Neolith

A High-Order Parallel Population Dynamics Code

Andrew Baggaley
School of Mathematics and Statistics
Newcastle Univeristy
U.K.
a.w.baggaley@gmail.com

March 9, 2010

Contents

1 Introduction

Welcome to the documentation for the NEOLITH population dynamics code. The code is written in well commented Fortran, arranged in a modular structure. The primary aim of this documentation is to highlight which module(s) are involved in the various sequences of the code. For a shorter guide refer to appendix B, which gives an outline of each of the .f90 files used in the code. To run NEOLITH you need to be on a *UNIX*-type system (*Make* and *bash* are heavily used) with a Fortran compiler linked to the MPI libraries (e. g. mpif90). All the Fortran code can be found in the directory *src*. MATLAB is used for post-processing, although it is probably very straightforward to convert the MATLAB scripts (found in *bin*) to work with *Octave*, freely available under the GNU license. Please do not hesitate to contact with any queries, or if you are interested in helping to further develop the code.

2 Equations

Currently we solve the two dimensional Fisher-Kolmogorov-Petrovsky-Piskunov (FKPP) equation [3], for the population density $N(\mathbf{x}, t)$ in Cartesian or spherical geometry, with a spacially varying diffusivity $\nu(\mathbf{x})$ and carrying capactiy $K\mathbf{x}$ with an advective velocity V:

$$\frac{\partial N(\mathbf{x}, t)}{\partial t} + (V \cdot \nabla)N = \gamma N \left(1 - \frac{N}{K}\right) + \nabla \cdot (\nu \nabla N). \tag{1}$$

The 'constants' (γ, K) can be altered in the run parameters file (*run.in*), note these can be altered and a simulation re-run without re-compilation. The initial conditions for N and ν can also be set in the same file. The code also features a Lagrangian mode, which simply tracks the propagation of the wavefront of the solution to the FKPP equation. For more details see §7

2.1 Co-ordinate systems

We can rewrite Eq. 1 as,

$$\frac{\partial N(\mathbf{x}, t)}{\partial t} = -(V \cdot \nabla)N + \gamma N \left(1 - \frac{N}{K}\right) + \nabla\nu \cdot \nabla N + \nu \nabla^2 N. \tag{2}$$

It is clear that we must evaluate both the gradient and Laplacian operators in our code. In two-dimensional Cartesian geometry these are given by,

$$\nabla N(x,y,t) = \frac{\partial N}{\partial x} + \frac{\partial N}{\partial y}, \qquad \nabla^2 N(x,y,t) = \frac{\partial^2 N}{\partial x^2} + \frac{\partial^2 N}{\partial y^2}. \tag{3}$$

In spherical polar co-ordinates (at fixed radius r),

$$\nabla N(\phi,\theta,t) = \frac{1}{r}\frac{\partial N}{\partial \theta} + \frac{1}{r\sin\theta}\frac{\partial N}{\partial \phi} \tag{4}$$

and

$$\nabla^2 N(\phi,\theta,t) = \frac{1}{r^2\sin\theta}\frac{\partial}{\partial\theta}\left(\sin\theta\frac{\partial N}{\partial\theta}\right) + \frac{1}{r^2\sin^2\theta}\frac{\partial^2 N}{\partial\phi^2}. \tag{5}$$

The latter of which, with a little manipulation, can be re-written in the following (numerically easy to implement) way,

$$\nabla^2 N(\phi,\theta,t) = \frac{\cot\theta}{r^2}\frac{\partial N}{\partial\theta} + \frac{1}{r^2}\frac{\partial^2 N}{\partial\theta^2} + \frac{1}{r^2\sin^2\theta}\frac{\partial^2 N}{\partial\phi^2}. \tag{6}$$

You can choose between the two in *cparam.local*.

