

      // Scan the whole grid
```

```
  for(i=0; i < nx_; i++)
  {     // set f_, fl_, fr_, x_ and i_
    set_neighbour_fcts(Fcts_,i);

        // Computes the force for point i and
        // stores it in Force
    Eqn(Force_);

    // Update each fct using Force : f(n+1) = f(n) + Force*dt
    // tmp1 is used to store the fct temporarily

    tmp1_[i] = f_[0]  + dt_*Force_;
  }

  /* Update Fct with the new fct */
  for(i=0; i < nx_; i++)
  { Fcts_[i] = tmp1_[i];
  }
  t_ += dt_;  // Udate the integration time
  return(t_);
}
```

Source file `pde_euler.cc`

3.4 A class to implement the 4th order Runge-Kutta method.

We now derive the class `pde_RK4` from the class `pde`. This class will implement the 4th order Runge-Kutta method as the integration procedure.

It overrides only 3 methods. The constructor allocates three extra arrays, while the destructor deletes them. The method `integrate_1step` performs a single integration step using 4 sub-steps. Each intermediate step is performed by the method `sub_RK4_1step`.

The Runge-Kutta method states that the value of the function $f(x, t+dt)$ can be evaluated from $f(x,t)$, using the expression $f(x,t+dt) = f(x,t) + dt(k_1(x) + 2k_2(x) + 2k_3(x) + k_4(x))/6$. Calling $F(t, f(x,t))$ the right-hand side of the equation for the function $f(x,t)$ at the specified value of x and t, then $k_1(x) = F(t, f(x,t))$, $k_2(x) = F(t+dt/2, f(x,t)+dtk_1/2)$, $k_3(x) = F(t+dt/2, f(x,t)+dtk_2/2)$ and $k_4(x) = F(t+dt, f(x,t)+dtk_3)$.

The method `sub_RK4_1step(double *f,double *ef,double *mf,double *nef, double Cnef,double Cmf)` evaluate the right-hand side of the equation using `Eqn` for the function estimation `ef`. It return the value $f(x,t) + dtk_iC_{nef_i}$ in `nef`, for $i = 1$ to 3 and where $C_{nef_1} = C_{nef_2} = 0.5$, and $C_{nef_3} = 1$. It also adds the value $dtk_iC_{mf_i}$ for $i = 1$ to 4 to the array `mf` where $C_{mf_1} = C_{mf_4} = 1/6$ and $C_{mf_2} = C_{mf_3} = 2/6$.

The method `integrate_1step` calls `sub_RK4_1step` 4 times, to compute each k_i and sum up their value in the array `tmp1_`. Then it performs the actual integration by adding $dt(k_1(x) + 2k_2(x) + 2k_3(x) + k_4(x))/6$ to the function $f(x,t)$.

Notice that the method `sub_RK4_1step` calls the method `set_neighbour_fcts` for a different array at each step. This is how the variable `f_`, `fl_` and `fr_` are set to point to the correct function values. This shows why it is important to only use these 3 pointers to evaluate the equation rather than using the variable `Fcts_` directly.

```
#ifndef PDE_RK4_H
#define PDE_RK4_H
#include <pde.h>

class pde_RK4 : public pde {
public:
  pde_RK4(double xmin,double xmax,int nx);

  ~pde_RK4();
  virtual double integrate_1step();
  void sub_RK4_1step(double *f,double *ef,double *mf,double *nef,
                     double Cnef,double Cmf);
protected:
  double *tmp1_;        // Same size as Fcts_
  double *tmp2_;        // Same size as Fcts_
  double *tmp3_;        // Same size as Fcts_
};
#endif
```

Source file `pde_RK4_.h`

```
#include <pde_RK4.h>
#include <iostream>

using namespace std;

// Constructor for RK4 pde
// Allocated 2 extra arrays
//*****************************
pde_RK4::pde_RK4(double xmin,double xmax,int nx)
   : pde(xmin,xmax,nx)
{ tmp1_ = new double[nx_];
  tmp2_ = new double[nx_];
  tmp3_ = new double[nx_];
}

// Destructor
//***********
pde_RK4::~pde_RK4()
{ delete [] tmp1_;
  delete [] tmp2_;
  delete [] tmp3_;
}

/*******************************************************/
/* Perform a 4th order Runge-Kutta integration step on    */
/* f(n+1) = fn + 1/6(k1 +2 k2 + 2 k3 + k4)                */
```

```
/* k1 = dt F(t,fn)                                       */
/* k2 = dt F(t+dt/2,fn+k1/2)                             */
/* k3 = dt F(t+dt/2,fn+k2/2)                             */
/* k4 = dt F(t+dt,fn+k3)                                 */
/* the grid. Update the time in pde.                     */
/* Eqn(Force) : a function which computes the right      */
/*     hand side of the equation.                        */
/*       Force : vector where to store the result.       */
/*       i_    : lattice index                           */
/* Return : the updated time                             */
/*********************************************************/
double pde_RK4::integrate_1step()
{ int i;

    // tmp1 is used to compute fn + 1/6(k1 + 2 k2 + 2 k3 + k4)
    // We initialise it to 0
  for(i=0; i < nx_; i++)
  { tmp1_[i] = 0;
  }

    // tmp1 = 0
  sub_RK4_1step(Fcts_,Fcts_,tmp1_,tmp2_,0.5,1.0);

    // tmp1 = k1
    // tmp2 = fn + 1/2 k1
  t_ += 0.5*dt_;
  sub_RK4_1step(Fcts_,tmp2_,tmp1_,tmp3_,0.5,2.0);

    // tmp1 = fn + 1/6 (k1 + 2 k2 )
    // tmp3 = fn + 1/2 k2 */
  sub_RK4_1step(Fcts_,tmp3_,tmp1_,tmp2_,1.0,2.0);

    // tmp1 = fn + 1/6 (k1 + 2 k2 + 2 k3)
    // tmp3 = fn + k3 */
  t_ += 0.5*dt_;
  sub_RK4_1step(Fcts_,tmp2_,tmp1_,0,0.0,1.0);

    // tmp1 = (k1 + 2 k2 + 2 k3 + k4)
    // we update fn => fn + tmp1 / 6
  for(i=0; i < nx_; i++)
  { Fcts_[i] += tmp1_[i]/6.0;
  }

    // Time has been updated before the last step
  return(t_);
}

/*************************************************************/
/* Perform one RK intermediate step                          */
/* f : f(n) the fcts at time tn                              */
/* ef : estimated f for this step.                           */
/* mf : modified f : used to add the contribution of each step*/
/* nef : estimated f for the next step, must be computed here */
/* Cnef : coeficiant used to computed nef                    */
```

```
/* Cmf  : coeficiant used to computed mf                      */
/************************************************************/
void pde_RK4::sub_RK4_1step(double *f,double *ef,double *mf,double *nef,
                             double Cnef,double Cmf)
{ int i;

  for(i=0; i < nx_; i++)
  { set_neighbour_fcts(ef,i);

        // Computes the force for point i and
        // stores it in Force
    Eqn(Force_);

        // Update each fct using Force : f(n+1) = f(n) + Force*dt
    { mf[i] += dt_ * Force_ * Cmf;

      if(nef) // nef will be 0 for the last step
      { nef[i] = f[i] + dt_ * Force_ * Cnef;
      }
    }
  }
}
```

Source file `pde_RK4.cc`

3.5 The main program.

The main program starts by deriving a class, `dsg_pde`, which overrides the functions `Eqn` and `Init`. The two methods implement the diffusion sine-Gordon equation for an initial condition corresponding to a Gaussian lump.

Next follows the initialisation of a few parameters followed by the main part of the application. First we instantiate a `dsg_pde` for which we set the time increment to $0.5dx^2$.

We then create an instance of a `grf_pde` and pass its pointer to the `pde`.

Finally we compute the initial condition and integrate the equation up to `tmax`.

```
#include <pde_euler.h>
#include <grf_pde.h>
#include <iostream>
#include <iostream>

using namespace std;

// sine-Gordon diffusion Equation

class dsg_pde : public pde_euler
{
public:
  dsg_pde(double xmin,double xmax,int nx) :
    pde_euler(xmin,xmax,nx)
```

```
  {
  }

  // Evaluate the sine-Gordon equation
  // functions:
  // f[0] : f : sine Gordond fct
  // F : array for result
  //**********************************
  void Eqn(double &F)
  { eval_derivatives();

    Force_ = fxx_ - sin(f_[0]);
  }

  // Compute the initial condition.
  // Fcts_[i]   : f(x_i)
  //*******************************
  void Init(double x,double &F)
  { double xm;
    xm = (Xmin_+Xmax_)*0.5;
    F = exp(-(x-xm)*(x-xm));
  }
};

double xmin = -10.0;
double xmax =  10.0;
double tmax =   5.0;
double grf_dt = 0.5;

int nx      =   101;
char *file_name = "test";

int main(int argc,char **argv)
{ try
  {    // Build a pde for sineGordon
    dsg_pde pde(xmin,xmax,nx);

       // Set dt by hand : dx * par.dt_r
    pde.set_dt(pde.dx()*pde.dx()*0.25);

       // Create a new graphic object
    grf_pde *grfp = new grf_pde(&pde,file_name);
    pde.init_grf(grfp);

       // compute the initial condition
    pde.initial_value();

       // integrate up to tmax;
    pde.integrate_grf_until(tmax,grf_dt);
  }
  catch(string e)
  { cerr << e << std::endl;
  }
}
```

Source file `difSineGordon.cc`

There are several points that we have not addressed yet in this program. We will consider them in the next section one by one.

4 A General Purpose PDE Application.

The main limitation of our first program is that it can only solve a single equation. To solve a system of equations, we must somehow replace the scalar function value by a vector. To do this, we must make a few changes to the classes `pde` and `grf_pde`.

To start with we will rename the classes `pde_1d` and `grf_pde_1d` respectively in anticipation that we might create a similar class to solve higher dimensional problems.

The full listing of the program is given in section 7.

4.1 System of Partial Differential Equations.

The class `pde_1d` will be a copy of the class `pde` but several changes and addition will be made to it:

- A new member,`int NoEq_`, is added to the class to specify the number of equations that is being solved.
- `double *Fcts_` is now allocated as an array of size `nx_*NoEq_`, *i.e* an array of `nx_` vectors of `NoEq_` elements each.
- `double Force_` must be replaced by `double *Force_` and become an allocated array, of size `NoEq_`, created by the constructor.
- `double fx_` and `double fxx_` must be replaced by respectively `double *fx_` and `double *fxx_` an become allocated arrays of size `NoEq_` created by the constructor. The method `eval_derivatives()` must compute the derivative of every function and store them in these arrays.
- The pointers `f_`, `fl_`, `fll_`, `fr_` and `frr_` point to the first element of the vector of functions.
- The method `Eqn(double &F)` becomes `Eqn(double *F)`, *i.e* its argument becomes an array of size `NoEq_`.
- As there are several function to plot, the graphic class `grf_pde` must use a different filename for each graph. One must thus add the member `char const **Fct_Names_` to the class `pde`. This member points to an array of `NoEq_` null terminated strings. Each string is a short name for the corresponding function. The functions will then be output in the

files whose names are made by concatenating the strings `file_name_`, to the corresponding entry in `Fct_Names_` and the `grf_index_`.

4.2 Reading the Input Parameters.

The parameters are still entered as constant value in our first program. To change this we use the class `input_par`, described in the appendix, to read each parameter from a file. Each line in the parameter file is of the form `NAME VALUE` where `NAME` is a predefined name and `VALUE` is the corresponding value.

First of all, we group all the parameters into a global structure called `par`:

```
struct {
  double x_min;
  double x_max;
  int nx;
  double dt_r;
  double t_max;
  double k;
  int steps;
  int grf_steps;
  int stats_steps;
  char grf_name[256];
} par;
```

The parameters are then read in the correct order as follow

```
// The parameter filename is the 1st argument
input_par ip(argv[1]);

ip.get("x_min",par.x_min);  // read xmin
ip.get("x_max",par.x_max);  // read xmax
ip.get("nx",par.nx);
ip.get("dt_r",par.dt_r);
ip.get("t_max",par.t_max);
ip.get("k",par.k);
ip.get("steps",par.steps);
ip.get("grf_steps",par.grf_steps);
ip.get("stats_steps",par.stats_steps);
ip.get("grf_name",par.grf_name);
```

where `get` reads a double value, an integer value or a string. This is described in more detail in the first appendix.

4.3 Derived quantities.

Aside from the solutions of the system of PDEs, it is also often useful to output other quantities. These quantities can be split in three categories:

- Densities: these are functions defined on the same interval as the solution of the PDE and computed from these solutions and their derivatives. A good example is the energy density.

- Integrated densities. The integrated value of these densities over the whole lattice range can also be useful. Integrating energy density for a solution will yield its total energy.

- Properties of the solutions, like for example the position of a lump, or the maximum value of a function. These are numbers so from now on we will refer to these quantities as scalars.

To output the values of the integrated densities and the scalars, we create a new file, suffixed with the character `s`. The integration method then append these data at regular interval into that file. The first column will be the integration time while the other column will be the numerical values of these quantities.

We thus make the following addition to the class `pde_1d`:

- Add the member `int NoDens_` corresponding to the number of densities that are computed.

- Add the method `double *Densities_`. This is an array of size `nx_*NoDens_` in which the densities are stored.

- The method `Dens(double *D)`, like `Eqn`, is overridden in the instantiated class derived from `pde_1d` for any specific problem. This method evaluates each density locally, exactly like `Eqn`. The densities are put into `D` an array of `NoDens_` elements.

  The method also return the measure that must be used to integrate the density. If we call $M(i)$ the value returned at point i and $D(i)$ the density value at that point, the integrated density is given by the expression $\sum_{i=0..(nx-1)}(M(i)D(i))$. The measure is usually 1 or any other scaling constant. When solving a radial problem, the measure is then $2\pi x$ where x is the radial variable (cfr problem 6.7).

- The method `char const **Density_Names_` is a pointer to an array of names for the densities. It plays the same role as `Fct_Names_` but for the densities.

- Add the member `int NoScalars_` corresponding to the number of scalars quantities that are computed.

- Add the member `double *Scalars_`. This is an array of size `nx_*NoScalars_` in which the scalars are stored.

- The method `Scalars()` computes all the scalar quantities and store them in the array `Scalars_`.

- The method `char const **Scalars_Names_` is a pointer to an array of names for the scalars. It plays the same role as `Fct_Names_` but for the scalar quantities.

- The method `write_grf()` generates a graphic file for each function and each density.
- The method `write_scalars()` append the scalar data and the integrated densities into the scalar file as a single line.
- The method `write_header_scalars()` writes the header for the scalar data file. The header list the name of the Integrated densities and the scalar data as comments. This method is called once by the integration method.

4.4 Boundary conditions.

Several other changes have been made to the class `pde_1d` and `grf_pde_1d`:

- The method `void Boundary(double *fct)` has been added to the class `pde_1d`. Like the method `Eqn` and `Dens` it is overridden in the derived class. It can be used to implement non trivial boundary conditions. If, for example, one wants to impose a von Neumann boundary condition on both sides of the grid for the first functions, then `Boundary` will contain the statements

