Simulations of Knots in Excitable Media

Carl Whitfield

Department of Physics, University of Warwick, Coventry, United Kingdom, CV3 7AL

(Dated: January 3, 2017)

A summary of (and user guide for) simulations of knotted vortex lines in a Fitzhugh-Nagumo reaction-diffusion system in C++. I first introduce the problem briefly by discussing relevant background before detailing the contents of the code and outlining some preliminary results.

CONTENTS

I. BACKGROUND

A. Excitable Media and the FitzHugh-Nagumo Model

An excitable medium is one where the individual constituents can change state due to stimulation by an external cue, such as a chemical concentration. This covers a wide range of systems from biological tissues to wildfires. Of particular interest are excitable media that recover (i.e. can be re-excited), which include many biological examples as well as some chemical reactions. A simple model of the continuum behaviour of such a system is given by the FitzHugh-Nagumo model equations for fields u and v

$$\frac{\partial u}{\partial t} = \frac{1}{\epsilon}(u - u^3/3 - v) + \nabla^2 u \tag{1}$$

$$\frac{\partial v}{\partial t} = \epsilon(u + \beta - \gamma v)\,. \tag{2}$$

Originally, the model was used to describe axon activity where u is the membrane potential and v a recovery variable [**? ?**] and did not include the diffusion term. In the absence of diffusion the equations are a generalised form of the van der Pol oscillator with amplitude of driving ϵ. The inclusion of the diffusion term couples neighbouring oscillators allows travelling wave solutions. Thus this system of equations is commonly used to describe phenomena in various reaction diffusion systems including spiral waves in cardiac tissue.

II. NUMERICAL SIMULATION CODE

A. Outline

Generating knotted vortex lines in the FitHugh-Nagumo equations requires formulating initial conditions with the correct topology and geometry. This is achieved using a continuous circle-valued function Φ with a singularity located along the knotted curve K that we wish to initialise, around which Φ winds by 2π. The initial conditions for u and v are then given by [**?**]

$$u_0 = \cos\Phi - 0.4 \tag{3}$$

$$v_0 = \sin\Phi - 0.4\,. \tag{4}$$

The offset -0.4 is the approximate value of u and v at the centre of the scroll wave core, and approximately the average value about which u and v oscillate. Not only does this mean that locally around the filament, the u and v fields resemble a scroll wave (figure ...) but also they have matching topology. Wavefronts of u and v form orientable surfaces bounded only by the knot curve (Seifert surfaces). Similarly, the 'function' Φ foliates all of space with a family of Seifert surfaces, therefore the wavefronts of u_0 and v_0 are also Seifert surfaces.

The phase Φ is calculated as the scalar potential of a curl-free magnetic field (or equivalently an irrotational flow in a inviscid fluid) around a current carrying wire K with position vector $\boldsymbol{l}$ via the Biot-Savart law:

$$\Phi = \int_{\boldsymbol{r}_0}^{\boldsymbol{r}} \boldsymbol{B}(\boldsymbol{r}').\mathrm{d}\boldsymbol{r}' \tag{5}$$

where

$$\boldsymbol{B}(\boldsymbol{r}) = \frac{1}{2}\int_K \frac{(\boldsymbol{r}-\boldsymbol{l})\times \mathrm{d}\boldsymbol{l}}{|\boldsymbol{r}-\boldsymbol{l}|^3}\,. \tag{6}$$

Thus, our chosen Φ is harmonic as well as having the desired topological properties. By Stokes' theorem, the integral can be rewritten in terms of a surface integral

$$\Phi = \frac{1}{2}\int_S \frac{(\boldsymbol{l}-\boldsymbol{r})}{|\boldsymbol{l}-\boldsymbol{r}|^3}\cdot \mathrm{d}\boldsymbol{S}\,. \tag{7}$$

On initialisation the simulation code defines the grid of points discretising the three-dimensional domain to be simulated. The size of the box in the x direction is defined by `size` (L) and the number of points discretising this direction by `Nx` (N_x) such that the spatial step size $h = L/(N_x - 1)$. The size of the box in y and z is then just $N_y h$ and $N_z h$ so that the grid spacing is uniform in all directions. Arrays defining quantities on this grid (*e.g.* `phi`) are one dimensional, with entries $n = N_z(N_y i + j) + k$ where i, j, k are the indices for the x, y, z coordinates respectively. Usually, loops over the whole array are of the format

```
for(i=0;i<Nx;i++)
{
      for(j=0; j<Ny; j++)
      {
          for(k=0; k<Nz; k++)
          {
               n = pt(i,j,k);
               .....
          }
     }
}
```

where the inline function `pt` simply returns the one-dimensional index n. Henceforth we will use subscript n or i, j, k to denote array indices on the three-dimensional grid, depending on the context.