3 Numerical Details

3.1 Finite difference schemes

We numerically discretise the domain into an evenly spaced mesh with spacing of the size δx. High order (6^{th}) explicit finite difference schemes are used such that for a function, $f(\mathbf{x},t)$, the first derivative is given by [2],

$$f'_i = (-f_{i-3} + 9f_{i-2} - 45f_{i-1} + 45f_{i+1} - 9f_{i+2} + f_{i+3})/(60\delta x), \tag{7}$$

and the second derivative by,

$$f''_i = (2f_{i-3} - 27f_{i-2} + 270f_{i-1} - 490f_i + 270f_{i+1} - 27f_{i+2} + 2f_{i+3})/(180\delta x^2). \tag{8}$$

Lower order derivatives are available for testing (in Cartesian geometry only at present), there is a flag in in *cparam.local*, which can be altered to implement lower order derivatives. These derivatives are calculated in the subroutine *get_rhs* which can be found in the *derivatives.f90* module.

3.2 $2N$ Runge-Kutta

For the time integration we use a high order, memory efficient $2N$-schemes [5]. If we seek to evaluate

$$\frac{\partial N(\mathbf{x},t)}{\partial t} = F(\mathbf{x},t), \tag{9}$$

one possibility is to integrate in time using Euler's method,

$$N(\mathbf{x},t+\delta t) = N(\mathbf{x},t) + \delta t F(\mathbf{x},t). \tag{10}$$

Whilst this method is easy to implement it is relatively inaccurate, even if small time-steps (δt) are used. A better option is to use a higher order scheme which is relatively memory efficient, we use the so called $2N$ Runge-Kutta schemes. These schemes use an iterative formula,

$$w_i = \alpha_i w_{i-1} + \delta t\, F(t_{i-1}, N_{i-1}), \quad N_i = N_{i-1} + \beta_i w_i, \tag{11}$$

where w is a helper array in the code of the same dimensions (without ghost zones) as N. In the code we use a three-step scheme, hence $i = 1, ..., 3$. In order to advance the variable N from $N^{(n)}$ at time $t^{(n)}$ to $N^{(n+1)}$ at time $t^{(n+1)} = t^{(n)} + \delta t$ we set in Eq.11

$$N_0 = N^{(n)} \quad \text{and} \quad N^{(n+1)} = N_3, \tag{12}$$

with N_1 and N_2 being intermediate steps. In order to be able to calculate the first step, $i = 1$, we require $\alpha_1 = 0$, we follow [2] and set $\alpha_2 = -2/3$, $\alpha_3 = -1$, $\beta_1 = 1/3$, $\beta_2 = 1$, and $\beta_3 = 1/2$. For testing purposes one can easily change a flag in *cparam.local* to implement Euler's scheme. The routines which carry out these steps can be found in the subroutines *get_rk3* and *get_euler* which are found in the main code *run.f90*.

3.3 Boundary Conditions

Two possible boundary conditions can be implemented in the code at present, the first is absorbing boundary conditions,

$$N(0,y) = N(L,y) = 0, \qquad 0 \le x \le L. \tag{13}$$

We also have the option of setting zero flux conditions at each of the boundaries such that,

$$\frac{\partial N(0,y)}{\partial x} = \frac{\partial N(L,y)}{\partial x} = 0. \tag{14}$$

Of course both conditions are implemented in the y direction as well, note at the moment it is not possible to have different boundary conditions in x and y. Because we use 6^{th} order derivatives in the code the main array has an extra 3 mesh points at the start and end of each component, commonly referred to as 'ghost' zones. This is important when considering the use of MPI, which we discuss below. To change between these conditions simply alter the parameter *boundary* in *run.in*. The boundary conditions are set in the routine *get_ghost_zones*.

3.4 MPI

In order to achieve much higher spatial resolution the code implements Message Passing Interface (MPI) routines, enabling it to run on many processes[1] in parallel. Note that one executable file instructs all the processes, hence it is classified as a Single Program Multiple Data (SPMD) code. The parallel environment consists of a number (*nprocs*) of processes, where each process can be uniquely identified by its rank. The numerical domain as a whole is divided into *nprocs* smaller sub-domains and each process is allocated one of these smaller arrays for which it is responsible. Interprocess communication is via the routines of the Message Passing Interface (MPI). Possibly the most important MPI routine in a finite difference code, such as ours, is the communication of boundaries, which must be done each time the array for N is updated. In our code ghost zones are calculated on each process and then passed to the appropriate neighbouring processes in the routine *get_ghost_zones*, which is located in the module *mpi_routines.f90*. Also in this module are subroutines which set up the initial conditions and determine the configuration of the processes used in the simulation. For further information we refer the reader to the following documentation `http://www.redbooks.ibm.com/abstracts/sg245380.html`.