  ```
  fct[0] = fct[NoEq_]; // Left hand side
  fct[(nx_-1)*NoEq_] = fct[(nx_-2)*NoEq_];
  ```

  The derivative of the function is set to 0 by hand. The method argument, `fct`, is a pointer to the first element of the function array. It will sometime point to the allocated array `Fcts_` but often, it will refer to an auxiliary allocated array. For that reason, the pointer `fct` must be used instead of modifying `Fcts_` directly.

4.5 Periodic boundary conditions.

To implement periodic boundary conditions, several modifications must be implemented. We can choose between three techniques:

- The first one consist in identifying the first and the last point on the grid: `f[nx_ -1] = f[0]`. If we do this, we must also modify the `pde_1d::computeDensities` method to perform the integration of the densities properly: the last point must not be taken into account. We must also modify every implementation of the method `integrate_1step` in derived classes, to copy `f[nx_ -1]` into `f[0]`. This means in particular that for every integration method we must derive a new class specifically for periodic boundary condition. The key modification is to change the method `pde_1d::set_neighbour_functions` so that the points at the edges of the grid are made neighbours of each other.

- The second method is to discretized the compact interval differently by having `nx_` points and `nx_` intervals, or in other words use `dx = (Xmax-Xmin)/nx`. With this technique, we must also modify the method `pde_1d::set_neighbour_functions` to make the points at the edges of the grid neighbours of each other. We must also derive a new class for the periodic version of every integration method, except that this time we only need to override the constructor; the method `integrate_1step` does not need to be changed. It is also advisable to derive new classes for the class `grf_pde_1d`. The reason for this is that when plotting the graphic of a periodic function one expects the value of $f(Xmax)$ to be the last function value. As it is, the class `grf_pde_1d` would output $f(Xmax - dx)$ as the last value.

- The last method consist in adding one parameter to describe the type of grid and to check that parameters in every method that depends on it. In anticipation that one might generalise the class `pde_1d` to a two-dimensional version, and that there are more types of boundary condition in dimension higher that 1, one would add to the class `pde_1d` an enumeration listing every recognised type of lattice. In this case we would have two type: `LATTICE_TYPE_REGULAR` and `LATTICE_TYPE_PERIODIC`. One must still chose between the two types of discretisation described in the two paragraphs above.

The last technique is probably the best as it avoids having to derive a new class for every type of boundary condition. From a pedagogical point of view, the other two are better as they highlight the difference between periodic and non-periodic boundary conditions. There are very few differences between the first two methods, but we have chosen the second one.

Having defined a new class, say `pde_1d_RK4_periodic`, all one has to do to solve a given equation with periodic boundary conditions is to derive the class for that equation from `pde_1d_RK4_periodic` instead of `pde_1d_RK4`.

For the sin-Gordon equation it is slightly more difficult because the function f is defined modulo 2π and one is thus obliged to modify the method `eval_derivatives` to take this fact into account. This will be set as one of the problems to solve.

4.6 Running `sineGordon`.

To run the program `sineGordon`, one starts by compiling it by executing the command

```
% make sineGordon
```

Then one has to create a parameter file, let us call it `sineGordon.par`, looking like this:

```
x_min -10
x_max 10
nx 101
dt_r 0.25
t_max 20
grf_dt 1
scalars_dt 1
grf_name DATA/SGtests
k 1.0
```

and run the command

```
% sineGordon sineGordon.par
```

The graphic data files, `SGtests_0.txt`, `SGtests_1.txt` ... `SGtests_20.txt`, will be saved in the directory `DATA` and the scalar file will be called `DATA/SGtests.s`.

5 Relaxation method.

In this section we will look at the problem of computing the static solutions of a differential equation using the relaxation method. The problem is to find the static, time independent, solutions of the equation

$$\frac{\partial^2 f}{\partial t^2} = \frac{\partial^2 f}{\partial x^2} - V(f), \tag{10}$$

where $V(f)$ is any potential of f, or equivalently solve the equation

$$\frac{\partial^2 f}{\partial x^2} - V(f) = 0. \tag{11}$$

In one dimension this is the same as solving an Ordinary Differential Equation with fixed boundary values, which can be solved using the so called shooting method. In these notes we will restrict ourselves to the relaxation method.

Notice that the solution of 11 are the function $f(x)$ which minimise the hamiltonian

$$H = \int [\frac{1}{2}\frac{\partial f}{\partial x}^2 + W(f)]dx \tag{12}$$

where $V = \frac{dW}{df}$. This implies that to find these solutions, one can add a friction term to 10

$$\frac{\partial^2 f}{\partial t^2} = \frac{\partial^2 f}{\partial x^2} + V(f) - k\frac{\partial f}{\partial t}, \tag{13}$$

where k is a friction parameter, and determine the solution $f(x)$ by solving the time evolution of 13 for a reasonable, guessed, initial condition for f. The friction term will decrease the energy of the system until it settles to a static solution.

This method works, but it is not very efficient. Moreover, the speed of such dynamical relaxation depends crucially on the value of k.

An alternative is to solve the first order time dependant equation

$$\frac{\partial f}{\partial t} = \frac{\partial^2 f}{\partial x^2} + V(f) \tag{14}$$

which does not contain any parameter. This equation can be seen as the heat equation with a particular diffusion term $V(f)$ and it is well know that the time evolution dissipates the heat until the temperature reaches a static configuration dictated by the boundary conditions.

As we are only interested in the asymptotic time independent solution, we are not bothered about the quality of the time evolution and we can thus use the Euler method for it. After discretisation, the equation becomes

$$\frac{f_{[i,n+1]} - f_{[i,n]}}{dt} = \frac{f_{[i+1,n]} + f_{[i-1,n]} - 2f_{[i,n]}}{dx^2} + V(f_[i,n]) \tag{15}$$

where i is the spacial index while n is the time index. We can then rewrite it as

$$f_{[i,n+1]} = f_{[i,n]} + dt * \frac{f_{[i+1,n]} + f_{[i-1,n]} - 2f_{[i,n]}}{dx^2} + V(f_[i,n]). \tag{16}$$

and we should try to take dt as large as possible as the quality of the time integration is not relevant here for as long as it leads to the static solution. It is a well know fact that the Euler method is stable only if we have

$$dt < \frac{dx^2}{2} \tag{17}$$

so we can take $dt = omega\frac{dx^2}{2}$ where we expect $omega$ to be a parameter smaller than 1. In fact the method works if $omega < 2$, and the optimum value depends on the coarseness of the grid. For grid with over 200 points, $omega - 1.9$ works usually very well.

Another important point is that if we used the Euler method to integrate 15, we must use a separate array to save the values of the fields $f_{[i,n+1]}$, otherwise as the program loops over i, the right hand side of 15 would use $f_{[i-1,n+1]}$ instead of $f_{[i-1,n]}$ to compute $f_{[i,n+1]}$. The fact is that here using $f_{[i-1,n+1]}$ not only works but it works even better. This means that the relaxation method can be written as a simple double loop as follow

```
for(n=0; n < n_max; n++)
{ for(i=0; i < nx_; i++)
  { f[i] += omega*dx*dx/2*Force(i);
  }
}
```

where `Force(i)` can be defined as

```
double Force(i)
{ return((f[i+1] +f[i-1] - 2*f[i])/(dx*dx) + V(f[i]));
}
```

5.1 Implementing the relaxation method.

From a programming point of view, the relaxation method is very similar to the time integration of a partial differential equation, so it is natural to implement it by deriving a class from `pde_1d`. We will call that class `pde_1d_relax` and, apart from its constructor, it will have two new methods : `relax_1step` and `relax_grf_until`. `relax_1step` will perform one relaxation step while `relax_grf_until` will perform several relaxation steps in successions, until a certain precision has been reached. It will also produce graphic files at regular interval.

To use the class, one must derive a new class from `pde_1d_relax` exactly as before, and call the method `relax_grf_until` to perform the relaxation. The source files `pde_1d_relax.cc`, `pde_1d_relax.h` are given together with the example: `sineGordon_relax.cc` in appendix C.

6 Modifying the code.

In this section we describe step by step how to modify the program `sineGordon` to solve another equation. As an example, we will solve the radial heat equation

$$\frac{\partial f}{\partial t} = \frac{\partial^2 f}{\partial r^2} + \frac{d-1}{r}\frac{\partial f}{\partial r} - \frac{l}{r^2} f \tag{18}$$

where d is the number of dimensions and l is an angular anisotropy parameter. We also use the following initial condition:

$$f(x, t=0) = A\exp(-kr^2)r^{\lambda} \tag{19}$$

where A and k are two parameters and $\lambda = l^{1/2}$. We output only the energy density which is given by the expression

$$E = \frac{\partial f}{\partial r}^2 + Lf^2 \tag{20}$$

where L is related to l through an expression that depends on d, and we integrate the energy density with the measure $2\pi r^{d-1}$.

We will also compute the local maxima of the energy density as well as their positions.

The complete source files are given in section 8.

6.1 Modifying the Makefile.

To solve this problem, we make a copy of the file `sineGordon.cc` and call it `radialHeat.cc`. We also add the following two lines to the `Makefile`:

```
radialHeat: radialHeat.o $(OBJ)
        $(CC) -o $@ $(CFLAG) $(INCLUDES) $@.o $(OBJ) -lm
```

and add the name `radialHeat` just after `sineGordon` on the last line.

6.2 Solving another equation.

To solve a different equation, we must modify the file `radialHeat.cc` (a copy of `sineGordon.cc`) as follow:

- Set `NO_EQ` to 1 as we are solving a single PDE.
- Modify `VERSION` to change the program name and the date.
- Modify `Fcts_Names_`. We only have one function and we will call it `f`. We can thus remove the string `df`.
- Add two double parameters, `A` and `l`, and one integer parameters, `d` to the global structure `par`.
- Change the text for the array of string `help`.
- Modify the class `sg_pde` and rename it `radialHeat_pde`.
- Modify the method `radialHeat::Eqn`. In our example all we have to modify is the value of `F` as follow:

```
void Eqn(double *F)
{ if(!fl_ || !fr_)  // Do nothing at the edge of the box
  { for(int i=0; i< NoEq_; i++) F[i] = 0;
    return;
  }
  eval_derivatives();
  F[0] = fxx_[0]+(par.d-1)*fx_[0]/x_-par.l*f_[0]/(x_*x_);
}
```

- Modify the method `pde_1d::Boundary`. We must impose the condition that the derivative of the function f vanishes at the origin when $l = 0$. When $l > 0$ then we must have $f(0) = 0$.

```
void Boundary(double *fct)
{ // df/dr is 0 at the origin if l=0
  if (par.l > 0) fct[0] = 0;
  else           fct[0] = fct[NoEq_];
}
```

- Modify the method `pde_1d::Init` to compute the initial condition:

```
void Init(double x,double *F)
{ F[0] = par.A*exp(-par.k*x*x)*pow(x,sqrt(par.l));
}
```

- In the main program, read the 3 new parameters just after reading the value of `k`:

```
ip.get("A",par.A);
ip.get("l",par.l);
ip.get("d",par.d);
```

- Modify the type of the `pde_1d` class that is instantiated. It is now a `radialHeat_pde` instead of `sg_pde`.

- Set the time increment to dx^2dt_r:

```
pde.set_dt(pde.dx()*pde.dx()*par.dt_r).
```

6.3 Computing other densities.

- Set `NO_DENS` to 1 as we are computing only one density.

- Modify `Densities_Names_`. We only have one density which we call `E`.

- Modify the method `radialHeat::Dens`. As we don't change the boundary condition, we only modify the last few lines of the method:

```
double Dens(double *D)
{ if(!fl_ || !fr_) // Desnisties are 0 at the edge of the box
  { for(int i=0; i < NoDens_; i++) D[i] = 0;
    return(1.0);
  }
  eval_derivatives();
  D[0] = fx_[0]*fx_[0]+par.l*f_[0]*f_[0]/(x_*x_);
  return(2.0*pi*x_);
}
```

Notice that we have taken $L = l$ which is correct when $d = 2$. Moreover in that case, we must take $l = n^2$ where n can be any integer. For higher dimensions, refer to the properties of the spherical harmonics. The measure of integration for the energy density is now $2\pi r$ so we must now return the value `2*pi*x` where `pi` is defined as a global variable and initialised at the beginning of the main function.

6.4 Computing other scalar data.

- Define a new constant `NO_MAX` corresponding to the number of local maxima that we want to track. We set it to 2.

- Set `NO_SCALARS` to twice the value of `NO_MAX`.

- Modify Scalars_Names_. We call the value of the local maxima and their position Max1 , Pos1, Max2 and Pos2.
- Modify the method radialHeat::Scalars to track the NO_MAX largest local maxima.

6.5 Generating other graphic files.

We will now generate the graphic files in a different format. We will just output the data as pairs of values x f(x) without any comment at the top of the file. We will also change the name of the files: each file name will be constructed by appending a dot to the graphic name followed by the index value. We won't use the suffix. To do this we create two new files called grf_pde_plain.h and grf_pde_plain.cc. We derive a new class called grf_pde_plain from the class grf_pde_1d and override the methods write_1_grf and make_file_name. We must also override the constructor, but this is trivial.

To use this new graphic class in the program radialHeat we must still do a few modification:

- In the Makefile, we must add the file grf_pde_plain.o to the list of object files. In other words we append the name grf_pde_plain.o at the end of the definition of OBJ.
- In the file radialHeat.cc we must include the file grf_pde_plain.h instead of grf_pde_1d.h.
- In the file radialHeat.cc we must also instantiate a grf_pde_plain object instead of a grf_pde_1d.

6.6 Running radialHeat.

To run the program radialHeat, one starts by creating a parameter file, let us call it radialHeat.par, looking like this:

```
x_min 0
x_max 10
nx 101
dt_r 0.25
t_max 20
grf_dt 1
scalars_dt 1
grf_name DATA/radHeat
k 1
A 1
l 2
d 2
```

and then run the command