B. Initialisation Method I: Surface Integral

Seifert surfaces have been created by hand using Surface Evolver to triangulate and refine the surface with a fixed boundary curve (generally obtained from KnotPlot). These are stored as `.stl` files, which simply list the properties of each triangular face in the following format

```
facet normal nx ny nz
   outer loop
     vertex v1x v1y v1z
     vertex v2x v2y v2z
     vertex v3x v3y v3z
   endloop
  endfacet
```

where `ni` is the ith component of the (normalised) vector normal to the face and `vij` is the jth component of the position vector of the ith vertex (all given in floating point format). This is read by the C++ code via the function `init_from_surface_file`. This currently requires only 2 variables as input, the filename of the `.stl` file and a pointer to a vector of the structure `triangle` (both of which are declared as a global variables in the code). The structure `triangle` is defined

```
struct triangle
{
    double xvertex[3];
    double yvertex[3];
    double zvertex[3];
    double normal[3];
    double area;
    double centre[3];
};
```

such that `xvertex` stores the x-components of the 3 vertices, `normal` stores the components of the normal vector, `area` stores the area of the triangle and `centre` the components of the centroid of the triangle.

The function `init_from_surface_file` reads in the values for the vertices and normals until it reaches the `endsolid` tag at the end of the `.stl` file, storing the total number of faces in N_K (`NK` in the code). Generally the index used to denote the triangle is `s` in the code, so we will use subscript s to indicate quantities of the triangle. The centroid is then simply calculated as the mean of the vertices. The function also records the maximum and minimum x, y and z coordinates of vertices on the surface as `minxin` ($x^{\min}$), `maxxin` ($x^{\max}$) *etc.* All surface points $\boldsymbol{r}_s$ are then transformed

$$r_s^{(x)} = \frac{X^{\max}}{x^{\max} - x^{\min}} \left[r_s^{(x)} - \frac{1}{2}\left(x^{\max} + x^{\min}\right) \right] \tag{8}$$

$$r_s^{(y)} = \frac{Y^{\max}}{y^{\max} - y^{\min}} \left[r_s^{(y)} - \frac{1}{2}\left(y^{\max} + y^{\min}\right) \right] \tag{9}$$

$$r_s^{(z)} = \frac{Z^{\max}}{z^{\max} - z^{\min}} \left[r_s^{(z)} - \frac{1}{2}\left(z^{\max} + z^{\min}\right) \right] \tag{10}$$

where ($X^{\max}$, $Y^{\max}$, $Z^{\max}$) are the user defined bounds for the knot size (`xmax`,`ymax`,`zmax`). The surface normal is then scaled accordingly:

$$n_s^{(x)} = n_s^{(x)} \frac{Y^{\max} Z^{\max}}{\left(y^{\max} - y^{\min}\right)\left(z^{\max} - z^{\min}\right)} \tag{11}$$

$$n_s^{(y)} = n_s^{(y)} \frac{X^{\max} Z^{\max}}{\left(x^{\max} - x^{\min}\right)\left(z^{\max} - z^{\min}\right)} \tag{12}$$

$$n_s^{(z)} = n_s^{(z)} \frac{X^{\max} Y^{\max}}{\left(x^{\max} - x^{\min}\right)\left(y^{\max} - y^{\min}\right)} \tag{13}$$

and then normalised. Finally, the area of triangle n is calculated from the vertices (Heron's formula):

$$A_s = \sqrt{p(p - r_{21})(p - r_{20})(p - r_{10})} \tag{14}$$

where $p = (r_{21} + r_{20} + r_{10})/2$ and $r_{ij} = |\boldsymbol{r}_i - \boldsymbol{r}_j|$ and $\boldsymbol{r}_i$ are the vertices of the triangles.

