Finite Elementmethods
Lotka-Volterra equations

Finn Joel Bjervig[*]

Applied Finite Elementmentods Class,
Department of Information Technology,
Division of Scientific Computing,
Uppsala university, Sweden

December 2020

[*]Electronic address: joelfbjervig@gmail.com

1 INTRODUCTION

During the course of Applied Finite Element Methods, a project was initiated with the aim of studying the one -and two dimensional diffusive Lotka-Volterra equations, also known as the predator-prey equations, or the reaction-diffusion equations. The project consists of three parts, each with increasing complexity. These systems of PDE's describe dynamical interactions between species, one of which are predators, and the other prey. The reaction diffusion equation also describes phenomenons in chemical systems, ecology, geology and physics.

ONE DIMENSIONAL SIMPLIFICATION

A Posteriori estimate

To start things off, we consider the one dimensional differential equation

$$\begin{cases} -\delta u''(x) = f(x) \\ u(-1) = u(1) = 0 \\ x \in (-1, 1) \end{cases} \tag{1}$$

The estimated solution of u is u_h, which is computed later on using finite elements in Matlab. To measure how well the estimated solution is, the *A Posteriori error estimate* will be used. Its expression is derived as follows. Let the error be $e = u - u_h$, for which we construct the energy norm.

$$\|e\|_E^2 = \int_{-1}^{1} e'e'dx \tag{2}$$

we know that the interpolation of e is $e \approx \pi_h e$ and from the Galerkin orthogonality that the inner product of the error and such an interpolant is zero since the $\pi_h e \in V_{h,0}$, as the approach requires. Thus we can just add it to the RHS, enabling us to carry on with the proof

$$\|e\|_E^2 = \int_{-1}^{1} e'e'dx - \int_{-1}^{1} e'(\pi_h e)'dx \tag{3}$$

Move terms under the same integration and introduce the notion of the linear behavior of the interpolant, meaning we sum all integration on the sub intervals x_{i-1} to x_i

$$\int_{-1}^{1} e'(e - \pi_h e)'dx = \sum_{i=1}^{n} \int_{x_{i-1}}^{x_i} e'(e - \pi_h e)'dx \tag{4}$$

integrate by parts

$$= \sum_{i=1}^{n} \int_{x_{i-1}}^{x_i} -e''(e - \pi_h e)dx + e'(x_i)\left[e(x_i) - \pi_h e(x_i)\right] - e'(x_{i-1})\left[e(x_{i-1}) - \pi_h e(x_{i-1})\right] \tag{5}$$

The second and third term are equal to zero since the interpolant $\pi_h e(x_i)$ is equal to e at these discrete set of points $x = x_i$, $i = [1, n]$. Now we can write the expression as

$$\sum_{i=1}^{n} \int_{x_{i-1}}^{x_i} -(u'' - u_h'')(e - \pi_h e)dx \tag{6}$$

here we identify that $u'' = -\delta^{-1} f(x)$ from the given differential equation 47. Further we acknowledge the residual as $R(u_h) = u'' - u_h'' = -\delta^{-}1 f(x) - u_h''$

$$\sum_{i=1}^{n} \int_{x_{i-1}}^{x_i} R(u_h)(e - \pi_h e)dx \tag{7}$$

By applying the Cauchy-Bunyakovsky–Schwartz inequality, we start to form something that looks like an error estimate

$$\sum_{i=1}^{n} \int_{x_{i-1}}^{x_i} R(u_h)(e - \pi_h e)dx \leq \sum_{i=1}^{n} \left\| R(u_h) \right\|_{L^2(I_i)} \left\| (e - \pi_h e) \right\|_{L^2(I_i)} \tag{8}$$

$$\leq \sum_{i=1}^{n} \left\| R(u_h) \right\|_{L^2(I_i)} ch_i \left\| e' \right\|_{L^2(I_i)} \tag{9}$$

Use Cauchy Schwartz again

$$\leq \sqrt{\sum_{i=1}^{n} \left(ch_i \left\| R(u_h) \right\|_{L^2(I_i)} \right)^2} \sqrt{\sum_{i=1}^{n} \|e'\|_{L^2(I_i)}^2} \tag{10}$$