```
% radialHeat radialHeat.par
```

The graphic data files, `radHeat_0.txt`, `radHeat_1.txt` ... `radHeat_20.txt`, will be saved in the directory `DATA` and the scalar file will be called `DATA/radHeat.s`.

7 Problems.

To solve the problems below, all that is required is to make a copy of the file `sineGordon.cc` and modify it accordingly. None of the other files needs to be modified at all.

Don't forget that every class can be derived from and that method can be overridden.

7.1 Other Conserved Quantities.

The sine-Gordon equation has conserved quantities other than the total energy. Modify `sineGordon.cc` to evaluate the following densities and verify that the corresponding charges are conserved

$$d_1 = 1 - \cos(f) \tag{21}$$
$$d_2 = f_x^2 \tag{22}$$
$$d_3 = \frac{1}{4} f_x^4 - f_{xx}^2. \tag{23}$$

7.2 One soliton.

Modify the program `sineGordon.cc` to make it use an initial condition corresponding to the single sine-Gordon soliton given by the following expression:

$$f(x,t) = 4 \arctan\left(C \exp\left[(x - vt)/(1 - v^2)^{1/2} \right] \right). \tag{24}$$

The constant C can be set to one and $0 \leq v < 1$ is the soliton speed.

Evaluate $f(x, 0)$ and $\frac{\partial f}{\partial t}(x, 0)$ and use this as the initial condition for f and its first derivative.

Run the program and verify that the soliton travels at a constant speed.

Note: make a copy of the file `sineGordon.cc` and rename it `sineGordon_1soliton.cc` and modify that file. You will have to add an entry to the Makefile to be able to compile it. Do not modify any of the other files.

7.3 The Breather.

The breather solution, centred at the origin, for the sine-Gordon equation is given by the expression

$$f(x,t) = 4 \arctan\left(\frac{(1 - v^2)^{1/2}}{v} \frac{\sin(vt)}{\cosh((1 - v^2)^{1/2} x)} \right). \tag{25}$$

Use this expression for the initial condition.

Note: make a copy of the file `sineGordon.cc`, rename is `sineGordon_breather.cc` and modify that file. You will have to add an entry in the Makefile to be able to compile it. Do not modify any of the other files.

7.4 Two solitons.

The simplest way to make two solitons scatter, is to put 2 single solitons sufficiently apart so that their overlap is very small, and to send them toward each other. So if we define

$$s(x,t;x_0,v) = 4\arctan\left(\exp\left[(x-x_0-vt)/(1-v^2)^{1/2}\right]\right), \tag{26}$$

the initial condition is given by

$$f(x,t) = s(x,t;-L,v) + s(x,t;L,-v). \tag{27}$$

After solving this problem numerically answer the following question: when two such solitons scatter, do they just go through each other or do they bounce? Hint: look at the energy density plots.

It is also interesting to plot the position of the soliton as a function of time. This is done using a scalar quantity. The position of a soliton can be defined either as the local maximum of the energy density, or as the point where $f = \pi \, modulo \, 2\pi$. The two definitions give very similar results.

7.5 The Korteweg-Devries equation.

Like the sine-Gordon equation, the Korteweg-Devries equations,

$$u_t + 6uu_x + u_{xxx} = 0, \tag{28}$$

is integrable and admit soliton solutions. The single soliton solution is given by

$$u = \frac{1}{2}c\cosh^{-2}\left(\frac{1}{2}c^{1/2}(x-ct)\right) \tag{29}$$

where $c > 0$ is the speed of the soliton.

Write a program say `kdv.cc` based on `sineGordon.cc` to solve equation (28). First choose the soliton (29) as the initial condition. Then modify the program to put two such solitons with different values of c away from each other and make them scatter.

Notice that the KdV equation if a first order equation in t. On the other hand, it is of 3rd order in x. You must thus override the `eval_derivatives()` method to compute the third derivatives as well and use a 5 points expression that uses `f_`, `fl_`, `fr_`, `fr_` and `frr_`.

Compute the following densities.

$$D_1 = u \tag{30}$$
$$D_2 = u^2 \tag{31}$$
$$D_3 = u^3 + \frac{1}{2}u_x^2 \tag{32}$$
$$D_4 = 5u^4 + 10uu_x^2 + u_{xx}^2 \tag{33}$$
$$D_5 = 21u^5 + 105u^2u_x^2 + 21uu_{xx}^2 + u_{xxx}^2 \tag{34}$$

Their integration over the real line are conserved quantities of the KdV equation [3].

7.6 Periodic boundary conditions.

Modify `kdv.cc` to make it solve the KdV equation using periodic boundary conditions. For this you will have to derive the class `kdv_pde_1d` from the class `pde_1d_RK4_periodic` instead of `pde_1d_RK4`. It is also advisable to evaluate the initial condition as a superposition of the soliton in the box plus its two nearest neighbour images. This is necessary to make the initial condition smoother at the edge of the lattice.

7.7 Radial two-dimensional Equation.

The d-dimensional sine-Gordon equation

$$\partial^\mu \partial_\mu f + \sin f = 0 \tag{35}$$

does not have soliton solutions when the number of spacial dimensions is greater than 1. Nevertheless, it does have a breather like solutions in 2 and 3 spacial dimension[5]. These solutions are of the form $f = f(t, r)$ and they satisfy the following equation:

$$\frac{\partial^2 f}{\partial t^2} - \frac{\partial^2 f}{\partial r^2} - \frac{d-1}{r}\frac{\partial f}{\partial r} + \sin f = 0. \tag{36}$$

Write a program say `radSineGordon.cc` based on `sineGordon.cc` to solve equation (36) on the interval $[0, R]$ for R sufficiently large. The boundary condition for this equation are $\frac{\partial f}{\partial r} = 0$ at the origin and $f = 0$ at infinity.

A good initial condition for the 2 dimensional breather is given by the following expression

$$f(r, 0) = 4\arctan(C\exp(-\frac{2r}{K\pi}\arctan(r/K))) \tag{37}$$

with $K = 10^{1/2}$ and $C = \tan(\pi/8)$.

7.8 Static solutions.

One can compute the solution of boundary value ODEs by solving a PDE. If we want to compute numerically the soliton solution of the sine-Gordon equation, we can solve the equation

$$\frac{\partial f}{\partial t} - \frac{\partial^2 f}{\partial x^2} + \sin f = 0. \tag{38}$$

by starting from any initial condition such that $f(xmin) = 0$ and $f(xmax) = 2\pi$. One can prove that the profile will relax to the static solution. One can actually use the Euler method in this case as the accuracy of the time integration does not matter, only the final result counts. Unfortunately, this technique is very slow. Nevertheless one can use the so called relaxation method or the Gauss-Seidel relaxation methods [1] which are crudely based on this idea but which are much faster.

Derive a subclass from `pde_1d` that implements one of the two relaxation method and use it to determine numerically the static solution of the sine-Gordon equation.

7.9 Generating Surface Plots.

Derive a new class from `grf_pde_1d` to produce a surface plot of the time evolution of the solution. Instead of outputting every graph in a separate file at regular interval, the data will be appended to a single file, that is one file for each function and density. The data must be in the following format:

```
x0 t0 f0
x1 t0 f1
...
x(nx-1) t0 f(n-1)

x0 t1 f0
x1 t1 f1
...
x(nx-1) t1 f(n-1)


...
x0 tm f0
x1 tm f1
...
x(nx-1) tm f(n-1)
```

Notice that each set of data for a given value of `t` must be followed by a blank line.

To do this the method `write_1_grf` must be overridden. The file name must be constructed without the index number as follow: `file_name_+"_"+name+".txt"`. The new method must open the file to append data to it each time it is called.

7.10 More Scattering.

Having solved the sine-Gordon or the KdV equations for various simple configurations, try more adventurous experiments. How does a sine-Gordon soliton scatter with a breather? Can one generate travelling waves for the sine-Gordon solutions and if so how do they interact with the solitons.

What happens if you take another type of kink (sine-Gordon) or lump (KdV) for the initial solution. Do you still get some solitons?

7.11 Generating Movies Files.

Following the previous problem, derive a new class that produces `ppm` files using the `Magick++` library. The files must be suffixed by a 3 or for digit index so that the are ordered alphabetically in the correct order for the movie: `grf_001.xpm, grf_002.xpm, ... grf_123.xpm ....`

You can then use the command `convert` to create a `mng` movie file from the `ppm` files created by your program.

7.12 Better Design.

The design of our classes could be improved. One objection one might have is that the number of functions, densities and scalars are not checked for consistency by the compiler. To solve that problem one must modify the `pde_1d` and `grf_pde_1d` classes and convert them to templates with parameters corresponding to `NO_EQ` `NO_DENS` and `NO_SCALARS`.

The main problem with that approach is that the whole library must be compiled each time the program is compiled.

A `sineGordon` sources.

A.1 `pde_1d.h`.

The class `pde_1d` is a virtual class used to solve a system of PDEs. Its main methods are:

`pde_1d(...)` : the constructor for the class. It argument are:

- double xmin : the lower bound for the lattice range.
- double xmax : the upper bound for the lattice range.
- int NoEq : the number of differential equations/functions.
- int NoDens : the number of computed densities.
- int NoScalars : the number of scalar data.
- int nx : the number of lattice points.
- string file_name : the prefix for the output file-name.

- char const *Fct_Names[] : a constant array of functions names. The names are used to create a filename for each function.
- char const *Density_Names[] : a constant array of densities names. The names are used to create a filename for each function as well as to create the scalar data header.
- char const *Scalars_Names[] : a constant array of scalar data names. The names are used to create the scalar data header.

`init_grf(grf_pde_1d *grfp)` : this method is used to pass a `grf_pde_1d` pointer to the `pde_1d` to create graphic files.

`void set_dt(double dt)` : set the integration time step.

`void Eqn(double *F)` : a method that must be overridden for every specific `pde_1d`. It evaluates the left hand side of the equation and stores the results in the vector `F` of size `NoEq`. The equation is evaluated by using the variables The method can use the following members which are guaranteed to be correctly initialised:

- `f_`: the value of the function at the local point.
- `fl_`: the value of the function one point to the left.
- `fll_`: the value of the function two points to the left.
- `fr_`: the value of the function one point to the right.
- `frr_`: the value of the function two points to the right.
- `x_`: the value of `x` at the local point.
- `i_`: the index value of the local point. One should try to avoid using it.

These can be used to evaluate the derivatives but one should rather call the method `eval_derivatives()` instead.

`double Dens(double *D)` : like `Eqn` this method is overridden to compute the `NoDens` densities. The method returns the measure of integration to integrate the densities over the lattice range.

`void Boundary(double *fct)` : this method is overridden to implement some specific boundary condition. The parameter `fct` points to the leftmost first function value.

`Init(double x,double *F)` : this method is overridden to evaluate the initial values for the functions.

`Scalars()` : this method is overridden to compute `NoScalar` scalar densities.

`void eval_derivatives()` : This method evaluates the first and the second derivatives of every function. It can be overridden to compute higher order derivatives or only compute the derivatives that are required.

The class `pde_1d_Euler` is derived from the class `pde_1d`. It overrides the method `integrate_1step()` to implement the Euler method.

The class `pde_1d_RK4` is derived from the class `pde_1d`. It overrides the method `integrate_1step()` to implement the 4th order Runge-Kutta method.

```
#ifndef PDE_1D_H
#define PDE_1D_H

#include <cmath>
#include <time.h>
#include <cstring>
#include <cstdio>
#include <string>

class grf_pde_1d;

class pde_1d {
public:
  pde_1d(double xmin,double xmax,int NoEq,int NoDens,int NoScalars,int nx,
         std::string file_name,char const *Fct_Names[],
         char const *Density_Names[],char const *Scalars_Names[]);
  virtual ~pde_1d();
  void init_grf(grf_pde_1d *grfp) { grfp_ = grfp; };

  // Method to Override for specific equation
  virtual void Eqn(double *F)=0;           // Equations
  virtual double Dens(double *D)=0;        // Density function
  virtual void Boundary(double *fct) {};   // Boundary function
  virtual void Init(double x,double *F)=0; // Initial condition function
  virtual void Scalars() {};               // Compute scalars

  void set_dt(double dt) { dt_ = dt;}

  // Get some parameters
  double dx()   { return(dx_);}
  double xmin() { return(Xmin_);}
  double xmax() { return(Xmax_);}
  int nx()      { return(nx_);}

  double *fcts()      { return(Fcts_);}
  double *densities() { return(Densities_);}
  double *intDens()   { return(IntDens_);}
  double *scalars()   { return(Scalars_);}

  /************* METHODS ****************/
  virtual void   initial_value();
```

```
  virtual void   usefulVars();
  virtual void   computeDensities();
  virtual void   set_neighbour_fcts(double *f,int i);

  /************* OUTPUT DATA ****************/
  void write_header_scalars();
  void write_scalars();
  void write_grf();

  /************* INTEGRATION ****************/
  virtual double integrate_1step() {return(0.0);};
  virtual double integrate_by_steps(int n);
  virtual double integrate_until(double t);
  virtual double integrate_grf_by_steps(int n,int scalars_step,int grf_step);
  virtual double integrate_grf_until(double tmax,double scalar_dt,
                                     double grf_dt);

  /************* EVAL DERIVATIVES ****************/
  virtual void eval_derivatives();            // Evaluates fx_ and fxx_

protected:
  double dt_;
  double t_;
  double Xmin_;
  double Xmax_;
  double dx_;

  int NoEq_;
  int NoDens_;
  int NoScalars_;
  int nx_;

  // Useful variables
  double dxs_;       // dx*dx
  double tdx_;       // 2*dx
  int lastPoint_;    // nx_-1;
  int f_array_size_; // NoEq_*nx_

  // allocate arrays for the fcts and densities
  double *Fcts_;       // a vector of size nx* NoEq
  double *Densities_;  // a vector of size nx* NoDensity
  double *IntDens_;    // a vector of size NoDensity
  double *Scalars_;    // a vector of size NoScalars
  double *Force_;      // a vector of size NoEq

  // Allocated arrays for Evaluated quantities
  double *fx_,*fxx_,*fxxx_,*fxxxx_;  // df/fx, d^2f/dx^2
  double *f_,*fl_,*fll_,*fr_,*frr_;  // local fcts ptrs.
  double x_;                 // local value of x = Xmin_+index*dx_
  int i_;                    // local value of grid index

  // Graphic variables
  grf_pde_1d *grfp_;
  int grf_index_;
```

```
  std::string file_name_;

  // Array of names for the Fcts, Densities and Data
  char const **Fct_Names_;     // [NoEq]
  char const **Density_Names_; // [NoDens]
  char const **Scalars_Names_; // [NoScalars]
};

/******************************** Euler ************************************/
class pde_1d_Euler : public pde_1d {

public:
  pde_1d_Euler(double xmin,double xmax,
               int NoEq,int NoDens,int NoScalars,int nx,std::string f_name,
               char const *Fct_Names[],char const *Density_Names[],
               char const *Scalars_Names[]);
  ~pde_1d_Euler();
  double integrate_1step();

protected:
  double *tmp1_;        // same as Fcts
};

/******************************** RK4 *************************************/
class pde_1d_RK4 : public pde_1d {

public:
  pde_1d_RK4(double xmin,double xmax,
             int NoEq,int NoDens,int NoScalars,int nx,std::string f_name,
             char const *Fct_Names[],char const *Density_Names[],
             char const *Scalars_Names[]);
  ~pde_1d_RK4();
  double integrate_1step();
  void sub_RK4_1step(double *f,double *ef,double *mf,double *nef,
                     double Cnef,double Cmf);

protected:
  double *tmp1_;        // same as Fcts
  double *tmp2_;        // same as Fcts
  double *tmp3_;        // same as Fcts
};

/******************************** PERIODIC PDE ***************************/

/******************************** Periodic Euler ***************************/
class pde_1d_Euler_periodic : public pde_1d_Euler {

public:
  pde_1d_Euler_periodic(double xmin,double xmax,
               int NoEq,int NoDens,int NoScalars,int nx,std::string f_name,
               char const *Fct_Names[],char const *Density_Names[],
               char const *Scalars_Names[]);
  virtual void  set_neighbour_fcts(double *f,int i);
};
```

```
/******************************* Periodic RK4 *************************/
class pde_1d_RK4_periodic : public pde_1d_RK4 {

public:
  pde_1d_RK4_periodic(double xmin,double xmax,
             int NoEq,int NoDens,int NoScalars,int nx,std::string f_name,
             char const *Fct_Names[],char const *Density_Names[],
             char const *Scalars_Names[]);
  virtual void  set_neighbour_fcts(double *f,int i);
};
#endif
```

Source file `pde_1d.h`

A.2 `grf_pde_1d.h`.

The class `grf_pde_1d` is a class used to generate functions and densities graphic files. Its main methods are:

`grf_pde_1d(pde_1d *pde,string file_name)` : the first argument is a pointer to a `pde_1d` (or a class derived from it). It is used to read the PDE integration parameters. The second argument is a name used as a prefix for the graphic file-name.

`void write_1_grf(const char *name,int index,double *p,int vsize, double t)` : This method produces a graphic file for a function or a density. It is called by the `pde_1d` integration method.

The argument are the following:

- `name`: the prefix for the output file name. The filename is build as `make_file_name(name,index,"txt")`.
- `index`: the file index. To differentiate between the graphic files produced at different interaction times, the files are characterised by an index that is incremented each time. The first file produced, the initial condition, has an index value of `0`.
- `p`: a pointer to an array of size `vsize`: `p[0]` is the first value, `p[vsize]` is the second ... and `p[vsize*(nx_-1)]` is the last one.
- `vsize`: the size of the density or function array pointed to by `p`.
- `t`: the integration time. It is written as a comment in every file.

`string make_file_name(string name,int index,string suffix)` : This method creates a filename from the prefix `name` the `index` value and the `suffix` as follow: `name+"_"+index+".txt"` . It can be overridden to generate a different filename.