The code next calls `initialise_knot` which in turn calls `phi_calc` where the phase field Φ is calculated at every point in the domain. The surface integral of equation (7) is approximated by

$$\Phi_n = \sum_{s=0}^{N_K} \frac{(\boldsymbol{c}_s - \boldsymbol{x}_n)}{|\boldsymbol{c}_s - \boldsymbol{x}_n|^3} \cdot \hat{\boldsymbol{n}}_s A_s \tag{15}$$

where $\boldsymbol{c}_s$ is the position vector of the centre of triangle s. Φ_n is calculated for all n, but contributions to the sum in (15) where $|\boldsymbol{c}_s - \boldsymbol{x}_n| = 0$ are skipped. Then, as Φ is circle valued we redefine it to lie in the range $\Phi \in [-\pi, \pi]$. The resulting Φ field should be a foliation of the three-dimensional domain with the knot as its boundary, an example plot of level-sets of Φ is given in figure 1.

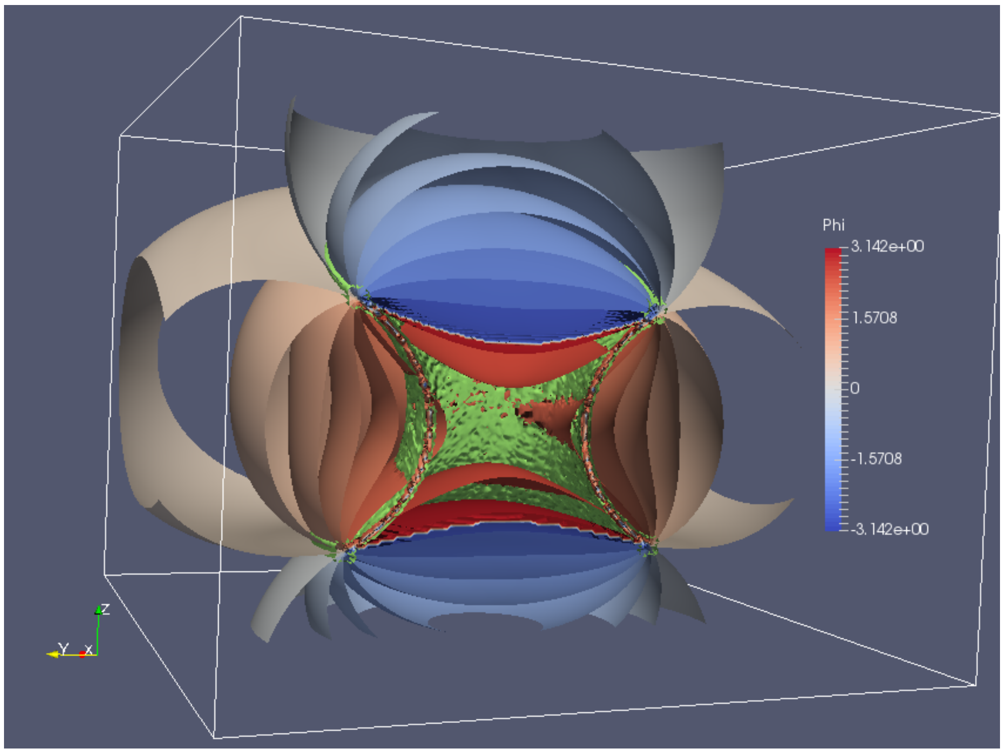

FIG. 1. Level-set surfaces of the function Φ sliced along the y direction for visibility. The green surface shows the input surface (which has an unlink as its boundary).

C. Initialisation Method II: Line Integral

The option `FROM_KNOT_FILE` indicates that the phase field ϕ should be calculated by line integral via the function `init_from_knot_file`. This takes a `.txt` file with a list of point coordinates as input for the knot curve storing them in the vectors `px`, `py`, `pz` and and also storing the maximum and minimum values of the x, y and z components in this list as `minxin`, `maxxin` *etc.* The knot curve position vector (xt, yt, zt) is then scaled accordingly as in equations (8), (9) and (10).

This curve is then further subdivided by linear interpolation between the points to have $N_k = 2L/h$ (rounded to nearest integer) points, stored in the (vector) arrays `X`, `Y` and `Z`. We refer to the position vector of each point as $\boldsymbol{l}_s = (X_s, Y_s, Z_s)$.

The magnetic field is then computed by the function `B_field_calc` called by `initial_cond` at all points on an $N_x \times N_y \times N_z$ grid by summing this quantity over all the points on the loop $s = (1..N_k)$ according to equation (6):

$$\boldsymbol{B}_{i,j,k} = \sum_{s=1}^{N_k} \frac{(\boldsymbol{x}_{i,j,k} - \boldsymbol{l}_s) \times \mathrm{d}\boldsymbol{l}_s}{|\boldsymbol{r}_{i,j,k} - \boldsymbol{l}_s|^3} \tag{16}$$

where $d\boldsymbol{l}_s = (d\boldsymbol{l}_{s+1} - d\boldsymbol{l}_{s-1})/2$.