[1]Note the use of the term 'process' rather than 'processor', as a single processor can accommodate multiple processes.

4 Running the code

Parameters which can be changed without recompilation are found in *run.in*. In this file you can change the number of time-steps to take (*nsteps*) and how often to print to file (*shots*). The initial and boundary conditions are also set here as character parameters, they accept the following options.

The initial conditions for N - *initial_config*

benchmark_sine	$sin(\pi y/L)$, $0 \leq y \leq L$
gaussian1	gaussian centred in box
gaussian2	same as above but with smaller variance
gaussian3	gaussian 2 repeated in opposite corners
jericho	gaussian centered at the historic sight of Jericho (31.8^o N, 35.5^o E)

The initial conditions for ν - *initial_nu*

constant	ν is set to a constant value given by *nu_mag*.
gaussian1	ν increases linearly from 0 to *nu_mag* along y/θ.
global	Reads in data from *global/nu.dat* to set realistic diffusivity.

The initial conditions for K - *initial_k*

constant	The carrying capacity K is set to a constant value given by *kay*.
global	Reads in data from *global/K.dat* to set realistic carrying capacity.

Boundary conditions - *boundary*

absorbing	absorbing boundaries see Eq. 13
zero_flux	zero flux boundaries see Eq. 14

The initial conditions for the Lagrangian front - *lagrangian_config*

centre	A loop in the centre of the box
jericho	A loop centred at Jericho, needs the initial conditions for N to match

You can find more about the global initial conditions later in the manual in §6. One thing to note is that the mesh size (with global conditions only) is not free to choose any longer and unless you match the dimensions found in *globe/globe_dims.log*, the code will exit. For more on the Lagrangian mode see §7 Parameters which, when changed, force a recompile are found in *cparam.local*. These include mesh size, domain size, what order of derivative to use and the coordinate system. The module *cdata.f90* holds many of the key variables globally. Of particular attention is the main array f, which is a derived type and holds the population density (N), diffusivty (ν) and carrying capacity (K). Finally in *cdata.f90* is the routine *read_run_file* which reads in the value of the variables in *run.in*. Before commencing a run make sure there is a data directory (you could run *mkdir data*, however it is probably better to create a symbolic link to a directory which is not on a network disk). To compile and run the code simply type:

```
> ./run.sh -c <N>
```

where <N> is the number of processes you wish to run on, for more options run,

```
> ./run.sh --help
```

The code can be compiled without running using the following command,

```
> make
```

to tidy up, run:

```
> make clean
```

It is highly recommended to inspect the Makefile and *run.sh* before running the code.

5 MATLAB:Visualisation

MATLAB is used to visualise the data and a variety of scripts are located in the directory *bin*. The code outputs data from each process into a directory *data/proc<N>* as well as dimensions information in *data/dims.dat*, the MATLAB scripts *start.m* and *gather.m* will extract this data and create one large array. Before running any of the commands outlined below add the *bin* directory to the MATLAB path using,