```
#ifndef M_GRF_PDE_1D_H
#define M_GRF_PDE_1D_H
#include <fstream>
#include <string>
#include <pde_1d.h>

class pde_1d;

class grf_pde_1d {
public:
  grf_pde_1d(pde_1d *pde,std::string file_name);

  virtual ~grf_pde_1d() {};

  virtual void write_1_grf(const char *name,int index,double *p,
                                     int vsize,double t);
  virtual std::string make_file_name(std::string name,int index,
                                     std::string suffix);

protected:
  int nx_;
  double xmin_;
  double xmax_;
  double dx_;

  std::string file_name_;
};

/***************************** PERIODIC GRF ************************/
class grf_pde_1d_periodic : grf_pde_1d {
public:
  grf_pde_1d_periodic(pde_1d *pde,std::string file_name) :
               grf_pde_1d(pde,file_name) {};

  virtual void write_1_grf(const char *name,int index,double *p,
                                    int vsize,double t);

};

#endif
```

Source file `grf_pde_1d.h`

A.3 `input_par.h`.

This class is used to read a list of parameters from a file. The parameters must be listed in the file as a parameter names, without blanks, followed by the corresponding value.

`input_par(char *name)` : the constructor takes one parameter: the name of the file to read.

`void get(char *name,double &v)` : reads the next parameter from the parameter file. Check that the parameter name in the file matches `name` and write the corresponding value into `v`, a reference to a `double`.

`void get(char *name,int &v)` : reads the next parameter from the parameter file. Check that the parameter name in the file matches `name` and write the corresponding value into `v`, a reference to an `int`.

`void get(char *name,char *v)` : reads the next parameter from the parameter file. Check that the parameter name in the file matches `name` and write the corresponding string into `v`, a character buffer. The string value cannot contain white spaces.

```
#ifndef INPUT_PAR_H
#define INPUT_PAR_H
#include <fstream>
#include <string>

class input_par
{
public:
  input_par(char *name);
  void get(char *,double &v);
  void get(char *,int &v);
  void get(char *,char *v);

private:
  void check_par_name(char *Name);

  std::string file_name_;
  int line_no_;
  std::ifstream par_file_;
};
#endif
```

Source file `input_par.h`

A.4 `sineGordon.cc`.

At the beginning of the files, we include a few header files. Then we define 3 constants corresponding to the number of functions, densities and scalar data respectively. As the sine-Gordon equation is a second order equation, there are 2 first order equations to solve. We compute 3 densities, the kinetic, potential and total energies. We also compute two scalar data: the maximum value of `f` and the position of that maximum.

We then define a constant string called `VERSION`. It is output when the program is called with the argument `-version` or `-v`.

Next we define three arrays of constant string. They correspond to the names of the functions, densities and scalar data. The names are used to output the graphic data in files named after what they correspond to and to create a column header for the scalar data file.

The structure that follows collects all the program parameters together in a global structure called `par`.

The array of constant string `help` is displayed on the screen as a help message when the number of argument is incorrect or when the argument is `-help` or `-h`.

The class `sg_pde` is then derived from the class `pde_1d_RK4` to solve the sine-Gordon equation.

The method `Eqn()` computes the right hand side of the equation. Notice that the equation value is set to `0` at the edge of the lattice to implement the so called fixed boundary condition.

The method `Dens()` computes the three densities and return `1.0` as the integration measure.

The method `Scalars()` finds the maximum value of `f` and determines its position.

The method `Init()` compute the initial condition: a Gaussian lump centred in the middle of the lattice.

Then we define the global function `Help` which displays the array of string `help` on the screen.

The integration of the equation then take place in the main function. Notice that the whole function body in included inside a `try` and `catch` statement. All the methods that we use throw a `string` error message on error. We thus catch them and display them on the screen.

The first thing we do is to check that `sineGordon` is called with one argument: the parameter file. If the argument is `-help` or `-h` the help message is displayed on the screen. If the argument is `-version` or `-v` the version string is displayed on the screen.

Then we read the parameters from the file one by one.

After this we crate the `sg_pde` to solve the equation. We set the time increment to be `dx*dt_r` where `dt_r` will typically be 0.25. We then create an instance of a `grf_pde_1d` which we pass to the `sg_pde` using the `init_grf` method.

We can then compute the initial condition using `initial_value()` and integrate the equation up to `t_max` producing scalar data and graphic files at intervals given respectively by `scalars_dt` and `grf_dt`.

```
#include "input_par.h"
#include <string>
#include <pde_1d.h>
#include <grf_pde_1d.h>
#include <iostream>

#define NO_EQ      2
#define NO_DENS    3
#define NO_SCALARS 2
```

```
using namespace std;

char *VERSION = "sineGordon: v1.0; 13/05/03; Author: B. Piette";

char const *Fcts_Names[] =
{ "f", "df", 0
};

char const *Densities_Names[] =
{ "En", "Kin", "Pot", 0
};

char const *Scalars_Names[] =
{ "Max" , "MaxPos", 0
};

struct {
  double x_min;
  double x_max;
  int nx;
  double dt_r;
  double t_max;
  double grf_dt;
  double scalars_dt;
  char grf_name[256];

  double k;
} par;

char *help[] = {
 "sineGordon PAR_FILE: ",
 "sineGordon -h|-help: ouput this help message",
 "sineGordon -v|-version: ouput the program version number.",
 "  Solve the sine-Gordon equation.",
 " PAR_FILE:",
 "   x_min DOUBLE",
 "   x_max DOUBLE",
 "   nx INT",
 "   dt_r DOUBLE",
 "   t_max DOUBLE",
 "   grf_dt DOUBLE",
 "   scalars_dt DOUBLE",
 "   grf_name STRING",
 "   k DOUBLE",
 0
};


/* sine-Gordon Equation */
class sg_pde : public pde_1d_RK4
{
public:
  sg_pde(double xmin,double xmax,
```

```
        int NoEq,int NoDens,int NoScalars,int nx,string f_name) :
  pde_1d_RK4(xmin,xmax,NoEq,NoDens,NoScalars,nx,f_name,
        &Fcts_Names[0],&Densities_Names[0],&Scalars_Names[0])
{
}

/* Evaluate the Equation to solve */
/* functions:                      */
/* f[0] : f : sine Gordond fct    */
/* f[1] : df : df/dt              */
/* F : array for result           */
/**********************************/
void Eqn(double *F)
{
  if(!fl_ || !fr_)  // Do nothing at the edge of the box
  { for(int i=0; i< NoEq_; i++) F[i] = 0;
    return;
  }

  eval_derivatives();

  F[0] = f_[1];
  F[1] = fxx_[0] -par.k*sin(f_[0]);
}

/* Compute Densities                         */
/* D[0] :En  : Total energy density         */
/* D[1] :Kin : Kinetic energy density       */
/* D[2] :Pot : Potential energy density     */
/*******************************************/
double Dens(double *D)
{ double fx,K,P;
  int i;

  if(!fl_ || !fr_) // Desnisties are 0 at the edge of the box
  { for(i=0; i < NoDens_; i++) D[i] = 0;
    return(1.0);
  }

  fx = (fr_[0]-fl_[0])/tdx_;
  K = 0.5*f_[1]*f_[1];
  P = 0.5*fx*fx+ par.k*(1.0-cos(f_[0]));
  D[0] = K+P;
  D[1] = K;
  D[2] = P;

  return(1.0);
}

/* Compute the scalar data:       */
/* Max : maximum value of f       */
/* MaxPos : position of maximum   */
/**********************************/
void Scalars()
```

```
  { double fmax = -HUGE_VAL,xmax=Xmin_;

    for(int i = 0; i < nx_; i++)
    { if(Fcts_[i*NoEq_] > fmax)
      { fmax = Fcts_[i*NoEq_];
        xmax = Xmin_+i*dx_;
      }
    }
    Scalars_[0] = fmax;
    Scalars_[1] = xmax;
  };

  /* Implement some boundary conditions       */
  /*******************************************/
  void Boundary(double *fct)
  { // Nothing Here
  }

  /* Compute the initial condition. */
  /* Fcts_[i*NO_EQ]   : f           */
  /* Fcts_[i*NO_EQ+1] : df          */
  /*********************************/
  void Init(double x,double *F)
  { double xm;

    xm = (Xmin_+Xmax_)*0.5;

    F[0] = exp(-(x-xm)*(x-xm));
    F[1] = 0.0;
  }
};

/*****************************************/
/* Ouput a help message on cerr and exit */
/*****************************************/
void Help()
{ int i=0;
  while(help[i]) cerr << help[i++] << std::endl;
  exit(1);
}

int main(int argc, char **argv)
{ grf_pde_1d *grfp;

  try
  { // One argument expected.
    if((argc != 2) || !strcmp(argv[1],"-help") || !strcmp(argv[1],"-h"))
    { Help();
    }
    if(!strcmp(argv[1],"-version") || !strcmp(argv[1],"-v"))
    { cerr << VERSION<<"\n";
      exit(0);
    }
```

```
        // Read the parameters for the input file
        // The parameter filename is the 1st argument
    input_par ip(argv[1]);

    ip.get("x_min",par.x_min);
    ip.get("x_max",par.x_max);
    ip.get("nx",par.nx);
    ip.get("dt_r",par.dt_r);
    ip.get("t_max",par.t_max);
    ip.get("grf_dt",par.grf_dt);
    ip.get("scalars_dt",par.scalars_dt);
    ip.get("grf_name",par.grf_name);

    ip.get("k",par.k);

          // Build a pde for sineGordon
    sg_pde pde(par.x_min,par.x_max,NO_EQ,NO_DENS,NO_SCALARS,par.nx,
              par.grf_name);

          // Set dt by hand : dx * par.dt_r
    pde.set_dt(pde.dx()*par.dt_r);

          // Create a new graphic object
    grfp = new grf_pde_1d(&pde,par.grf_name);
    pde.init_grf(grfp);

    pde.initial_value();

    pde.integrate_grf_until(par.t_max,par.scalars_dt,par.grf_dt);
  }
  catch(string e)
  { cerr << e << std::endl;
  }
}
```

Source file `sineGordon.cc`

A.5 pde_1d.cc.

```
#include <pde_1d.h>
#include <grf_pde_1d.h>
#include <string>
#include <iostream>
#include <fstream>
#include <cstdio>
#include <cmath>

using namespace std;

/*************************************************************************/
/* Constructor :                                                         */
```

```
/* size : initial array size                                            */
/* xmin,xmax : grid range                                               */
/* NoEq,NoDens,NoScalars : No of equations, densities and scalars       */
/* file_name : file name for the data files.                            */
/* nx : No of grid points                                               */
/* Fct_Names: const array of names for the fcts                         */
/* Density_Names: const array of names for the densities                */
/* Fct_Names[] : array of str: name of each fct (do not delete!)        */
/* Density_Names[] : array of str: name of each density (do not delete!)  */
/* Scalars_Names[] : array of str: name of each Scalar (do not delete!)   */
/* Allocates all the fcts and densities arrays.                         */
/* gtype : type of grid: GRID_1D_REGULAR or GRID_1D_PERIODIC            */
/* throw string on error                                                */
/************************************************************************/
pde_1d::pde_1d(double xmin,double xmax,int NoEq,int NoDens,int NoScalars,
               int nx,string file_name,char const *Fct_Names[],
               char const *Density_Names[],char const *Scalars_Names[]) :
  t_(0.0),Xmin_(xmin),Xmax_(xmax),NoEq_(NoEq),NoDens_(NoDens),
  NoScalars_(NoScalars),nx_(nx),file_name_(file_name),
  Fct_Names_(Fct_Names),
  Density_Names_(Density_Names), Scalars_Names_(Scalars_Names)
{
  dx_ = (Xmax_-Xmin_)/(nx_-1);
  dt_ = dx_*0.25;
  grf_index_ = 0;


  usefulVars();

  try
  { Fcts_     = new double[nx_*NoEq_];
    Densities_  = new double[nx_*NoDens_];
    IntDens_    = new double[NoDens_];
    Scalars_    = new double[NoScalars_];
    Force_      = new double[NoEq_];
    fx_         = new double[NoEq_];
    fxx_        = new double[NoEq_];
    fxxx_       = new double[NoEq_];
    fxxxx_      = new double[NoEq_];
  }
  catch(...) { throw((string)"No more memory!"); }
}

/**************/
/* Destructor */
/**************/
pde_1d::~pde_1d()
{ delete [] Fcts_;
  delete [] Densities_;
  delete [] IntDens_;
  delete [] Scalars_;
  delete [] Force_;
  delete [] fx_;
  delete [] fxx_;
```

