Numerics

Jonathan A. Pearson*
Centre for Particle Theory, Department of Mathematical Sciences, Durham University, South Road, Durham, DH1 3LE, U.K.
(Dated: April 16, 2014)

Contents

I. INTRODUCTION

These notes contain descriptions of the numerics and algorithms.
The idea is to construct a code to evolve

$$\Box\Phi + m^2\Phi = 0, \tag{1.1}$$

where

$$\Box\Phi = g^{\mu\nu}\nabla_\mu\nabla_\nu\Phi, \tag{1.2}$$

in which the metric is

$$g_{\mu\nu}\mathrm{d}x^\mu\mathrm{d}x^\nu = a^2(\tau)\Big[-(1+2V)\mathrm{d}\tau^2 + (1-2V)\mathrm{d}x_i\mathrm{d}x^i\Big], \tag{1.3}$$

*Electronic address: jonathan.pearson@durham.ac.uk

and V solves the Poisson equation,

$$\nabla^2 V = 4\pi G a^2 \rho_{\rm dust}\Big(|\Phi|^2 - 1\Big), \qquad \rho_{\rm dust} = \bar{\rho}_{\rm dust}/a^3 \tag{1.4}$$

A. Literature review

[1], [2], [3]

B. Discretization onto a lattice

Space will be discretized onto a lattice; the spacing between lattice sites is h. A quantity, Q, in 2D is discretized so that it lives on the lattice; $Q(x,y) \to Q_{i,j}$.

The discretization of the first derivative of a function $f(\mathbf{x})$ at lattice site "i" is

$$\frac{\partial f}{\partial x} = \frac{f_{i+1} - f_{i-1}}{2h}, \tag{1.5}$$

and the second derivative discretized to second order is

$$\frac{\partial^2 f}{\partial x^2} = \frac{f_{i+1} + f_{i-1} - 2f_i}{h^2}. \tag{1.6}$$

We could also discretize second derivatives to fourth order via

$$\frac{\partial^2 f}{\partial x^2} = \frac{-f_{i+2} + 16 f_{i+1} - 30 f_i + 16 f_{i-1} - f_{i-2}}{12 h^2}. \tag{1.7}$$

II. METHODS TO DISCRETIZE DERIVATIVES

The method of finite differences for computing derivatives on a lattice is (very) quick, but has rather large errors for noisy data. There are also *spectral methods*, which use the discrete Fourier transform of the data. These are especially useful for periodic systems. To see how spectral methods work, note that from the (inverse Fourier transform of) a function

$$f(x) = \int \mathrm{d}k \hat{f}(k) e^{\mathrm{i}kx} =: \mathcal{F}^{-1}(\hat{f}) \tag{2.1}$$

one can easily compute the first and second derivatives

$$\frac{\partial}{\partial x} f = \mathcal{F}^{-1}(\mathrm{i}k\hat{f}), \qquad \frac{\partial^2}{\partial x^2} f = \mathcal{F}^{-1}(-k^2 \hat{f}). \tag{2.2}$$

We have chose to implement the discrete Fourier transform (and its inverse) via the library `FFTW3`. We have tested accuracy .vs. time taken to compute the derivatives, we have constructed a simple code which computes the derivative of sine waves in a box with space step-size `h`, and `imax` grid-points. We compute the time taken to construct derivative using finite difference and FFT schemes, and the error on each (computed against exact analytic expressions for the derivatives).

By way of setup,

```
// Space step-size
double h = 0.05;
// Number of grid-points
int imax = 200;
// Function type (just for this testing purpose)
int fn_type = 2;
// How many waves in the box?
double wavn =  6.0;
double omega = wavn * 2.0 * PI / ( h * imax );
double fn(double x){
    if(fn_type == 1) return sin( x * omega );
    if(fn_type == 2) return sin( x * omega ) + sin( x * 2.0 * omega );
}
```