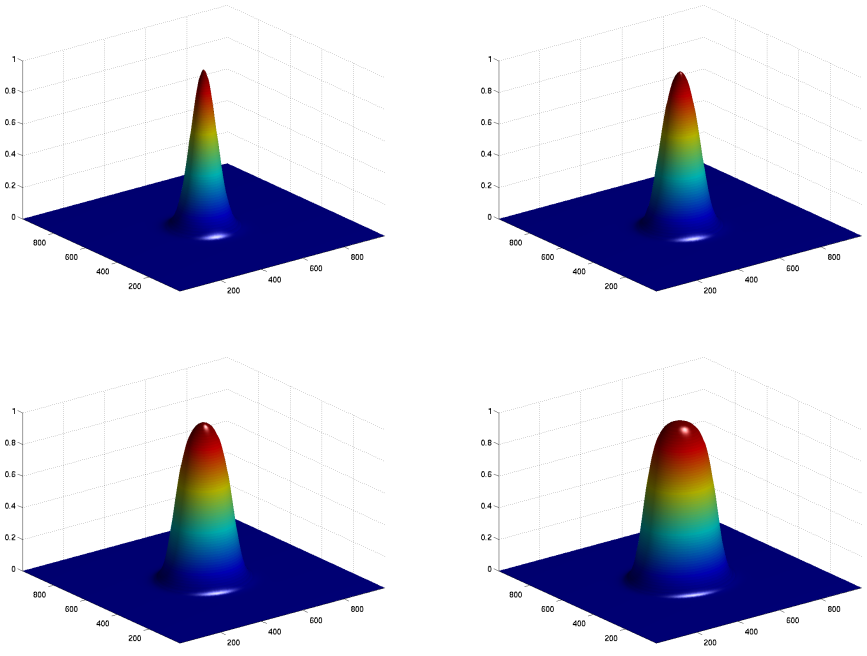

Figure 1: A numerical solution to the FKPP equation, Eq. 1, whose initial condition for $N(\mathbf{x}, t)$ is gaussian. Shown here are surface plots of N at increasing times, showing the formation and then propagation of a travelling wave. Parameters for the simulation are $\nu = 5\text{km}^2/\text{year}$ and $\gamma = 0.05\text{year}^{-1}$, we confirm that the speed of the travelling wavefront is indeed $\sqrt{1/2}\text{km/year}$

```
> addpath ./bin
```

Other scripts will then act on this array, for example to create a surface plot run the following in MATLAB:

```
> start
> gather(1)
> hsurf
```

Note the script *gather* accepts an argument which is the data file you want to extract from. Figure 1 shows the results of a FKPP simulation with an initial gaussian profile for N ,note the propagation of a travelling wave. The

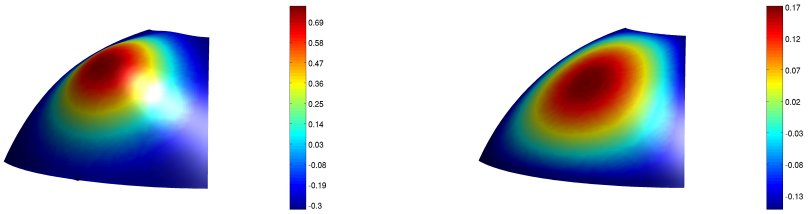

Figure 2: A numerical solution to the heat equation, Eq. 1 with $\gamma = 0$, in spherical geometry at a fixed radius. The initial condition for $N(\mathbf{x}, t)$ is gaussian. Shown here are surface plots of N at the start of the simulation (left) and later on in the run (right).

dimensions information (*dims.dat*) also lets the MATLAB scripts know the geometry of the system and all the routines will adapt to spherical geometry, see Fig. 2. For 'global' simulations (when the topographic data is used) it is perhaps better to run,

```
> start
> gather(1)
> neo_colour(1)
```

and if we want to plot the results of a Lagrangian simulation (overlayed on top of a contour plot of the carrying capcity) run,

```
> start
> front(1)
```

where (as with gather) the number arguement is the data-file you wish to plot.

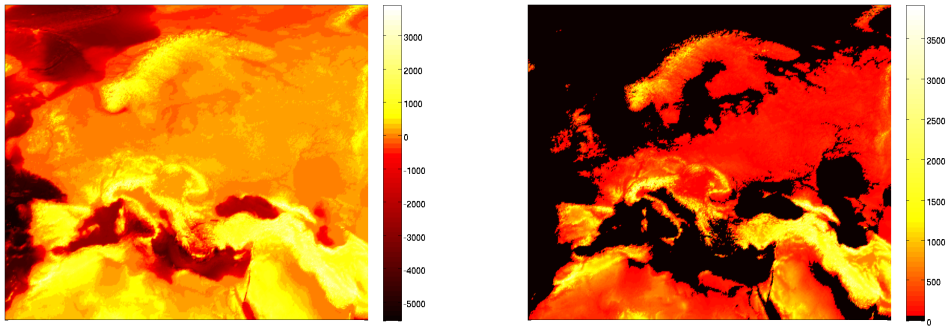

Figure 3: The left panel shows the total altitude data obtained from the NGDC, the right panel shows the correspoding data, once the seas have been identified.