```
  delete [] fxxx_;
  delete [] fxxxx_;
}

/************************ Evaluates various quantities  ****************/
/***********************************/
/* Initialise the Fcts_            */
/* Calls Init() for every value x_i */
/***********************************/
void pde_1d::initial_value()
{ int i;
  double x;

  x = Xmin_;

  for(i=0; i < nx_; i++)
  { Init(x,&Fcts_[i*NoEq_]);
     x += dx_;
  }
}

/****************************/
/* Eval the useful variables */
/****************************/
void pde_1d::usefulVars()
{ dxs_ = dx_*dx_;
  tdx_ = 2.0*dx_;
  lastPoint_ = nx_-1;
  f_array_size_ = NoEq_*nx_;
}

/***********************************************/
/* Eval the fcts derivatives                   */
/* On the edge of the grid use the inside points */
/* Uses the following variables:               */
/* f_, fl_, fr_ and tdx and dxs.               */
/* fl_ and fr_ can be null                     */
/***********************************************/
void pde_1d::eval_derivatives()
{ int i;

  if(!fl_)
  { for(i=0; i< NoEq_; i++)
    { fx_[i] = (fr_[i]-f_[i])/dx_;
      fxx_[i] = (f_[i]+fr_[NoEq_+i]-2*fr_[i])/dxs_;
    }
    return;
  }
  if(!fr_)
  { for(i=0; i< NoEq_; i++)
    { fx_[i] = (f_[i]-fl_[i])/dx_;
      fxx_[i] = (fl_[-NoEq_+i]+f_[i]-2*fl_[i])/dxs_;
    }
    return;
```

```
  }

  for(i=0; i< NoEq_; i++)
  { fx_[i] = (fr_[i]-fl_[i])/tdx_;
    fxx_[i] = (fl_[i]+fr_[i]-2*f_[i])/dxs_;
  }
}

/************************************************************/
/* Computes the densities and store them in Densities_     */
/* Dens(Vec) : a function which computes all               */
/*     the densities.                                      */
/*       use f_,fl_,fr_ : pointers to the vector of fcts   */
/*       Vec : vector where to store the results.          */
/*       i_  : lattice index                               */
/*     return the integration mesure.                      */
/************************************************************/
void pde_1d::computeDensities()
{ int i,d;
  double mesure;

     /* Initialise the Integration variables to 0 */
  for(d = 0; d < NoDens_; d++)
  { IntDens_[d] = 0.0;
  }

        /* Scan the whole grid, including the first and the last point */
  for(i=0; i <= lastPoint_; i++)
  {     /* fct_vec : the fct at the lattice point i */
    set_neighbour_fcts(Fcts_,i);

        /* Computes the densities and store them in pde->Densities */
    mesure = dx_*Dens(&Densities_[i*NoDens_]);

        /* Integrate each density */
    for(d = 0; d < NoDens_; d++)
    { IntDens_[d] += mesure*Densities_[i*NoDens_+d];
    }
  }
}

/*******************************************/
/* Set the pointers to the neighbour fcts  */
/* f : fct array.                          */
/* i : index position of the grid          */
/*******************************************/
void pde_1d::set_neighbour_fcts(double *f,int i)
{
  x_ = Xmin_+i*dx_;
  i_ = i;

  f_   = &f[i*NoEq_];
  fl_  = &f[(i-1)*NoEq_];
  fll_ = &f[(i-2)*NoEq_];
```

```
  fr_  = &f[(i+1)*NoEq_];
  frr_ = &f[(i+2)*NoEq_];

  if(i==0)                  { fl_ =  0; fll_ = 0; }
  else if(i==1)             { fll_ = 0; }
  else if(i==(lastPoint_-1)){ frr_ = 0; }
  else if(i==lastPoint_)    { fr_ =  0; frr_ = 0; }
}

/************************ Data files  ***************************/
/*******************************************************/
/* Write a header describing the scalars columns:        */
/* # intDens1  intDens2 ...  scalars1 ...                */
/* Append data to the existing file                      */
/* scalars_name : name of the ouput file                 */
/* throw m_except on error                               */
/*******************************************************/
void pde_1d::write_header_scalars()
{ int i;
  string scalars_name = file_name_+".s";
  ofstream ofs(scalars_name.c_str());

  if(!ofs)
  { throw((string)" Error: Can't open file : "+scalars_name);
  }

  // Write Fcts and Densities titles as comments
  ofs <<"# t";
  for(i = 0; i < NoDens_; i++)
  { ofs << " "<<Density_Names_[i];
  }
  for(i = 0; i < NoScalars_; i++)
  { ofs << " "<<Scalars_Names_[i];
  }
  ofs <<"\n";

}
/*********************************************************/
/* Write the integrated densities and the scalars in a file */
/* Append data to the existing file                         */
/* scalars_name : name of the ouput file                    */
/* Output one line of the type:                             */
/* t intDens1  intDens2 ...  scalars1 ...                   */
/* throw m_except on error                                  */
/*********************************************************/
void pde_1d::write_scalars()
{ int i;
  string scalars_name = file_name_+".s";
  ofstream ofs(scalars_name.c_str(),ios::app);

  if(!ofs)
  { throw((string)" Error: Can't open file : "+scalars_name);
  }
```

```
  ofs <<t_;

  // Write Integrated densities:
  for(i=0; i < NoDens_; i++)
  { ofs <<" "<< intDens()[i];
  }

  // Write Scalar data
  for(i=0; i < NoScalars_; i++)
  { ofs <<" "<< scalars()[i];
  }
  ofs <<"\n";
}

/**************************************************************/
/* Create a graphic file for the selected fcts and densities   */
/* Throw a string on error                                     */
/**************************************************************/
void pde_1d::write_grf()
{ int i;

  // Write fcts graphs
  for(i=0; i < NoEq_; i++)
  { grfp_->write_1_grf(Fct_Names_[i],grf_index_,&Fcts_[i],NoEq_,t_);
  }

  // Write Density graphs
  for(i=0; i < NoDens_; i++)
  { grfp_->write_1_grf(Density_Names_[i],grf_index_,&Densities_[i],NoDens_,t_);
  }

  grf_index_++;
}

/************************ Integration ***************************/
/********************************************************/
/* Perform n integration steps on the grid             */
/* Update the time t_                                  */
/* pde : pde parameters                                */
/* Return : the updated time                           */
/********************************************************/
double pde_1d::integrate_by_steps(int n)
{ while(n--)
  { integrate_1step();
  }
  return(t_);
}

/********************************************************/
/* Perform n integration steps on the grid             */
/* Update the time t_                                  */
/* pde : pde parameters                                */
/* Return : the updated time                           */
/********************************************************/
```

```
double pde_1d::integrate_until(double t)
{
  while(integrate_1step() < t);
  return(t_);
}

/*************************************************************/
/* Perform n integration steps on the grid with graphics       */
/* Output scalars every scalars_step                           */
/* Output graphs every grf_step                                */
/* Update the time t_                                          */
/* n : No of integration steps                                 */
/* scalars_step : no of steps between scalars outputs          */
/* grf_step : no of steps between graphics                     */
/* Return : the updated time                                   */
/*************************************************************/
double pde_1d::integrate_grf_by_steps(int n,int scalars_step,int grf_step)
{ int i;
  string grf_name;

  write_header_scalars();

  i = 0;
  while(i < n)
  { if(((i%scalars_step) == 0) || ((i%grf_step) == 0))
    { computeDensities();
      Scalars();
      if((i%grf_step) == 0)   write_grf();
      if((i%scalars_step) == 0) write_scalars();
    }
    integrate_1step();
    i++;
  }

  if(((i%scalars_step) == 0) || ((i%grf_step) == 0))
  { computeDensities();
    Scalars();
    if((i%grf_step) == 0)   write_grf();
    if((i%scalars_step) == 0) write_scalars();
  }
  return(t_);
}

/*************************************************************/
/* Perform n integration step on the grid with graphics        */
/* Output scalars every scalars_dt                             */
/* Output graphs every grf_dt                                  */
/* Update the time t_                                          */
/* tmax : integration bound.                                   */
/* scalars_dt : time interval between scalars outputs          */
/* grf_dt : time interval between graphics                     */
/* Return : the updated time                                   */
/*************************************************************/
double pde_1d::integrate_grf_until(double tmax, double scalar_dt,double grf_dt)
```

```
{ int i;
  string grf_name;
  double scalar_t,grf_t;
  write_header_scalars();

  scalar_t = t_ - dt_*0.01;
  grf_t = t_ - dt_*0.01;

  while(t_ < tmax)
  { if((t_ >= scalar_t) || (t_ > grf_t))
    { computeDensities();
      Scalars();
      if(t_ >= grf_t)   { write_grf(); grf_t += grf_dt; }
      if(t_ >= scalar_t){ write_scalars(); scalar_t += scalar_dt; }
    }
    integrate_1step();
    i++;
  }

  if((t_ > scalar_t) || (t_ > grf_t))
  { computeDensities();
    Scalars();
    if((t_ >= grf_t) == 0)  write_grf();
    if((t_ >= scalar_t) == 0) write_scalars();
  }

  return(t_);
}

/******************** Euler **********************************/
/********************************/
/* Constructor for Euler pde_1d */
/* Allocated 2 extra arrays     */
/********************************/
pde_1d_Euler::pde_1d_Euler(double xmin,double xmax,int NoEq,int NoDens,
          int NoScalars,int nx,string f_name,char const *Fct_Names[],
          char const *Density_Names[],char const *Scalars_Names[]) :
    pde_1d(xmin,xmax,NoEq,NoDens,NoScalars,nx,f_name,Fct_Names,
             Density_Names,Scalars_Names)
{ tmp1_ = new double[nx_*NoEq_];
}

pde_1d_Euler::~pde_1d_Euler()
{ delete [] tmp1_;
}

/******************************************************/
/* Perform a single euler integration step on the grid    */
/* Update the time t_                                     */
/* pde : pde parameters                                   */
/* Eqn(Force) : computes the right hand side of           */
/*     the equation.                                      */
/*       uses f_,fl_,fr_: local fct pointer               */
/*       Force : vector where to store the result.        */
```

```
/*       i_    : lattice index                            */
/* Return : the updated time                              */
/*********************************************************/
double pde_1d_Euler::integrate_1step()
{ int i,j,grid_size;

  grid_size = NoEq_*nx_;

      /* Scan the whole grid */
  for(i=0; i <= lastPoint_; i++)
  {     /* fct_vec : the fct at the lattice point i */
    set_neighbour_fcts(Fcts_,i);

        /* Computes the force for point i and  */
        /* stores it in Force                  */
    Eqn(Force_);

    /* Update each fct using Force : f(n+1) = f(n) + Force*dt */
    /* tmp1 is used to store the fct temporarily              */
    for (j = 0; j < NoEq_; j++)
    { tmp1_[i*NoEq_+j] = f_[j]  + dt_*Force_[j];
    }
  }

  /* Update Fct with the new fct */
  for(i=0; i < grid_size; i++)
  { Fcts_[i] = tmp1_[i];
  }

  Boundary(Fcts_);
  t_ += dt_;  /* Udate the time */
  return(t_);
}



/******************** RK4 **********************************/
/*****************************/
/* Constructor for RK4 pde_1d */
/* Allocated 2 extra arrays   */
/*****************************/
pde_1d_RK4::pde_1d_RK4(double xmin,double xmax,int NoEq,int NoDens,
          int NoScalars,int nx,string f_name,char const *Fct_Names[],
          char const *Density_Names[],char const *Scalars_Names[]) :
       pde_1d(xmin,xmax,NoEq,NoDens,NoScalars,nx,f_name,Fct_Names,
              Density_Names,Scalars_Names)
{ tmp1_ = new double[nx_*NoEq_];
  tmp2_ = new double[nx_*NoEq_];
  tmp3_ = new double[nx_*NoEq_];
}

pde_1d_RK4::~pde_1d_RK4()
{ delete [] tmp1_;
  delete [] tmp2_;
```

```
  delete [] tmp3_;
}

/*******************************************************/
/* Perform a 4th order Runge-Kutta integration step on    */
/* f(n+1) = fn + 1/6(k1 +2 k2 + 2 k3 + k4)               */
/* k1 = dt F(t,fn)                                        */
/* k2 = dt F(t+dt/2,fn+k1/2)                              */
/* k3 = dt F(t+dt/2,fn+k2/2)                              */
/* k4 = dt F(t+dt,fn+k3)                                  */
/* the grid. Update the time in pde.                      */
/* Eqn(Force) : a function which computes the right       */
/*     hand side of the equation.                         */
/*       Force : vector where to store the result.        */
/*       i_    : lattice index                            */
/* Return : the updated time                              */
/*******************************************************/
double pde_1d_RK4::integrate_1step()
{ int i;

    /* tmp1 is used to compute fn + 1/6(k1 + 2 k2 + 2 k3 + k4) */
    /* We initialise it to 0                                    */
  for(i=0; i < f_array_size_; i++)
  { tmp1_[i] = 0;
  }

    /* tmp1 = 0 */
  sub_RK4_1step(Fcts_,Fcts_,tmp1_,tmp2_,0.5,1.0);

    /* tmp1 = k1 */
    /* tmp2 = fn + 1/2 k1 */
  t_ += 0.5*dt_;
  sub_RK4_1step(Fcts_,tmp2_,tmp1_,tmp3_,0.5,2.0);

    /* tmp1 = fn + 1/6 (k1 + 2 k2 ) */
    /* tmp3 = fn + 1/2 k2 */
  sub_RK4_1step(Fcts_,tmp3_,tmp1_,tmp2_,1.0,2.0);

    /* tmp1 = fn + 1/6 (k1 + 2 k2 + 2 k3) */
    /* tmp3 = fn + k3 */
  t_ += 0.5*dt_;
  sub_RK4_1step(Fcts_,tmp2_,tmp1_,0,0.0,1.0);

    /* tmp1 = (k1 + 2 k2 + 2 k3 + k4) */
    /* we update fn => fn + tmp1 / 6  */
  for(i=0; i < f_array_size_; i++)
  { Fcts_[i] += tmp1_[i]/6.0;
  }
  Boundary(Fcts_);

  /* Time has been updated before the last step */
  return(t_);
}
```

```
/**********************************************************/
/* Perform one RK intermediate step                       */
/* f : f(n) the fcts at time tn                           */
/* ef : estimated f for this step.                        */
/* mf : modified f : use add the contribution of each step    */
/* nef : estimated f for the next step, must be computed here */
/* Cnef : coeficiant used to computed nef                 */
/* Cmf  : coeficiant used to computed mf                  */
/**********************************************************/
void pde_1d_RK4::sub_RK4_1step(double *f,double *ef,double *mf,double *nef,
                              double Cnef,double Cmf)
{ int index,i,j;

  for(i=0; i <= lastPoint_; i++)
  {     /* fct_vec : the fct at the lattice point i */

    index = i*NoEq_;

    set_neighbour_fcts(ef,i);

        /* Computes the force for point i and  */
        /* stores it in Force                  */
    Eqn(Force_);

    /* Update each fct using Force : f(n+1) = f(n) + Force*dt */
    for (j = 0; j < NoEq_; j++)
    { mf[index+j] += dt_ * Force_[j] * Cmf;

      if(nef) /* nef will be 0 for the last step */
      { nef[index+j] = f[index+j] + dt_ * Force_[j] * Cnef;
      }
    }
  }