Finally the scalar potential Φ is computed at all points by the path integral in equation (5) for all points. We choose the initial point $\boldsymbol{r}_0$ to be the origin of our coordinate system and set $\phi = 0$ at $\boldsymbol{r} = \boldsymbol{r}_0$. To save computation time, we first calculate the path integral on the vertices of the 3-dimensional domain filling in all the values of Φ on the chosen path in the process, and looping inwards (skipping points that are already filled). **This integral could perhaps be sped up more by looping over the faces of the domain first in this manner and working inwards, as this should assign most points in the domain relatively quickly.** Given a path of N_p gridpoints (i_p, j_p, k_p) connecting $\boldsymbol{r}_0$ to the destination point $\boldsymbol{r}$, Φ is computed iteratively along the path as:

$$\Phi_{i_p,j_p,k_p} = \Phi_{i_{p-1},j_{p-1},k_{p-1}} + \frac{1}{2}(\boldsymbol{B}_{i_p,j_p,k_p} + \boldsymbol{B}_{i_{p-1},j_{p-1},k_{p-1}}) \cdot (\boldsymbol{r}_{i_p,j_p,k_p} - \boldsymbol{r}_{i_{p-1},j_{p-1},k_{p-1}}) \tag{17}$$

for $p \in [1 : N_p - 1]$ where $\Phi_{0,0,0} = 0$ and $(i_0, j_0, k_0) = 0$.

The path from the origin to the point in question is found by the function `pathfind`. It was found that simple straight-line paths can give rise to significant numerical errors resulting in incorrect initialisation when the paths come within some small neighbourhood of the knot curve (of approximately one ‘core diameter’ $d_0 = \lambda/\pi$). To avoid these points we fill an empty array `ignore` (or I) with the value 1 at every gridpoint where $|\boldsymbol{r}_{i,j,k} - \boldsymbol{l}_s| < d_0$ for any $s = 0..N_k - 1$ of the initial curve (this array is computed once during the magnetic field calculation).

In order to compute the path itself from $\boldsymbol{i}_0 = (i_0, j_0, k_0)$ to $\boldsymbol{i}_f = (i_f, j_f, k_f)$ we iteratively choose neighbouring gridpoints that are not forbidden. At any point on the path point $\boldsymbol{i}_s$ (so long as $(i_s, j_s, k_s) \neq (i_f, j_f, k_f)$, otherwise the pathfinding is complete) we define a preferred direction $\boldsymbol{p} = (\mathrm{sgn}(i_f - i_s), \mathrm{sgn}(j_f - j_s), \mathrm{sgn}(k_f - k_s))$, then if the point corresponding to $\boldsymbol{i}_s + \boldsymbol{p}$ (where $\boldsymbol{i}_s = (i_s, j_s, k_s)$) is free to move into, this becomes the next point in the path. Note that a path is forbidden from revisiting a previous point, each time a point is visited the zero array `track` (or T) is changed so that $T_{i_s,j_s,k_s} = 1$. If, this point is blocked ($I_{\boldsymbol{i}_s+\boldsymbol{p}} = 1$ or $T_{\boldsymbol{i}_s+\boldsymbol{p}} = 1$) then instead we sample all neighbouring points and choose the one that maximises a particular weight function W. We choose W to be the sum of the alignment with direction to final point and alignment to the magnetic field (which helps to avoid the path spiralling around when it hits an obstacle) as follows,

$$W = \left(\frac{\boldsymbol{i}_f - \boldsymbol{i}_{s'}}{|\boldsymbol{i}_f - \boldsymbol{i}_{s'}|} + \frac{\boldsymbol{B}_{i_{s'},j_{s'},k_{s'}}}{|\boldsymbol{B}_{i_{s'},j_{s'},k_{s'}}|} \right) \cdot \frac{\boldsymbol{i}_{s'} - \boldsymbol{i}_s}{|\boldsymbol{i}_{s'} - \boldsymbol{i}_s|}\,, \tag{18}$$

where s' denotes the neighbouring point in question. If no neighbour points are viable, we step back to the previous point on the path and repeat the previous step, now with the dead-end point unable to be revisited as $T = 1$ at that point. This process is continued until the path is complete or it returns to the origin or the number of points in the path exceeds a large value, in which case the algorithm quits and fails to perform a path integral to this point.