6 Environmental factors

Perhaps the most important aspect of the code is to be able to solve the FKPP equation with parameters that depend on the topography of the region of the Earth that you are focusing on. Firstly, one must obtain the data and currently we use NOAA's National Geophysical Data Center (NGDC). Once the data has been obtained there is are a collection of MATALB scripts located in *bin/globe*. All topographic data is held in the *globe* directory. In the top level of the directory we store the data that is used by the code. The directory *globe_altitude* holds a text file with from NOAA with latitiude, longitude and altitude data, and a header file which gives the dimension of the array. The directory *globe_river* holds coast and river data which we describe in more detail in §6.1. To produce the spacially dependent variables ν and K use the script *globe_altitude.m*, which will load in the data and perform the required manipulation. For the techincal details we refer you to the script, here we shall outline the manipulations The first step is to identify the seas (which we set as any altitude less than 0), see Fig. 3. We then adjust the data to take into account mountain ranges and adjust both the carrying capacity and diffusivity (note the only population we track at the moment is farmers), such that,

$$K, \nu \propto \frac{1}{2} - \frac{1}{2} \tanh\left(\frac{a - 1000m}{100m}\right), \tag{15}$$

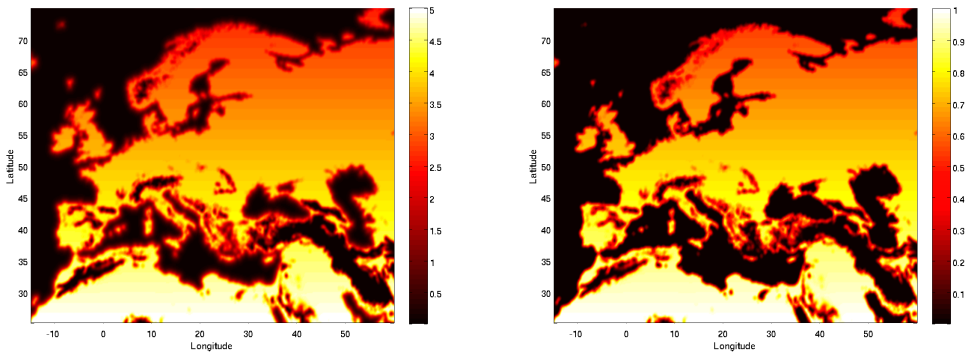

Figure 4: The left panel shows the adjusted diffusivity based on the altitude data, and the right panel is the carrying capacity. Note the region of the Earth featured runs from a latitude of 25^oN to 75^oN and a longitude of 15^oW to 60^oE.

where a is the altitude. At low altitude ($a \sim 0$) we must also adjust both K and ν. For K we again use a tanh function such that,

$$K \propto \frac{1}{2} \tanh\left(\frac{-a}{50m}\right) - \frac{1}{2}. \tag{16}$$

For the diffusivity we must take into the account the fact that people can travel a short distance offshore, and set $\nu \propto \exp(-d/l)$, where d is the distance from land and $l = 40$km. The final step is to account for latitude. Note there is also the option to smooth the data in the code, this is a very local smoothing and can be found in the *smoothc.m* script. There is significant evidence [4] that the latitude had a bearing on the productivity of the landscape, due to a decrease in temperture as latitude increases, on average. The precise form of the latitudinal dependence is not know, so we set both the diffusivity and the carrying capacity decrease linearly, for more details see the MATLAB script. Figure 4 shows the subsequent diffusivity and carrying capacity upon carrying out the described adjustments.

6.1 Waterways

To produce velocity vectors which represent the enchanced mobility of a population we use the high resolution World Data Bank (WDB) coastal and

river data. This manifests itself as an advective velocity (V) in Eq. 1. The WDB data is ideal as it is vector data which come in segements with a rank determined by the importance of the feature (see the README file in *globe/globe_river*) The matlab scripts *global_river.m and global_coasts.m* maps this data onto a mesh (size defined by the altitude data) using a distanced weighted average over 10km. These vectors are still to implement the Fortran code.

6.2 A few results

In this section we will document results from the simulations which include the topographical features. Figure 6 shows some very early results from the code.