    /* Set the boundary condition for the estimated fct */
  if(nef) Boundary(nef);
}

/******************* PERIODIC PDE ******************************/
/****************************************/
/* Constructor for periodic Euler pde_1d */
/* Allocated 2 extra arrays             */
/****************************************/
pde_1d_Euler_periodic::pde_1d_Euler_periodic(double xmin,double xmax,int NoEq,
          int NoDens,
          int NoScalars,int nx,string f_name,char const *Fct_Names[],
          char const *Density_Names[],char const *Scalars_Names[]) :
    pde_1d_Euler(xmin,xmax,NoEq,NoDens,NoScalars,nx,f_name,Fct_Names,
              Density_Names,Scalars_Names)
{ dx_ = (Xmax_-Xmin_)/nx_;
}

/*****************************************/
/* Set the pointers to the neighbour fcts  */
```

```
/* When using periodic boundray conditions */
/* f : fct array.                           */
/* i : index position of the grid           */
/*******************************************/
void pde_1d_Euler_periodic::set_neighbour_fcts(double *f,int i)
{
  x_ = Xmin_+i*dx_;
  i_ = i;

  f_   = &f[i*NoEq_];
  fl_  = &f[(i-1)*NoEq_];
  fll_ = &f[(i-2)*NoEq_];
  fr_  = &f[(i+1)*NoEq_];
  frr_ = &f[(i+2)*NoEq_];

  if(i==0)            { fl_ = &f[(nx_-1)*NoEq_] ; fll_ = &f[(nx_-2)*NoEq_]; }
  else if(i==1)            { fll_ = &f[(nx_-1)*NoEq_]; }
  else if(i==(lastPoint_-1)){ frr_ = &f[0]; }
  else if(i==lastPoint_)    { fr_ =  &f[0]; frr_ = &f[NoEq_]; }
}

/**************************************/
/* Constructor for periodic RK4 pde_1d */
/* Allocated 2 extra arrays            */
/**************************************/
pde_1d_RK4_periodic::pde_1d_RK4_periodic(double xmin,double xmax,int NoEq,
          int NoDens,int NoScalars,int nx,string f_name,
          char const *Fct_Names[],
          char const *Density_Names[],char const *Scalars_Names[]) :
       pde_1d_RK4(xmin,xmax,NoEq,NoDens,NoScalars,nx,f_name,Fct_Names,
             Density_Names,Scalars_Names)
{ dx_ = (Xmax_-Xmin_)/nx_;
}

/*******************************************/
/* Set the pointers to the neighbour fcts  */
/* When using periodic boundray conditions */
/* f : fct array.                           */
/* i : index position of the grid           */
/*******************************************/
void pde_1d_RK4_periodic::set_neighbour_fcts(double *f,int i)
{
  x_ = Xmin_+i*dx_;
  i_ = i;

  f_   = &f[i*NoEq_];
  fl_  = &f[(i-1)*NoEq_];
  fll_ = &f[(i-2)*NoEq_];
  fr_  = &f[(i+1)*NoEq_];
  frr_ = &f[(i+2)*NoEq_];

  if(i==0)            { fl_ = &f[(nx_-1)*NoEq_] ; fll_ = &f[(nx_-2)*NoEq_]; }
  else if(i==1)            { fll_ = &f[(nx_-1)*NoEq_]; }
  else if(i==(lastPoint_-1)){ frr_ = &f[0]; }
```

```
  else if(i==lastPoint_)    { fr_ =  &f[0]; frr_ = &f[NoEq_]; }
}
```

Source file pde_1d.cc

A.6 grf_pde_1d.c.

```
#include <grf_pde_1d.h>
#include <pde_1d.h>
#include <cstring>
#include <fstream>
#include <iostream>

using namespace std;

/****************** GRF_PDE_1D ****************************************/
/**********************************************************************/
/* Creator for pde_1d graph.                                          */
/* pde : pointer to the m_pde_1d to display (do not free!)            */
/* file_name : filename prefix                                        */
/**********************************************************************/
grf_pde_1d::grf_pde_1d(pde_1d *pde,string file_name) : file_name_(file_name)
{
  nx_    = pde->nx();
  xmin_  = pde->xmin();
  xmax_  = pde->xmax();
  dx_    = pde->dx();
};

/**********************************************************/
/* Ouput a single txt 1D graphic file                     */
/* name : file descriptor (Energy of fct name)            */
/* findex : file index                                    */
/* p : pointer to the data array                          */
/* vsize : size of data vector (No of fcts)               */
/* t : value of t for the snapshot                        */
/* Output format:                                         */
/* The file should be a .m file. It can be execute by matlab  */
/*    # t = t                                             */
/*    ti Fi                                               */
/* Throw a string on error                                */
/**********************************************************/
void grf_pde_1d::write_1_grf(const char *name,int findex,double *p, int vsize,
                                double t)
{ int ix,index;
  double x;
  string fname = make_file_name(file_name_+"_"+name,findex,"txt");

  ofstream ofs(fname.c_str());

  if(!ofs)
  { throw((string)" Error: Can't open file : "+fname);
  }
```

```
  ofs <<"# t = "<<t<<";\n";

  index = 0 ;
  x = xmin_;
  for(ix = 0; ix < nx_; ix += 1)
  { ofs << x << " " << p[index] << "\n";
    index += vsize;
    x += dx_;
  }
}

// Create a file name of the form:
//    NAME_DDD.XXX
// where NAME is the string name
//       DDD is the index
//       XXX is the suffix
// Return the nameas a string
//******************************
string grf_pde_1d::make_file_name(string name,int index,string suffix)
{ char Buff[40];

  sprintf(Buff,"%d",index);

  return(name+"_"+Buff+"."+suffix);
}

/*********************** PERIODIC GRF **************************/
/************************************************************/
/* Ouput a single txt 1D graphic file for a periodic PDE      */
/* The only difference with grf_pde_1d is that the value of   */
/* the point, p{x_min}=p{xmax}, is added at the end           */
/* name : file descriptor (Energy of fct name)                */
/* findex : file index                                        */
/* p : pointer to the data array                              */
/* vsize : size of data vector (No of fcts)                   */
/* t : value of t for the snapshot                            */
/* Output format:                                             */
/* The file should be a .m file. It can be execute by matlab  */
/*    # t = t                                                 */
/*    ti Fi                                                   */
/* Throw a string on error                                    */
/************************************************************/
void grf_pde_1d_periodic::write_1_grf(const char *name,int findex,double *p,
          int vsize,double t)
{ int ix,index;
  double x;
  string fname = make_file_name(file_name_+"_"+name,findex,"txt");

  ofstream ofs(fname.c_str());

  if(!ofs)
  { throw((string)" Error: Can't open file : "+fname);
  }
```

```
  ofs <<"# t = "<<t<<";\n";

  index = 0 ;
  x = xmin_;
  for(ix = 0; ix < nx_; ix += 1)
  { ofs << x << " " << p[index] << "\n";
    index += vsize;
    x += dx_;
  }
  ofs << x << " " << p[0] << "\n";
}
```

Source file grf_pde_1d.cc

A.7 input_par.c.

```
#include "input_par.h"
#include <cstdio>

using namespace std;

/**********************************************/
/* Create input_par: open the file for reading */
/* Throw a string if the file can't be open    */
/**********************************************/
input_par::input_par(char *name) : file_name_(name), line_no_(1)
{ if(!name || (name[0]==0))
  { throw ((string) "ERROR : blank file name for parameter file!");
  }
  par_file_.open(name);
  if(!par_file_.good())
  { throw ((string) "ERROR : Can't open parameter file "+name);
  }
}

/****************************************************/
/* Get the next paramater from the parameter file     */
/* Read one word then compare it to the Name supplied */
/* Read the double Value that follows and put it in v */
/* The line must be of the form:                      */
/*    Name  Value                                     */
/* Throw a string on error                            */
/****************************************************/
void input_par::get(char *Name,double &v)
{
  // Check that the parameter name is correct
  check_par_name(Name);

  // Read the value that follows : an int
  par_file_ >> v;

  // Increment the line no
```

```
  ++line_no_;
}

/****************************************************/
/* Get the next paramater from the parameter file     */
/* Read one word then compare it to the Name supplied */
/* Read the int Value that follows and put it in v    */
/* The line must be of the form:                      */
/*    Name  Value                                     */
/* Throw a string on error                            */
/****************************************************/
void input_par::get(char *Name,int &v)
{
  // Check that the parameter name is correct
  check_par_name(Name);

  // Read the value that follows : a double
  par_file_ >> v;

  // Increment the line no
  ++line_no_;
}

/****************************************************/
/* Get the next paramater from the parameter file     */
/* Read one word then compare it to the Name supplied */
/* Read the string Value that follows and put it in v */
/* No space is allowed in the string value!           */
/* The line must be of the form:                      */
/*    Name  Value                                     */
/* Throw a string on error                            */
/****************************************************/
void input_par::get(char *Name,char *v)
{
  // Check that the parameter name is correct
  check_par_name(Name);

  // Read the value that follows : a string
  par_file_ >> v;

  // Increment the line no
  ++line_no_;
}

/*****************************************************/
/* Check the paramater name:                           */
/* Read one word then compare it to the Name supplied  */
/* The line in the parameter file must be of the form: */
/*    Name  Value                                      */
/* Throw a string if the parameter Name does not match */
/*****************************************************/
void input_par::check_par_name(char *Name)
{ string s;
  char Buff[40];
```

```
  par_file_ >> s;

  // Check that the parameter name is correct
  if(s != Name)
  { sprintf(Buff,"%d",line_no_);
    throw ((string)"Error line "+Buff+" in parameter file "+file_name_+
        "\n  Line "+Buff+" : invalid parameter "+s+"  when expecting "+Name);
  }
}
```

Source file `input_par.cc`

A.8 `Makefile`.

```
CFLAG = -ansi -O2 -Wall
INCLUDES = -I.
CC = g++

OBJ = pde_1d.o pde_1d_relax.o grf_pde_1d.o input_par.o

.cc:
$(CC) -c $< -o $*.o $(CFLAG) $(INCLUDES)
.cc.o:
$(CC) -c $< -o $*.o $(CFLAG) $(INCLUDES)

ALL: sineGordon sineGordon_relax

sineGordon: sineGordon.o $(OBJ)
$(CC) -o $@ $(CFLAG) $(INCLUDES) $@.o $(OBJ) -lm

sineGordon_relax: sineGordon_relax.o $(OBJ)
$(CC) -o $@ $(CFLAG) $(INCLUDES) $@.o $(OBJ) -lm

clean:
/bin/rm -f *.o core.* sineGordon
```

Source file `Makefile`

B The radial heat equation

This section lists the source files for the program `radialHeat`. The creation of these files from the program `sineGordon` is described in detail in section 5.

```
#ifndef M_GRF_PDE_FM_H
#define M_GRF_PDE_FM_H
#include <string>
#include <pde_1d.h>
```

```
#include <grf_pde_1d.h>

class grf_pde_plain : public grf_pde_1d {
public:
  grf_pde_plain(pde_1d *pde,std::string file_name) :
            grf_pde_1d(pde,file_name) {};

  virtual ~grf_pde_plain() {};
  virtual void write_1_grf(const char *name,int findex,double *p,
                           int vsize,double t);
  virtual std::string make_file_name(std::string name,int index,
                                     std::string suffix);
};
#endif
```

Source file grf_pde_plain.h

```
#include <grf_pde_plain.h>
#include <pde_1d.h>
#include <cstring>
#include <fstream>

using namespace std;

/***********************************************************/
/* Ouput a single txt 1D graphic file                      */
/* name : file descriptor (Energy or fct name)             */
/* findex : file index                                     */
/* p : pointer to the data array                           */
/* vsize : size of data vector (No of fcts)                */
/* t : value of t for the snapshot                         */
/* Output format:                                          */
/*    ti Fi                                                */
/* Throw a string on error                                 */
/***********************************************************/
void grf_pde_plain::write_1_grf(const char *name,int findex,double *p,
                                int vsize,double t)
{ int ix,index;
  double x;
  string fname = make_file_name(file_name_+"_"+name,findex,"txt");

  ofstream ofs(fname.c_str());

  if(!ofs)
  { throw((string)" Error: Can't open file : "+fname);
  }

  index = 0 ;
  x = xmin_;
  for(ix = 0; ix < nx_; ix += 1)
  { ofs << x << " " << p[index] << "\n";
    index += vsize;
    x += dx_;
  }
```

```
}

// Create a file name of the form:
//    NAME.DDD
// Return the name as a string
//******************************
string grf_pde_plain::make_file_name(string name,int index,string suffix)
{ char Buff[40];

  sprintf(Buff,"%d",index);

  return(name+"."+Buff);
}
```

Source file `grf_pde_plain.cc`

B.1 `radialHeat.cc`.