Here is the output

```
// Space step-size
double h = 0.2;

FT(data) :: 1.86043 milliseconds
iFT(d_data) :: 1.17383 milliseconds
FT(data) :: 0.041698 milliseconds
iFT(dd_data) :: 0.0388 milliseconds
// Errors
Error on FD first derivative: 0.028334
Error on FD second derivative: 0.0266803
Error on FFT first derivative: 1.45685e-14
Error on FFT second derivative: 1.6575e-13
// Compute times
FD first derivative compute time: 0.015488 milliseconds
FD second derivative compute time: 0.005887 milliseconds
FFT first derivative compute time: 3.11555 milliseconds
FFT second derivative compute time: 0.119432 milliseconds
```

III. ELLIPTIC PDES

Here we outline some methods of numerically solving elliptic partial differential equations. As an example, we will show how to solve the 2D Poisson equation,

$$\nabla^2 V = -\rho. \tag{3.1}$$

Here, $V = V(\mathbf{x})$ is a "potential", and $\rho = \rho(\mathbf{x})$ a source. Using the finite difference discretization scheme outlined in section I B, the discrete version of this is

$$V_{i+1,j} + V_{i-1,j} - 4V_{i,j} + V_{i,j+1} + V_{i,j-1} = -h^2 \rho_{i,j}. \tag{3.2}$$

Here we used second order accurate derivative discretization. This can be rearranged to find

$$V_{i,j} = \frac{1}{4}\Big[V_{i+1,j} + V_{i-1,j} + V_{i,j+1} + V_{i,j-1} + h^2 \rho_{i,j}\Big]. \tag{3.3}$$

There is clearly an issue: the value of the potential (which is to be found) relies on knowledge of the potential on its four neighbouring lattice sites. The remedy is to update the potential using (fictitious) time-steps;

$$V_{i,j}^{n+1} = \frac{1}{4}\Big[V_{i+1,j}^{n} + V_{i-1,j}^{n} + V_{i,j+1}^{n} + V_{i,j-1}^{n} + h^2\rho_{i,j}\Big]. \tag{3.4}$$

This is Jacobi's iterative method. As n increments, the values of the potential V converge onto those which solve the Poisson equation. There are clever ways of doing this, and each has different convergence properties.

A. Gauss-Seidel method

The Jacobi method (3.4) can be made faster by using the newly updated field values on the RHS,

$$V_{i,j}^{n+1} = \frac{1}{4}\Big[V_{i+1,j}^{n} + V_{i-1,j}^{n+1} + V_{i,j+1}^{n} + V_{i,j-1}^{n+1} + h^2\rho_{i,j}\Big]. \tag{3.5}$$

B. Successive over relaxation

Another method is to using a different linear combination,

$$V_{i,j}^{n+1} = (1-\omega)V_{i,j}^{n} + \frac{\omega}{4}\Big[V_{i+1,j}^{n} + V_{i-1,j}^{n+1} + V_{i,j+1}^{n} + V_{i,j-1}^{n+1} + h^2\rho_{i,j}\Big] \tag{3.6}$$

The parameter ω is the SoR parameter. Its value determines the convergence of the algorithm:

- Only convergent if $0 < \omega < 2$,
- Faster than Gauss-Seidel if $1 < \omega < 2$,
- Fastest on square lattice if $\omega \sim 2/(1+\pi/L)$, where L is the number of lattice points in each direction.

C. Checks against known analytic solutions

We want to check the convergence properties of the codes against known solutions.

$$\rho(x,y) = 2x^3 + 6xy(y-1), \qquad V(x,y) = x^3y(1-y) \tag{3.7}$$

IV. HYPERBOLIC PDES

Here we outline how to solve hyperbolic PDEs; that is, wave equations of the form

$$\ddot{\Phi} - \nabla^2\Phi + U'(\Phi) = S(\mathbf{x}). \tag{4.1}$$

Here, $S(\mathbf{x})$ is some source term (which will be important for the applications we have in mind), and $U(\Phi)$ is the scalar field potential; typically, $U = \frac{1}{2}m^2\Phi^2$ is a mass term. First, we trivially re-write this as

$$\ddot{\Phi} = \nabla^2\Phi - U' + S, \tag{4.2}$$

so that nothing on the RHS has time derivatives. We will write everything in 2D to reduce the number of indices needed to write the expressions down: the code is in 3D, and we shall explicitly point out any subtleties associates with going from 2D to 3D.