7 Lagrangian Mode

It is well known that solutions to the FKPP equation,

$$\frac{\partial N(\mathbf{x},t)}{\partial t} + (V\cdot\nabla)N = \gamma N\left(1-\frac{N}{K}\right) + \nabla\cdot(\nu\nabla N). \tag{17}$$

develop a propagation front, which advances at a easily determined speed. In two dimensions it can be shown that the speed of the travelling wave solution is given by $\sqrt{2\gamma\nu}$, and hence is independent of the carrying capacity. If one is only interested in the time of first arrival of the population then a great deal of computational time is spent numerically integrating the PDE, in areas that are of no interest (i. e. the first arrival front has already passed through). This is the motivation for the Lagrangian front mode of the code. We use an adaptive Lagrangian mesh made up of particles, which are the numerical discretisation of a continuous line represent the first arrival front. The numerical methods for adapting the mesh are very similar to those outlined in [1]. Particles are advected at a speed determined by the local diffusivity (using linear interpolation to the four nearest grid points), global growth rate, and in a direction that is always the outward normal to the propagating front,

$$\frac{d\mathbf{x}}{dt} = \sqrt{2\gamma\nu(\mathbf{x})}\hat{\mathbf{n}}. \tag{18}$$

The time integration is performed using Euler's method, purely for speed. If the advancing front splits around an object (for example a mountain range)

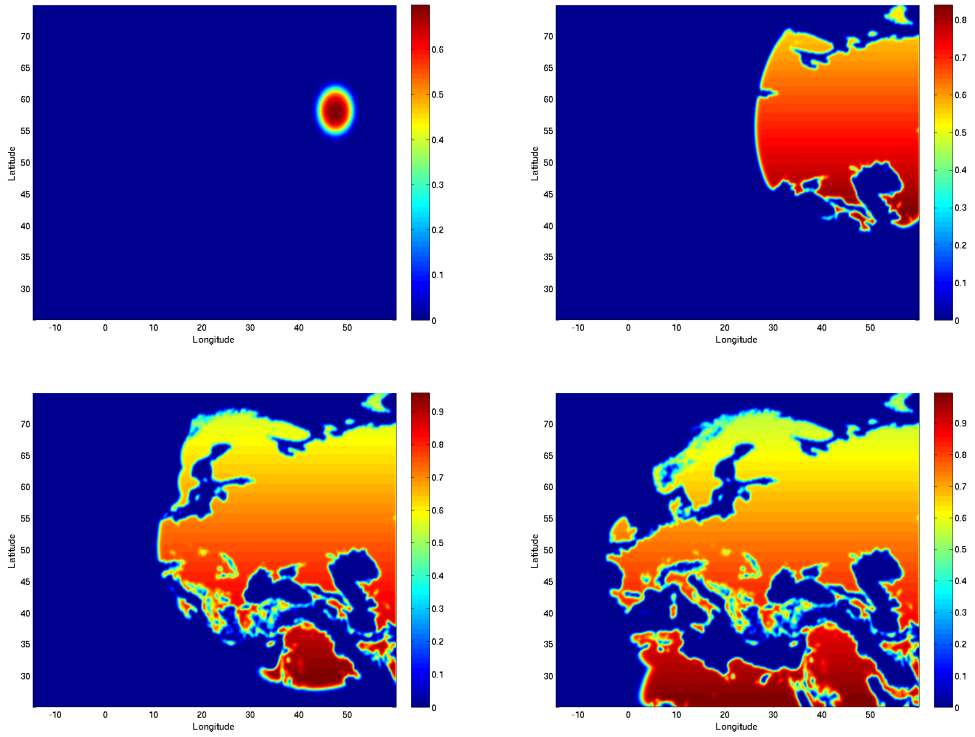

Figure 5: Early results from the code showing the spread of the Neolithic farmers across Europe.

and then comes back together at the other side of the obstacle then from the solution of the PDE it is clear that the two fronts would simply merge together. To achieve similar results in the Lagrangian scheme we allow separate sections of the front to 'reconnect' if they become very close. This can be thought of as splicing the two fronts together, again for more details on the numerical algorithms used to do this either consult the code or [1]. Note Lagrangian mode is fully compatible with both coordinate systems used in the code, and has a range of initial conditions which can be set in *run.in* see §4. Below we show snapshots of a simulation in cartesian coordinates, in compartison to the solution of the PDE, with an initially gaussian distribution in the centre of the box, the Lagrangian front accuratley maps the propagating front, where there are orders of magnitude separating the speeds of the two methods.