```
#include "input_par.h"
#include <string>
#include <pde_1d.h>
#include <grf_pde_plain.h>
#include <iostream>

using namespace std;

const int NO_MAX  = 2;

const int NO_EQ   = 1;
const int NO_DENS = 1;
const int NO_SCALARS = 2*NO_MAX;

char *VERSION = "radialHeat: v1.0; 20/05/03; Author: B. Piette";

char const *Fcts_Names[] =
{ "f", 0
};

char const *Densities_Names[] =
{ "E", 0
};

char const *Scalars_Names[] =
{ "Max1" , "Pos1", "Max2",  "Pos2", 0
};

struct {
  double x_min;
  double x_max;
  int nx;
  double dt_r;
  double t_max;
  double grf_dt;
```

```
  double scalars_dt;
  char grf_name[256];

  double k;
  double A;
  double l;
  int d;
} par;

char *help[] = {
 "radialHeat PAR_FILE: ",
 "radialHeat -h|-help: ouput this help message",
 "radialHeat -v|-version: ouput the program version number.",
 "  Solve the radial heat equation:",
 "   df/dt = d^2f/dr^2 + (d-1)/r df/dr - l/r^2 f",
 " PAR_FILE:",
 "   x_min DOUBLE",
 "   x_max DOUBLE",
 "   nx INT",
 "   dt_r DOUBLE",
 "   t_max DOUBLE",
 "   grf_dt DOUBLE",
 "   scalars_dt DOUBLE",
 "   grf_name STRING",
 "   k DOUBLE",
 "   A DOUBLE",
 "   l DOUBLE",
 "   d INTEGER",
 0
};

double pi;

/* Radial heat equation */
class radialHeat_pde : public pde_1d_RK4
{
public:
  radialHeat_pde(double xmin,double xmax,
          int NoEq,int NoDens,int NoScalars,int nx,string f_name) :
    pde_1d_RK4(xmin,xmax,NoEq,NoDens,NoScalars,nx,f_name,
          &Fcts_Names[0],&Densities_Names[0],&Scalars_Names[0])
  {
  }

  /* Evaluate the Equation to solve */
  /* functions:                      */
  /* f[0] : f : sine Gordond fct     */
  /* f[1] : df : df/dt               */
  /* F : array for result            */
  /**********************************/
  void Eqn(double *F)
  {
    if(!fl_ || !fr_)  // Do nothing at the edge of the box
    { for(int i=0; i< NoEq_; i++) F[i] = 0;
```

```
    return;
  }

  eval_derivatives();

  F[0] = fxx_[0]+(par.d-1)*fx_[0]/x_-par.l*f_[0]/(x_*x_);
}

/* Compute Densities                          */
/* D[0] :E  : Energy density                  */
/********************************************/
double Dens(double *D)
{ if(!fl_ || !fr_) // Desnisties are 0 at the edge of the box
  { for(int i=0; i < NoDens_; i++) D[i] = 0;
    return(1.0);
  }

  eval_derivatives();
  D[0] = fx_[0]*fx_[0]+par.l*f_[0]*f_[0]/(x_*x_);

  return(2.0*pi*x_);
}

/* Compute the scalar data:        */
/* Max : local maximum values of E */
/* MaxPos : position of maximum    */
/**********************************/
void Scalars()
{ double fm[NO_MAX],xm[NO_MAX],dm,x;
  int i,j,m,k = 0;

  for(int i = 1; i < nx_-1; i++)
  { // D[i] > D[i-1] && D[i] > D[i+1] => Local maximum
    if((Densities_[i*NoEq_] > Densities_[(i-1)*NoEq_])&&
       (Densities_[i*NoEq_] > Densities_[(i+1)*NoEq_]))
    { if (k < NO_MAX) { j = k++;}
      else
      { // Find the smalest maxima
        dm = fm[0]; j = 0;
        for(m = 1 ; m< NO_MAX; m++)
        { if(fm[m] < dm) { dm = fm[m]; j = m;}
        }
        if(Densities_[i*NoEq_] < fm[j]) j = -1; // Local max too small
      }
      if(j >= 0)
      { fm[j] = Densities_[i*NoEq_];
        xm[j] = Xmin_+i*dx_;
      }
    }
  }

  // If k < NO_MAX : use edge value:
  if (Densities_[0] > Densities_[nx_-1]) { dm = Densities_[0]; x = Xmin_; }
  else                                   { dm = Densities_[nx_-1]; x = Xmax_;}
```

```
    for(i = k ; i< NO_MAX; i++)
    { fm[i] = dm;
      xm[i] = x;
    }

    for(m = 0 ; m < NO_MAX; m++)
    { Scalars_[2*m]   = fm[m];
      Scalars_[2*m+1] = xm[m];
    }
  };

  /* Implement some boundary conditions       */
  /********************************************/
  void Boundary(double *fct)
  { // df/dr is 0 at the origin if l=0
    if (par.l > 0) fct[0] = 0;
    else           fct[0] = fct[NoEq_];
  }

  /* Compute the initial condition. */
  /* Fcts_[i*NO_EQ]   : f           */
  /* Fcts_[i*NO_EQ+1] : df          */
  /*********************************/
  void Init(double x,double *F)
  { F[0] = par.A*exp(-par.k*x*x)*pow(x,sqrt(par.l));
  }
};

/****************************************/
/* Ouput a help message on cerr and exit */
/****************************************/
void Help()
{ int i=0;
  while(help[i]) cerr << help[i++] << std::endl;
  exit(1);
}

int main(int argc, char **argv)
{ grf_pde_1d *grfp;

  pi = 4*atan(1.0);

  try
  { // One argument expected.
    if((argc != 2) || !strcmp(argv[1],"-help") || !strcmp(argv[1],"-h"))
    { Help();
    }
    if(!strcmp(argv[1],"-version") || !strcmp(argv[1],"-v"))
    { cerr << VERSION<<"\n";
      exit(0);
    }


        // Read the parameters for the input file
```

```
      // The parameter filename is the 1st argument
    input_par ip(argv[1]);

    ip.get("x_min",par.x_min);
    ip.get("x_max",par.x_max);
    ip.get("nx",par.nx);
    ip.get("dt_r",par.dt_r);
    ip.get("t_max",par.t_max);
    ip.get("grf_dt",par.grf_dt);
    ip.get("scalars_dt",par.scalars_dt);
    ip.get("grf_name",par.grf_name);

    ip.get("k",par.k);
    ip.get("A",par.A);
    ip.get("l",par.l);
    ip.get("d",par.d);

          // Build a pde for sineGordon
    radialHeat_pde pde(par.x_min,par.x_max,NO_EQ,NO_DENS,NO_SCALARS,par.nx,
              par.grf_name);

          // Set dt by hand : dx * par.dt_r
    pde.set_dt(pde.dx()*pde.dx()*par.dt_r);

          // Create a new graphic object
    grfp = new grf_pde_plain(&pde,par.grf_name);
    pde.init_grf(grfp);

    pde.initial_value();

    pde.integrate_grf_until(par.t_max,par.scalars_dt,par.grf_dt);
  }
  catch(string e)
  { cerr << e << std::endl;
  }
}
```

Source file `radialHeat.cc`

B.2 grf_pde_plain.h.

```
#ifndef M_GRF_PDE_FM_H
#define M_GRF_PDE_FM_H
#include <string>
#include <pde_1d.h>
#include <grf_pde_1d.h>

class grf_pde_plain : public grf_pde_1d {
public:
  grf_pde_plain(pde_1d *pde,std::string file_name) :
            grf_pde_1d(pde,file_name) {};
```

```
  virtual ~grf_pde_plain() {};
  virtual void write_1_grf(const char *name,int findex,double *p,
                           int vsize,double t);
  virtual std::string make_file_name(std::string name,int index,
                                     std::string suffix);
};
#endif
```

Source file grf_pde_plain.h

B.3 grf_pde_plain.cc.

```
#include <grf_pde_plain.h>
#include <pde_1d.h>
#include <cstring>
#include <fstream>

using namespace std;

/************************************************************/
/* Ouput a single txt 1D graphic file                      */
/* name : file descriptor (Energy or fct name)             */
/* findex : file index                                     */
/* p : pointer to the data array                           */
/* vsize : size of data vector (No of fcts)                */
/* t : value of t for the snapshot                         */
/* Output format:                                          */
/*    ti Fi                                                */
/* Throw a string on error                                 */
/************************************************************/
void grf_pde_plain::write_1_grf(const char *name,int findex,double *p,
                                int vsize,double t)
{ int ix,index;
  double x;
  string fname = make_file_name(file_name_+"_"+name,findex,"txt");

  ofstream ofs(fname.c_str());

  if(!ofs)
  { throw((string)" Error: Can't open file : "+fname);
  }

  index = 0 ;
  x = xmin_;
  for(ix = 0; ix < nx_; ix += 1)
  { ofs << x << " " << p[index] << "\n";
    index += vsize;
    x += dx_;
  }
}

// Create a file name of the form:
//    NAME.DDD
```

```
// Return the name as a string
//******************************
string grf_pde_plain::make_file_name(string name,int index,string suffix)
{ char Buff[40];

  sprintf(Buff,"%d",index);

  return(name+"."+Buff);
}
```

Source file `grf_pde_plain.cc`

B.4 `Makefile`.

```
CFLAG = -ansi -O2 -Wall
INCLUDES = -I.
CC = g++

OBJ = pde_1d.o grf_pde_1d.o input_par.o grf_pde_plain.o

.cc:
$(CC) -c $< -o $*.o $(CFLAG) $(INCLUDES)
.cc.o:
$(CC) -c $< -o $*.o $(CFLAG) $(INCLUDES)


sineGordon: sineGordon.o $(OBJ)
$(CC) -o $@ $(CFLAG) $(INCLUDES) $@.o $(OBJ) -lm

radialHeat: radialHeat.o $(OBJ)
$(CC) -o $@ $(CFLAG) $(INCLUDES) $@.o $(OBJ) -lm

clean:
/bin/rm -f *.o core.* sineGordon radialHeat
```

Source file `Makefile`

C The relaxation method

This section lists the source files for the program `sineGordon_relax.cc`. The creation of these files are described in detail in section 6. The files given below must be put in the same directory as the `sineGordon.cc` program as it uses the same files. The `Makefile` given here can be used to compile `sineGordon.cc` and `sineGordon_relax.cc` buy simply executing the command `make`.

The parameter file `sineGordon_relax.par` can be used as follow

```
sineGordon_relax sineGordon_relax.par
```

```
#include "input_par.h"
#include <string>
#include <pde_1d_relax.h>
#include <grf_pde_1d.h>
#include <iostream>

#define NO_EQ      1
#define NO_DENS    1
#define NO_SCALARS 2

using namespace std;

char *VERSION = "sineGordon_relax: v1.0; 26/10/04; Author: B. Piette";

char const *Fcts_Names[] =
{ "f", 0
};

char const *Densities_Names[] =
{ "En", 0
};

char const *Scalars_Names[] =
{ "Pos1" , "Err", 0
};

struct {
  double x_min;
  double x_max;
  int nx;
  double dt_r;
  int  max_steps;
  int  grf_steps;
  int  scalars_steps;
  char grf_name[256];

  double k;
  double omega; // Relaxation parameter
  double prec;  // Relaxation precision
  int  no_kink;
} par;

char *help[] = {
 "sineGordon_relax PAR_FILE: ",
 "sineGordon_relax -h|-help: ouput this help message",
 "sineGordon_relax -v|-version: ouput the program version number.",
 "  Compute the static solution of the sine-Gordon equation.",
 " PAR_FILE:",
 "   x_min DOUBLE",
 "   x_max DOUBLE",
 "   nx INT",
 "   dt_r DOUBLE",
 "   max_steps INT",
 "   grf_steps INT",
```

```
 "   scalars_steps INT",
 "   grf_name STRING",
 "   k DOUBLE",
 "   omega DOUBLE",
 "   prec DOUBLE",
 "   no_kink int",
 0
};


/* sine-Gordon Equation */
class sg_static_pde : public pde_1d_GSR
{
public:
  sg_static_pde(double omega,double xmin,double xmax,
          int NoEq,int NoDens,int NoScalars,int nx,string f_name) :
    pde_1d_GSR(omega,xmin,xmax,NoEq,NoDens,NoScalars,nx,f_name,
          &Fcts_Names[0],&Densities_Names[0],&Scalars_Names[0])
  { err_ = 0;
  }

  /* Evaluate the Equation to solve */
  /* functions:                     */
  /* f[0] : f : sine Gordond fct    */
  /* f[1] : df : df/dt              */
  /* F : array for result           */
  /*********************************/
  void Eqn(double *F)
  {
    if(!fl_ || !fr_)  // Do nothing at the edge of the box
    { for(int i=0; i< NoEq_; i++) F[i] = 0;
      return;
    }

    eval_derivatives();

    F[0] = fxx_[0] -par.k*sin(f_[0]);
  }

  /* Compute Densities                         */
  /* D[0] :En  : Total energy density          */
  /*******************************************/
  double Dens(double *D)
  { double fx;
    int i;

    if(!fl_ || !fr_) // Desnisties are 0 at the edge of the box
    { for(i=0; i < NoDens_; i++) D[i] = 0;
      return(1.0);
    }

    fx = (fr_[0]-fl_[0])/tdx_;
    D[0] = 0.5*fx*fx+ par.k*(1.0-cos(f_[0]));
```

```
    return(1.0);
  }

  /* Compute the scalar data:       */
  /* xpi : position of f(x) = Pi    */
  /* err : relaxation error         */
  /**********************************/
  void Scalars()
  { double fl,fr,xpi=Xmin_;

    fr = Fcts_[0];
    for(int i = 1; i < nx_; i++)
    { fl = fr;
      fr = Fcts_[i*NoEq_];
      if (((fl <= M_PI)&&(M_PI < fr)) ||
          ((fr <= M_PI)&&(M_PI < fl)) )
      { xpi = Xmin_+(i+(fr-M_PI)/(fl-fr))*dx_;
      }
    }
    Scalars_[0] = xpi;
    Scalars_[1] = err_;
  };

  /* Implement some boundary conditions       */
  /*******************************************/
  void Boundary(double *fct)
  { // Nothing Here
  }

  /* Compute the initial condition.  */
  /* Fcts_[i*NO_EQ]   : f(Xmin)=2 Pi */
  /*                    f(Xmax)=0    */
  /***********************************/
  void Init(double x,double *F)
  { double Fl;

    Fl = par.no_kink*8*atan(1.0); // 2 Pi
    F[0] = Fl*(Xmax_-x)/(Xmax_-Xmin_);
  }
};

/*****************************************/
/* Ouput a help message on cerr and exit */
/*****************************************/
void Help()
{ int i=0;
  while(help[i]) cerr << help[i++] << std::endl;
  exit(1);
}

int main(int argc, char **argv)
{ grf_pde_1d *grfp;

  try
```

```
  { // One argument expected.
    if((argc != 2) || !strcmp(argv[1],"-help") || !strcmp(argv[1],"-h"))
    { Help();
    }
    if(!strcmp(argv[1],"-version") || !strcmp(argv[1],"-v"))
    { cerr << VERSION<<"\n";
      exit(0);
    }


        // Read the parameters for the input file
        // The parameter filename is the 1st argument
    input_par ip(argv[1]);

    ip.get("x_min",par.x_min);
    ip.get("x_max",par.x_max);
    ip.get("nx",par.nx);
    ip.get("dt_r",par.dt_r);
    ip.get("max_steps",par.max_steps);
    ip.get("grf_steps",par.grf_steps);
    ip.get("scalars_steps",par.scalars_steps);
    ip.get("grf_name",par.grf_name);

    ip.get("k",par.k);
    ip.get("omega",par.omega);
    ip.get("prec",par.prec);
    ip.get("no_kink",par.no_kink);

          // Build a pde for sineGordon
    sg_static_pde pde(par.omega,par.x_min,par.x_max,
                      NO_EQ,NO_DENS,NO_SCALARS,par.nx,par.grf_name);

          // Set dt by hand : dx * par.dt_r
    pde.set_dt(pde.dx()*par.dt_r);

          // Create a new graphic object
    grfp = new grf_pde_1d(&pde,par.grf_name);
    pde.init_grf(grfp);

    pde.initial_value();

    pde.relax_grf_until(par.prec,par.scalars_steps,par.grf_steps,
                        par.max_steps);
  }
  catch(string e)
  { cerr << e << std::endl;
  }
}
```

Source file `sineGordon_relax.cc`

C.1 pde_1d_relax.cc.