We define an "equation of motion" term

$$\mathcal{E}^t_{i,j} \equiv \nabla^2\Phi^t_{i,j} - U'^t_{i,j} + S^t_{i,j}, \tag{4.3}$$

where the Laplacian is discretized as

$$\nabla^2\Phi^t_{i,j} = \frac{\Phi^t_{i+1,j} + \Phi^t_{i-1,j} - 2\Phi^t_{i,j}}{h^2} + \frac{\Phi^t_{i,j+1} + \Phi^t_{i,j-1} - 2\Phi^t_{i,j}}{h^2}. \tag{4.4}$$

The second time derivative at a given location is discretized to second order as

$$\ddot{\Phi} = \frac{\Phi^{t+1}_{i,j} + \Phi^{t-1}_{i,j} - 2\Phi^t_{i,j}}{h_t^2}. \tag{4.5}$$

Hence,

$$\Phi^{t+1}_{i,j} = h_t^2\mathcal{E}^t_{i,j} - \Phi^{t-1}_{i,j} + 2\Phi^t_{i,j}. \tag{4.6}$$

This gives a rule to update the value of the field at each location.

A. Tests of the code

To check the code, we have implemented a few different types of initial conditions and potentials (this also helps to build intuition). To make sure that the code accesses arrays correctly, we begin by solving the gradient flow equation

$$\dot{\Phi} = \nabla^2\Phi - U'(\Phi), \tag{4.7}$$

for two different potentials:

$$U_{(1)}(\Phi) = \frac{1}{2}\Phi^2, \qquad U_{(2)}(\Phi) = \frac{1}{4}(\Phi^2 - 1)^2. \tag{4.8}$$

For a system endowed with the potential $U_{(1)}$, any homogeneous initial field configuration will evolve towards $\Phi = 0$, since those are the values which minimise the potential. Similarly, a system endowed with $U_{(2)}$ with homogeneous initial conditions will evolve towards $\Phi = \pm 1$.

The space- and time-step sizes must be carefully chosen: not all values give numerically stable evolutions. For the free wave equation, $\ddot{\Phi} - \nabla^2\Phi = 0$, one typically requires

$$h_t < h, \tag{4.9}$$

and for the gradient flow equation $\dot{\Phi} - \nabla^2\Phi = 0$ one typically requires

$$h_t < h^2. \tag{4.10}$$

Appendix A: Using the code

1. Klein-Gordon solver

The code is written in C^{++}, and requires `GSL` and `boost` libraries. To check that these are both installed correctly, nagivate to the `KGsolve` root directory, the compiling and running of the code is performed with the following three commands:

```
make clean
make
./main
```

The code should compile without error. The source code is in the directory `src`.

The `GetParams` function reads in the `params.ini` file, then the `SetupGrid` function tells the struct

```
GRIDINFO grid;
```

how many grid-points are in the `i`, `j`, `k`-directions, and what the lattice and time step-sizes are. Immediately after the grid is setup, the field is setup in the function `SetupField`. The field is stored as a one-dimensional array, and is part of a struct named `field`. To generate one,

```
FIELDCONTAINER field;
```

This sets up a number of important entities:

- `field.vals` – array which holds all of the values of the field at all locations, for all components of the field (if the field has multiple components).
- `field.deriv_X` – array which holds the $\hat{X}$-derivative of each component of the field at a given location. Explicitly, there are `deriv_x, deriv_y, deriv_z`.
- `field.laplacian` – array holds the Laplacian of each component of the field at a given location.
- `field.dpot` – array holds the derivative of the field potential, with respect to a given component of the field, at a given location.
- `field.eom` – array holds everything which equals the time derivative of a given component of the field, at a given location.

Functions for computing the values of these quantities are found in the file `filestruct.h`.

The value of the field at an array location is `field.vals[pos]` (usually, `field->vals[pos]` needs to be typed outside of the `main.cpp` file). To access the value of the field corresponding to a given component, at a given time, and position on the grid, one should call the function `ind`. As an example, to get the current value of the c$^{\text{th}}$-component of the field, at the location whose lattice sites are (i, j, k), one calls

```
int pos = ind(now,c,i,j,k,grid,field);
field.vals[pos];
```

The code only stores two time-steps of the field: this is all thats required for second order accurate time derivatives. A log file is then written with all parameter choices etc. That completes the setup of the code.