D. Initialisation from file

There is also the option to initialise a simulation from a previous run of the code. One can initialise from the `phi.vtk` file outputted after calculation of the Φ field using the option `FROM_PHI_FILE` and defining the string variable `B_filename` as the local path to the input `.vtk` file. This `phi.vtk` file must be from a previous run of the code with the same grid size and spacing settings.

Similarly, one can continue from the Fitzhugh-Nagumo variables u and v using option `FROM_UV_FILE` and setting `B_filename` as the local path to the input `.vtk` file. Again, this `uv_plotXXX.vtk` file should be from a previous run of the code with the same grid size and spacing settings. In this case, it may be desirable to start the time counter from the last point of the previous run of the code, in which case set `starttime` = XXX. Otherwise `starttime` should be set to zero.

III. FITZHUGH-NAGUMO DYNAMICS

Once Φ is calculated the Fitzhugh-Nagumo variables are initialised in `uv_initialise` as given by equations (3) and (4). If initialisation method II is used (`FROM_KNOT_FILE`) then u and v are set to -0.4 in the regions that are skipped (as this is the approximate value they take at the centre of the scroll wave:

```
for(n=0; n<Nx*Ny*Nz; n++)
{
    u[n] = (2*cos(phi[n]) - 0.4);
    v[n] = (sin(phi[n]) - 0.4);
    if(option==FROM_KNOT_FILE && missed[n]==1)
    {
        u[n] = -0.4;
        v[n] = -0.4;
    }
}
```

Following initialisation the code iterates the FitzHugh-Nagumo dynamics through time with time spacing `dtime`. There are two method options in the code for this, set by the value of the preprocessor parameter `RK4`. If `RK4` is nonzero then the fourth-order Runge-Kutta method is used (function call `uv_update`), otherwise a simple Euler forward method is employed (function call `uv_update_euler`). In both cases, a simple central difference scheme is used to approximate the Lagrangian of u:

$$\nabla^2 u_{i,j,k} \approx \frac{1}{h^2}(u_{i-1,j,k} + u_{i,j-1,k} + u_{i,j,k-1} - 6u_{i,j,k} + u_{i+1,j,k} + u_{i,j+1,k} + u_{i,j,k+1})\,. \tag{19}$$

The boundary conditions are usually taken to be of the Neumann type, i.e. $\nabla u = 0$ at the wall. We take the walls to be positioned half a gridspace before the first gridpoint, and half a gridspace after the final gridpoint so that Neumann boundary conditions demand

$$u_{-1,j,k} = u_{0,j,k} \tag{20}$$

$$u_{N_x,j,k} = u_{N_x-1,j,k} \tag{21}$$

and likewise for the y and z directions. This is incorporated into the code by the function `incw(int i, int p, int N)` which is used to take increments of i, j, k around the current gridpoint. The function returns $i + p$ unless that is outside the range $[0 : N - 1]$ in which case the counter is 'reflected' back into the domain

```
inline int incw(int i, int p, int N)
{
    if(i+p<0) return (-(i+p+1));
    if(i+p>N-1) return (2*N-(i+p+1));
    return (i+p);
}
```

Currently, by setting the boolean parameter `periodic` as true then this changes the boundary conditions to periodic **in the z direction only**. Then, the periodic increment function `incp(int i, int p, int N)` is used

```
inline int incp(int i, int p, int N)
{
    if(i+p<0) return (N+i+p);
    else return ((i+p)%N);
}
```

in which case the counter continues back at 0 if it goes above $N-1$ and *vice versa*. To switch to fully periodic boundaries all occurences of `incw` should be replaced by `incp`.

1. Euler-forward update

The Lagrangian of u is stored in the array `D2u` and once this is defined everywhere u and v are updated by the discretised versions of equations (1) and (2)

$$u_{i,j,k}^{t+\delta t} = u_{i,j,k}^{t} + \delta t \left[\frac{1}{\epsilon} \left(u_{i,j,k}^{t} - \frac{1}{3}(u_{i,j,k}^{t})^3 - v_{i,j,k}^{t} \right) + \nabla^2 u_{i,j,k}^{t} \right] \tag{22}$$

$$v_{i,j,k}^{t+\delta t} = v_{i,j,k}^{t} + \epsilon \delta t \left(u_{i,j,k}^{t} + \beta - \gamma v_{i,j,k}^{t} \right) . \tag{23}$$

where the superscript indicates the time at which these quantities are defined.