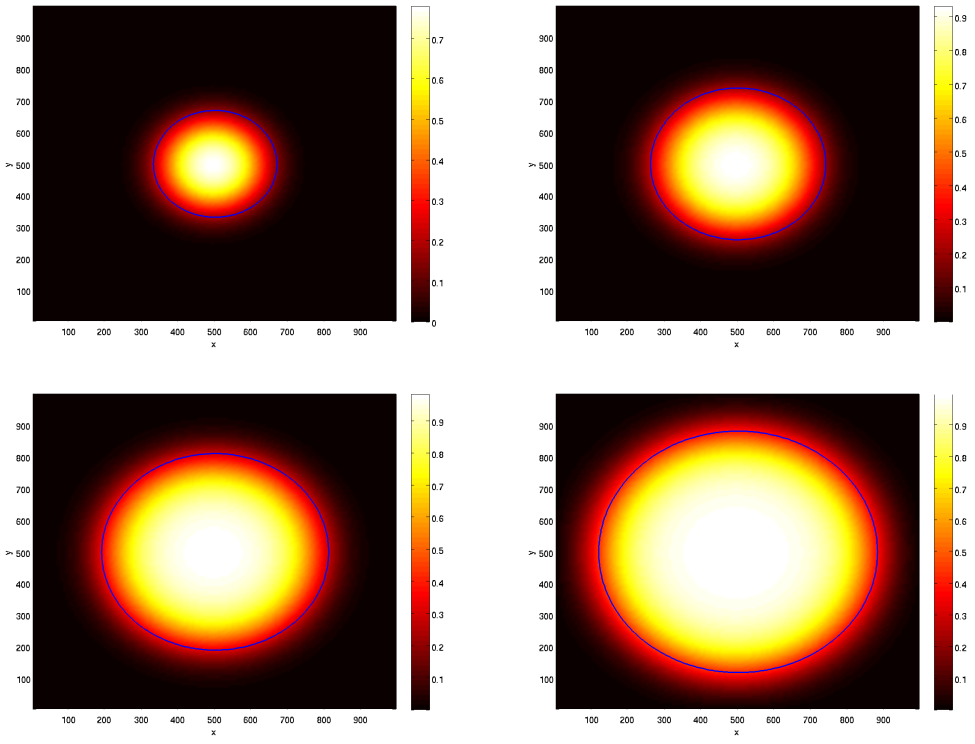

Figure 6: A comaprison of the Lagrangian method (shown in the thick blue line) to the results of solving the FKPP equation on a mesh.

A Benchmarking The Code

In this appendix we consider both the accuracy of the code and the scaling of the time to run with the number of processors.

A.1 Comparison with Heat Equation

If $\gamma = 0$ Eq. 1 reduces to the heat equation,

$$\frac{\partial \Phi(\mathbf{x}, t)}{\partial t} = \kappa \nabla^2 \Phi, \tag{19}$$

which in 1D reduces to,

$$\frac{\partial \Phi(x, t)}{\partial t} = \kappa \frac{\partial^2 \Phi}{\partial x^2}. \tag{20}$$

With absorbing boundary conditions $\Phi(0, t) = \Phi(L, t) = 0$, where $0 \leq x \leq L$, and an initial condition $\Phi(x, 0) = sin(\pi x/L)$, the exact solution can be shown to be

$$\Phi(\mathbf{x}, t) = \sin(\pi x/L) \exp\left(-\frac{\kappa \pi^2 t}{L^2}\right) \tag{21}$$

Figure A.1 shows the comparison between the numeric and analytic solutions in two different simulations, one at low resolution and one at a higher resolution, see the figure caption for more details. We also used these simulations to confirm that our parallel implementation is correct.