Next is the initial condition setup. The function `InitialConditions` is called which sets `field.vals` at all spatial locations, for the first two time-steps. The parameter `params.inittype` can be used to select how to do the initial conditions. There are three types coded up by default: `inittype = 0` sets the field homogeneous, `inittype = 1` sets the field at random values about the origin, and `inittype = 2` sets the field with a discontinuous kink in the x-direction.

After the initial conditions have been set, the function `SolveKG3D` is called which solves the "Klein-Gordon" equation. In the function, a struct

```
THIST timehistory;
```

is setup to contain "time history" information – this is usually (spatially) integrated information about the grid at a given time-step. There is a loop which runs untill `params.ntimsteps` is reached. At each time-step, there are loops over all 3 spatial directions. At each location the following sequence is called

- `field->GetDeriv` – gets all spatial derivatives of the field
- `field->Getdpot` – gets the derivative of the potential w.r.t each component of the field. The type of potential is selected via the parameter `params.pottype`. Currently, `pottype = 0` corresponds to a massive scalar field, and `pottype = 1` to a Higgs potential.
- `field->GetEoM` – constructs the RHS of the equation of motion. The type of equation of motion is selected via the parameter `params.eomtype`. Currently, `eomtype = 0` corresponds to $\mathcal{E}_i = \nabla^2\phi_i - \partial V/\partial\phi_i$. Here, $\mathcal{E}_i$ is the equation of motion of the ϕ_i-component of the field.
- `field->UpdateField` – updates the value of the field. The type of update rule is selected via the parameter `params.evoltype`. Currently, `evoltype = 0` corresponds to a gradient flow evolution, $\dot{\phi}_i = \mathcal{E}_i$, and `evoltype = 1` corresponds to a second order time derivative evolution, $\ddot{\phi}_i = \mathcal{E}_i$.

All of these functions are explicitly implemented in `fieldstruct.h`.

When the time-step number is divisible by the parameter `filefreq`, then the field is dumped to file via the function call `WriteFieldData` (which is defined in `fieldstruct.h`).

Once these loops are completed, the function `CleanField` deletes all the allocated memory, and the final parts to the logfile are written and printed to screen.

2. Contents of each source code file

- `main.cpp`
- `setup.cpp`
  - `SetupGrid`
  - `SetupField`

GetParams

CheckParams

checkdirexists

- initialconditions.cpp

  InitialConditions – calls SetInitialConditions at each location

  SetInitialConditions – set the initial conditions at a given location

- kgsolve.cpp

  SolveKG3D

- fieldstruct.h

  ind – returns array index for field values at a given time, component, location

  GetDeriv – selects which scheme to use for spatial derivatives

  GetDeriv_2 – returns second order accurate finite difference spatial derivatives

  GetDeriv_4 – returns fourth order accurate finite difference spatial derivatives

  Getpot – returns field potential

  Getdpot – returns derivative of field potential w.r.t each component as array

  GetEoM – returns RHS of equation of motion

  UpdateField – updates field value

  WriteFieldData – dumps field value to file

  CleanField – deletes the allocated memory

- gridstruct.h

  SetTime – sets the time-access index accordingly

  GetPos – sets values of `grid.loc_X`, and calls GetP and GetM

  GetP – sets values of `grid.Xp` (e.g. `ip=i+1`), using periodic boundaries

  GetM – sets values of `grid.Xm` (e.g. `im=i-1`), using periodic boundaries

- timehistorystruct.h

  SetFieldValDump – sets field locations to dump into timehistory file

  write – writes timehistory items

  CleanUp – deletes allocated memory

---

[1] L. M. Widrow and N. Kaiser, *Using the Schroedinger Equation to Simulate Collisionless Matter*, *Astrophysical Journal Letters* **416** (Oct., 1993) L71.

[2] L. M. Widrow, *Modeling collisionless matter in general relativity: A New numerical technique*, *Phys.Rev.* **D55** (1997) 5997–6001, [astro-ph/9607124].

[3] C. Uhlemann, M. Kopp, and T. Haugg, *Schrödinger method as N-body double and UV completion of dust*, arXiv:1403.5567.