2. Runge-Kutta update

In this method there is 3 extra 'correction' steps after the initial 'predictor' update which help to stabilise the dynamics at large values of δt (`dtime`). This involves 3 intermediate updates of u and v which we denote with superscripts $t+\delta t/4$, $t+\delta t/2$ and $t+3\delta t/4$ (although the steps do not correspond to these times) such that

$$u_{i,j,k}^{t+\delta t/4} = u_{i,j,k}^{t} + \frac{\delta t}{2} f\left(u_{i,j,k}^{t}, v_{i,j,k}^{t}\right) \tag{24}$$

$$u_{i,j,k}^{t+\delta t/2} = u_{i,j,k}^{t} + \frac{\delta t}{2} f\left(u_{i,j,k}^{t+\delta t/4}, v_{i,j,k}^{t+\delta t/4}\right) \tag{25}$$

$$u_{i,j,k}^{t+3\delta t/4} = u_{i,j,k}^{t} + \delta t\, f\left(u_{i,j,k}^{t+\delta t/2}, v_{i,j,k}^{t+\delta t/2}\right) \tag{26}$$

$$v_{i,j,k}^{t+\delta t/4} = v_{i,j,k}^{t} + \frac{\delta t}{2} g\left(u_{i,j,k}^{t}, v_{i,j,k}^{t}\right) \tag{27}$$

$$v_{i,j,k}^{t+\delta t/2} = v_{i,j,k}^{t} + \frac{\delta t}{2} g\left(u_{i,j,k}^{t+\delta t/4}, v_{i,j,k}^{t+\delta t/4}\right) \tag{28}$$

$$v_{i,j,k}^{t+3\delta t/4} = v_{i,j,k}^{t} + \delta t\, g\left(u_{i,j,k}^{t+\delta t/2}, v_{i,j,k}^{t+\delta t/2}\right) \tag{29}$$

where

$$f\left(u_{i,j,k}^{t}, v_{i,j,k}^{t}\right) = \frac{1}{\epsilon} \left(u_{i,j,k}^{t} - \frac{1}{3}(u_{i,j,k}^{t})^3 - v_{i,j,k}^{t} \right) + \nabla^2 u_{i,j,k}^{t} \tag{30}$$

$$g\left(u_{i,j,k}^{t}, v_{i,j,k}^{t}\right) = u_{i,j,k}^{t} + \beta - \gamma v_{i,j,k}^{t} \tag{31}$$

These operations are performed in the function `uv_add` where the return value of f and g (at the current intermediate timestep) is stored in the arrays `ku` and `kv` and then the array `kut` and `kvt` keep a running total of their contributions to the final update equations:

$$u_{i,j,k}^{t+\delta t} = u_{i,j,k}^{t} + \frac{\delta t}{6}\left[f_{i,j,k}^{t} + 2f_{i,j,k}^{t+\delta t/4} + 2f_{i,j,k}^{t+\delta t/2} + f_{i,j,k}^{t+3\delta t/4}\right] \quad (32)$$

$$v_{i,j,k}^{t+\delta t} = v_{i,j,k}^{t} + \frac{\delta t}{6}\left[g_{i,j,k}^{t} + 2g_{i,j,k}^{t+\delta t/4} + 2g_{i,j,k}^{t+\delta t/2} + g_{i,j,k}^{t+3\delta t/4}\right] \quad (33)$$

where the shorthand $f_{i,j,k}^{t} = f\left(u_{i,j,k}^{t}, v_{i,j,k}^{t}\right)$ is used. Note this method uses approximately twice as much RAM than the Euler forward method due the extra arrays that are necessary.

3. Output

The code tracks the time dependent behaviour of u and v by outputting these fields after every `skiptime` units of time. The `startime`

A. Numerical "Hummingbird"

To track the properties of the knotted vortex line we write a function which locates and traces the curve with a one-dimensional array of points, we nickname this function the "Hummingbird" due to Winfree's comment in [**?**].

IV. CONVERGENCE AND ACCURACY TESTS

V. RESULTS

VI. FUTURE DIRECTIONS AND OPEN QUESTIONS

[1] F. Maucher and P. Sutcliffe Phys. Rev. Lett. **116**(178101) (2016)