```
#include <pde_1d_relax.h>
#include <grf_pde_1d.h>
#include <string>
#include <iostream>
#include <fstream>
#include <cstdio>
#include <cmath>

using namespace std;

/***************************************************************/
/* Perform n relaxation step on the grid with graphics         */
/* Output scalars every scalars_step                           */
/* Output graphs every grf_step                                */
/* Update the time t_                                          */
/* prec : stopes when err < prec                               */
/* scalars_step : no of steps between scalars outputs          */
/* grf_step : no of steps between graphics                     */
/* max_steps : maximum no of steps (<=0 -> no limit)           */
/* Return : the error or -1 if max_steps_ exceeded.            */
/***************************************************************/
double pde_1d_relax::relax_grf_until(double prec,int scalars_step,int grf_step,
                               int max_steps)
{ int i;
  string grf_name;
  double err;

  write_header_scalars();

  i = 0;
  err = (prec+1)*2;
  while((err > prec)&& ((max_steps <= 0)||(i < max_steps)))
  { if(((i%scalars_step) == 0) || ((i%grf_step) == 0))
    { computeDensities();
      Scalars();
      if((i%grf_step) == 0)     { write_grf(); }
      if((i%scalars_step) == 0) { write_scalars(); }
    }
    err = relax_1step();
    i++;
  }

  if(err > prec) return(-1);

  // Always output graphics at the end
  computeDensities();
  Scalars();
  write_grf();
  write_scalars();
  return(err);
}
```

```
/******************************** RELAXATION METHOD **********************/
/******************************** GSR ************************************/
/******************************/
/* Constructor for GSR pde_1d */
/* Allocated 2 extra arrays   */
/******************************/
pde_1d_GSR::pde_1d_GSR(double omega,double xmin,double xmax,
          int NoEq,int NoDens,int NoScalars,int nx,std::string f_name,
          char const *Fct_Names[],char const *Density_Names[],
          char const *Scalars_Names[]) :
  pde_1d_relax(xmin,xmax,NoEq,NoDens,NoScalars,nx,f_name,
         Fct_Names,Density_Names,Scalars_Names),
  omega_(omega)
{
}

/*****************************************************************/
/* Perform a single Gauss-Seidel Relaxation step on the grid     */
/* pde : pde parameters                                          */
/* omega = relaxation parameter.                                 */
/* Eqn(Force) : a function which computes the right              */
/*     hand side of the equation.                                */
/*       Force : vector where to store the result.               */
/*       i_     : lattice index                                  */
/* Return : the largest error                                    */
/*****************************************************************/
double pde_1d_GSR::relax_1step()
{ int i,j;
  double *f,Error,er,dxsot;

  Error = 0;
  dxsot = dxs_*0.5;

      /* Scan every point on the grid */
  for(i=0; i <= lastPoint_; i++)
  { set_neighbour_fcts(Fcts_,i);

        /* Computes the force for point i and  */
        /* stores it in Force                  */
    Eqn(Force_);

    /* Update each fct using Force : f(n+1) = f(n) + pde->omega*Force */
    f = &Fcts_[i*NoEq_];
    for (j = 0; j < NoEq_; j++)
    { f[j] += omega_ * dxsot* Force_[j];

      er = Force_[j]*Force_[j];
      if (er > Error) Error = er;
    }
  }

  Boundary(Fcts_);
  t_ += 1.0;  /* Udate the time : interation number! */
```

```
  err_ = sqrt(Error);
  return(err_);
}
```

Source file pde_1d_relax.cc

C.2 pde_1d_relax.h.

```
#ifndef PDE_1D_RELAX_H
#define PDE_1D_RELAX_H

#include <cmath>
#include <time.h>
#include <cstring>
#include <cstdio>
#include <string>
#include "pde_1d.h"

class pde_1d_relax : public pde_1d {
public:
  pde_1d_relax(double xmin,double xmax,int NoEq,int NoDens,int NoScalars,
           int nx,std::string file_name,char const *Fct_Names[],
           char const *Density_Names[],char const *Scalars_Names[])
    : pde_1d(xmin,xmax,NoEq,NoDens,NoScalars,nx,file_name,Fct_Names,
             Density_Names,Scalars_Names) {};
  virtual ~pde_1d_relax() {};

  virtual double relax_1step()=0;
  virtual double relax_grf_until(double prec,int scalars_step,int grf_step,
                                 int max_steps=0);
  double err_;         /* Error, used by relaxation method          */
};

/******************************* RELAXATION METHOD **********************/
/******************************* GSR ***********************************/
class pde_1d_GSR : public pde_1d_relax {

public:
  pde_1d_GSR(double omega,double xmin,double xmax,
          int NoEq,int NoDens,int NoScalars,int nx,std::string f_name,
          char const *Fct_Names[],char const *Density_Names[],
          char const *Scalars_Names[]);
  pde_1d_GSR();
  double relax_1step();

protected:
  double omega_;       /* Over relaxation parameter (init to 1)     */
};
#endif
```

Source file pde_1d_relax.h

C.3 Makefile.

```
CFLAG = -ansi -O2 -Wall
INCLUDES = -I.
CC = g++

OBJ = pde_1d.o pde_1d_relax.o grf_pde_1d.o input_par.o

.cc:
$(CC) -c $< -o $*.o $(CFLAG) $(INCLUDES)
.cc.o:
$(CC) -c $< -o $*.o $(CFLAG) $(INCLUDES)

ALL: sineGordon sineGordon_relax

sineGordon: sineGordon.o $(OBJ)
$(CC) -o $@ $(CFLAG) $(INCLUDES) $@.o $(OBJ) -lm

sineGordon_relax: sineGordon_relax.o $(OBJ)
$(CC) -o $@ $(CFLAG) $(INCLUDES) $@.o $(OBJ) -lm

clean:
/bin/rm -f *.o core.* sineGordon sineGordon_relax
```

Source file Makefile

C.4 sineGordon_relax.par.

```
x_min -20
x_max 20
nx 401
dt_r 0.25
max_steps 0
grf_steps 10000
scalars_steps 10000
grf_name DATA/SG_rel_tests
k 1.0
omega 1.6
prec 1e-6
no_kink 2
```

Source file sineGordon_relax.par

D User guides

D.1 Using input_par.

Programs often need to use several parameters. One technique used by programmer is to write these parameters inside the program itself. This is a very bad idea as the program must be recompiled each time. A better approach is to write the parameter values into a file and get the application to read that file.

This is done by many programmers, but many simply put the values one after another in a predetermined order without any clue as to which value correspond to what parameters. The only solution to know the position of a given parameter in the file is to look at the source file. Again, this is a bad idea.

A better approach is to write each parameter as a pair of words. The first word is the name of the parameter, followed by one or more spaces and the value for the parameter. For example a parameter file for `sineGordon` will look like this

```
x_min -10
x_max 10
nx 101
dt_r 0.25
t_max 10
k 1.0
steps 1000
grf_steps 100
scalars_steps 10
grf_name DATA/SGtests
```

The order is predetermined but at least the name of each parameter appears in the file, making it much easier to modify.

To read the parameter file, one can use the class `input_par`. Using it is very simple. First one construct an instance of the class. The constructor takes a single argument which is the name of the parameter file. Then one reads each parameter one by one using the method `get`.

The first argument of `get` is the name of the parameter. `get` reads the next word in the file and check it again the parameter name. If they differ, it throws an error message.

The second argument is a reference to the variable to read the parameter into. This method is overloaded for three different types of this arguments: `doubles`, `ints` and `char *`'s. The compiler does pick the correct one automatically.

To read the parameter file above, one would proceed as follow:

```
struct {
  double x_min;
  double x_max;
  int nx;
  double dt_r;
  double t_max;
  double k;
```

```
  double grf_dt;
  double scalars_dt;
  char grf_name[256];
} par;

input_par ip("sineGordon.par");
ip.get("x_min",par.x_min);
ip.get("x_max",par.x_max);
ip.get("nx",par.nx);
ip.get("dt_r",par.dt_r);
ip.get("t_max",par.t_max);
ip.get("k",par.k);
ip.get("grf_dt",par.grf_dt);
ip.get("scalars_dt",par.scalars_dt);
ip.get("grf_name",par.grf_name);
```

D.2 Using `make`.

To be able to run a program written in C++, one must first convert it to an executable program. This process is called the compilation of the program and it is performed by a so-called compiler. In these notes, we use the GNU C++ complier called `g++`. To compile a single program, say `MyApp.cc` one just needs to execute the command

```
g++ -o MyApp MyApp.cc -lm
```

This creates the file `MyApp` which can then be run like any other application. The flag `-lm` is necessary to link the maths library to the program.

Most programs in these notes are actually composed of more than one source file (.cc files), so we must also group them together into a single program. This is called linking and it is also done with the compiler. For example, the program `sineGordon` is made out of the files `sineGordon.cc`, the main program, as well as `grf_pde_1d.cc`, `pde_1d.cc` and `input_par.cc`. To generate an executable file, the compiler is used to generate an object file (.o file) for each C++ files and then link these object files together as follow:

```
g++ -c sineGordon.cc
g++ -c grf_pde_1d.cc
g++ -c pde_1d.cc
g++ -c input_par.cc
g++ -o sineGordon sineGordon.o grf_pde_1d.o pde_1d.o input_par.o -lm
```

The first 4 lines create the 4 object files `difSineGordon.o`, `grf_pde_1d.o`, `pde_1d.o` and `input_par.o`. The last line links these object files into the executable file called `sineGordon`.

This can be achieved in a much simpler way by using a `Makefile`. `make` is a unix command that can be used to automatically compile programs that are made out of many files. The programmer describe, in a file called `Makefile`, all the components of the program to compile. In our example, the program `sineGordon` is made from the object files `sineGordon.o`, `pde_1d.o`, `grf_pde_1d.o` and `input_par.o`. Moreover each object file is created by compiling the corresponding `cc` file.

In our example the `Makefile` will thus look like this:

```
CFLAG = -ansi -O2 -Wall
INCLUDES = -I.
CC = g++
OBJ = pde_1d.o grf_pde_1d.o input_par.o

.cc.o:
	$(CC) -c $< -o $*.o $(CFLAG) $(INCLUDES)

sineGordon: sineGordon.o $(OBJ)
	$(CC) -o $@ $(CFLAG) $@.o $(OBJ) -lm

clean:
	/bin/rm -f *.o core.* sineGordon
```

The first line defines the variable `CFLAG` listing the set of compiler parameters that we want to use: `-ansi` force strict ANSI compliance; `-O2` tell the compiler to optimise the code; `-Wall` ask the compiler to display a comprehensive set of warning messages.

The second lines define the variable `INCLUDE` with is an include argument for the compiler. It tells the compiler to search the directory path-name following the `-I` for header (`.h`) files. We use the current directory in this case, but it is a good practise to put the header files in a separate directory, called say `include`. In that case we would have `INCLUDES = -I./include`.

The third line defines a variable `CC` which holds the name of the compiler.

The fourth line define the variable `OBJ` which contains the list of all the object files used by the main program.

The following two lines tells `make` how to create an object files from a `.cc` file. Any file called `X.cc` is compiled into an object files by executing the command `$(CC) -c X.cc -o X.o $(CFLAG) $(INCLUDES)`. Refer to the `make` documentation for more information. Notice that the second line is indented. This indentation MUST BE done by putting a TAB character as the FIRST character of the line, otherwise `make` produces an error message. White spaces do not work! Notice that `$<` and `$*` are special variables standing respectively for the name of the source file and the name of file to make without the suffix. So, when making `pde_1d.o`, `$<` and `$*` will stand respectively for `pde_1d.cc` and `pde_1d`.

The line `sineGordon:  sineGordon.o $(OBJ)` tells `make` that the file `sineGordon` is made from the object files `sineGordon.o` as well as the object files listed in the variable `$(OBJ)`. The following line tells the compiler how to link the 4 object file into the executable file `sineGordon`. Notice that the variable `$@` stands for the name of the target to make, *ie* `sineGordon` in our example.

To compile the program `sineGordon` all one has to do is to executes the command `make sineGordon`. `make` then searches the `Makefile` for a line staring with `sineGordon:`. This tells it that the `sineGordon` file is made from 4 object file. The lines above has also tells it how to create an object file from a `.cc` file. Make then creates the four object files from their C++ source file. Having done that it executes the indented commands following the `sineGordon:` line; it links the 4 object files into an the executable file `sineGordon`.

The beauty of `make` is that it is clever: it only do the work that needs to be done. Instead of compiling very single source file each time, it checks if the object files exist already. When they do, it then checks if they are younger that the corresponding source file. Those that are don't need to be compile again and make skip their compilation. The same apply to the linking command. If the file `sineGordon` is younger that the 4 object files, make does not link them again.

The last two lines are used to tidy up the directory. The command `make clean`, removes all the object files and the executable file `sineGordon`.

When solving the exercises from section 6, one needs to add one line for each program being written. For example, for the second exercise, add the lines

```
sineGordon_1soliton: sineGordon_1soliton.o $(OBJ)
	$(CC) -o $@ $(CFLAG) $@.o $(OBJ) -lm
```

just before the line starting with `clean:`. Remember that the first character of the second line MUST be TAB character. One can also insert blank lines after these.

WARNING: if one only modifies one or more header files, the `make` command will not recompile the sources that depend on it unless they have been modified themselves. One can construct `Makefile` that will do it, but this is rather tricky. When in doubt, do a `make clean` before recompiling.

D.3 Using `gnuplot`.

`gnuplot` is a public domain program that can be used to draw graphics. It can be used amongst other things to display a function stored in a file. Assuming the file `file.dat` contains pairs of value on every line, the following:

```
% gnuplot
gnuplot> plot 'file.dat'
```

will plot the data stored in `file.dat`: the values in the second column are plotted as a function of the first column of values.

To plot the data using lines instead of crosses one does the following:

```
gnuplot> plot 'file.dat' with lines
```

If the file `file.dat` contains more than one column, it is possible to plot any pair of columns versus each other. For example,

```
gnuplot> plot 'file.dat' using 1:3
```

will plot the third column of `file.dat` versus the first one.

It is also possible to plot more that one graph at a time. To plot simultaneously the second and third column of `file.dat` versus the first column:

```
gnuplot> plot 'file.dat' using 1:2, 'file.dat' using 1:3 with lines
```

The graphic appears on the screen. To quit `gnuplot`, simply type `exit`.

It is also possible to create a postscript file directly from `gnuplot`:

```
gnuplot> set term post landscape
Terminal type set to 'postscript'
Options are 'landscape monochrome "Courier" 14'
gnuplot> set output "file.ps"
gnuplot> plot 'file.dat'
```

In this example, a postscript file called `file.ps` is created. It can be sent to a postscript printer with the command `lpr file.ps`. The name we have used for the files in this example can obviously be changed.

In one of the problem we suggest to produce surface plots to show the time evolution of the solitons. The data files generated can be displayed with `gnuplot` as follow:

```
gnuplot> set parametric
gnuplot> splot 'file.dat' with lines
```

References

[1] W.H. Press et al, Numerical Recipes in C, *Cambridge University Press*

[2] Herbert Schildt, *The Complete Reference, C++* McGraw Hill.

[3] P.G. Drazin and R.S. Johnson. Solitons: an introduction. *Cambridge University Press*

[4] H. Sutter. *Exceptional C++* Addison-Wesley

[5] B. Piette, W.J. Zakrzewski. Metastable Stationary Solutions of the radial d dimensional Sine-Gordon Model. Nonlinearity **11** (1998) 1103-1110