B Code outline

run.f90 - The main code, this should be kept as short as possible to allow one to view the structure of the code, if editing please put all subroutines called in a relevant (or new) module and comment next to each subroutine called which module it belongs to. We now list the role of each module in the order that they are called in run.f90.
mpi_routines.f90 - This module sets up the run for parallel computations. At present the code needs to be compiled with mpif90 in order to run, which, while not elegant, is an easier job to program. Also in this module is the ghostzone routine which at present only passes N (the only dynamically changing quantity) at every timestep. Major additions to the code will warrant a modification to this module, however to most extents it should be

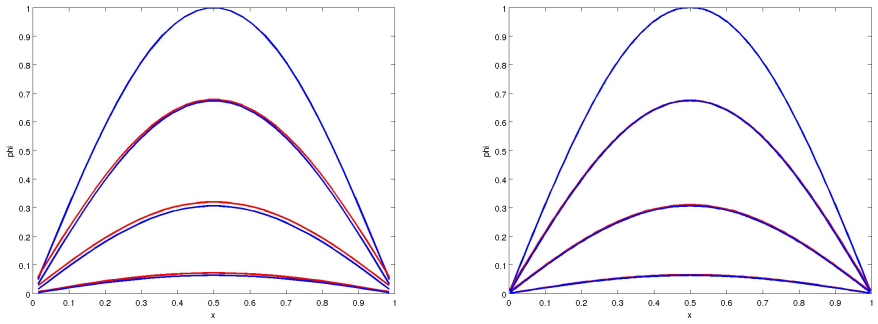

Figure 7: 1D simulations of the heat equation (Eq. 1 with $\gamma = 0$) with numerical results shown in red, and the exact solution shown in blue. The left panel uses 32 mesh points, $\delta x \sim 0.03$, and the simulation shown in the right panel uses 128 mesh points giving $\delta x \sim 0.008$. Note that the error from the 6^{th} order second derivative is $\mathcal{O}(\delta x^6)$. In both simulations $\eta = 1.$ and δt is the maximum acceptable value given by the CFL condition of the higher resolution run.

considered a black box.

init.f90 - The longest and most complex module in many respects. The routines here are called once at the start of the code, and set up the initial conditions for the variables (N, ν and K, as well as the Lagrangian front) depending on the parameters in *cparam.local* and *run.in*. Because of the need for the code to read in altitude and river/coast data the method employed is to generate a global array for each variable on each process and then to divide this up according to process domains. If this module continues to grow it may be sensible to split into two separate modules for different variables.

derivatives.f90 - This module contains all the routines used to take derivatives when integrating the FKPP PDE. The code can compute second, fourth and sixth order derivatives depending on the parameters in *cparam.local*. Also contained in this routine is the rhs routine which puts these derivatives together depending on the geometry of the problem (spherical or Cartesian) again set in *cparam.local*. Finally the unit normal vector used to advect the Lagrangian particles is located in this module, as whilst technically not a derivative it is very similar!

cdata.f90 - The purpose of this module is to hold all the main variables

globally in order to avoid very long subroutine passes, which inevitably lead to mistakes! Also located in this module is the subroutine which reads in the parameters file *run.in* at runtime, avoiding the need to recompile the code to change certain parameters.
timestep.f90 -
output.f90 -
lagrangian_routines.f90 -

B.1 Other Files

cparam.local - A parameters file which is used at compile time, essentially any change of parameters in this file forces a recompile of the code. The parameters include grid size, derivative order and geometry.
run.in - A parameters file which is used at runtime, meaning no recompile. These parameters include background values for ν, K and γ, as well as the initial conditions for the simulation.

C Still To Do

The following list is areas of the code that still need to be worked on,

- Implement waterways in the code,

References

[1] A. W. Baggaley, C. F. Barenghi, A. Shukurov, and K. Subramanian. Reconnecting flux-rope dynamo. *Phys. Rev. E*, 80(5):055301, Nov 2009.

[2] A. Brandenburg. *Computational aspects of astrophysical MHD and turbulence*, pages 269–+. 2003.

[3] T. Czárán. *Spatiotemporal models of population and community dynamics*. Springer, 1998.

[4] J. Steele, M. Gkiasta, and S. Shennan. The neolithic transition and european population history a response. *Antiquity*, 78:711–713, 2004.

[5] J. H. Williamson. Low-Storage Runge-Kutta Schemes. *Journal of Computational Physics*, 35:48–+, Mar. 1980.