Finally this leads to

$$\|e\|_E^2 = \left\| e' \right\|^2 \leq \sqrt{\sum_{i=1}^{n} \eta_i^2} \left\| e' \right\| \tag{11}$$

where $\eta^2 = (ch_i \left\| R(u_h) \right\|_{L^2(I_i)})^2$ Is a error estimate. We can cancel out the energy norm on both sides leaving us with the final form of the a posteriori error estimate

$$\|u - u_h\| \leq \sqrt{\sum_{i=0}^{n} \eta_i^2} \tag{12}$$

Plots

See appendix for code. The results can be seen in figure 1

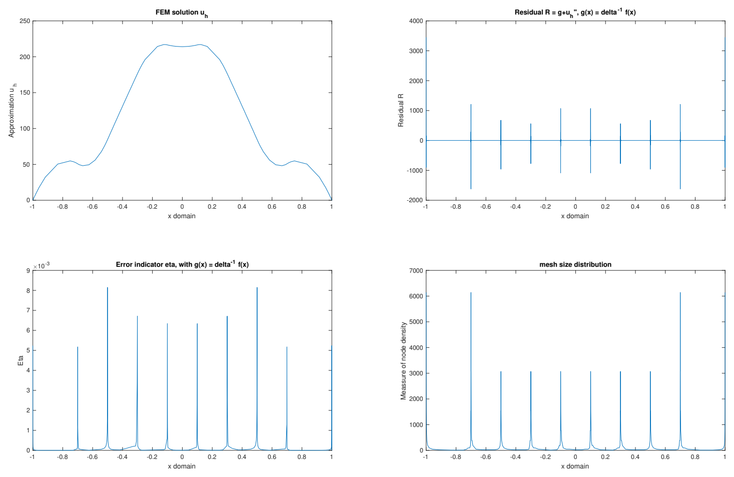

Figure 1: Note that the function used in the computation of u_h, u_h'' and η is not the original function $f(x)$, but $g(x) = \delta^{-1} f(x)$, thereby making the LHS of the DE 47 to be $u''(x)$

TWO DIMENSIONAL CONVERGENCE ANALYSIS

In this part we consider a simplified, scalar version of the predator-pray model in two dimensions. Piecewise linear basis function will construct the solution, which will later be implemented it in Matlab. Additionally the convergence analysis of the finite element method will be studied here.

Problem B1: 2D Garlekin finite elementmethods

$$\begin{cases} -\Delta u = f & \text{in} \quad \Omega \\ u = g & \text{on} \quad \partial\Omega \end{cases} \tag{13}$$

Standard Procedure to obtain the Garlekin Finite Elements of this DE is to first derive the weak form. By integration over our domain Ω and multiplying by a test function v, we can with Greens formula rewrite the DE as follows

$$\int_{\Omega} -(-\nabla u \cdot \nabla v) dx + \int_{\partial\Omega} \partial_n u v = \int_{\Omega} f v dx \tag{14}$$

where $\partial_n = \nabla \cdot \hat{n}$. The weak formulation states that we need to find:

$$u \in V_g := \{u : \|\nabla u\| < \infty, \ \|u\| < \infty, \ u = g \text{ on } \partial\Omega\} \tag{15}$$

such that

$$\int_{\Omega} -(-\nabla u \cdot \nabla v) dx = \int_{\Omega} f v dx \tag{16}$$

$$\forall v \in V_0 := \{v : \|\nabla v\| < \infty, \|v\| < \infty, v = 0 \ on \ \partial\Omega\} \tag{17}$$

Now a triangulated mesh is created

$$\tau_h = \{K\} \text{ , K elements, } \ h_k = diam(K) \text{ - meshsize} \tag{18}$$

Where the total set of all nodes $\{N\}$ on τ_h is the sum of all the internal nodes $\{N_h\}$ and its boundary nodes $\{N_b\}$

$$\{N\} = \{N_b\} + \{N_h\} \text{ , - all nodes on } \tau_h \tag{19}$$

Now we need to define the discrete trial -and testspaces:

Discrete trialspace

$$V_{g,h} \subset V_g : \ V_{g,h} = \{v : v \in C^0(\Omega), v|_k P_1(k), \ \forall k \in \tau_h, v = g \text{ on } \partial\Omega\} \tag{20}$$

Discrete testspace

$$V_{0,h} \subset V_0 : \ V_{0,h} = \{v : v \in C^0(\Omega), v|_k P_1(k), \ \forall k \in \tau_h, v = 0 \text{ on } \partial\Omega\} \tag{21}$$

Finally, we shall formulate the *Garlekin Finite Element Method*. Find $u_h \in V_{g,h}$ such that

$$\int_{\Omega} \nabla u \cdot \nabla v dx = \int_{\Omega} f v dx \quad \in V_{0,h} \tag{22}$$

The discrete solution u_h can be written as

$$u_h(x) = \sum_{N_j \in N_h} \xi_j \phi_j(x) + \sum_{N_j \in N_b} g(N_j) \phi_j(x) \tag{23}$$

Now insert this sum into the GFEM and get:
Find $\{\xi\}$, $\{N_j \in N_h\}$, such that

$$\sum_{N_j \in N_h} \xi_j \int_\Omega \nabla u \cdot \nabla v dx = \int_\Omega f \phi_i dx - \sum_{N_j \in N_h} g(N_b) \int_\Omega \nabla u \cdot \nabla v dx \,, \quad \forall N_j \in N_h \tag{24}$$

This can be rewritten as a matrix equation

$$A\vec{\xi} = \vec{b} \tag{25}$$

where the boundary conditions $g(N_b)$ are strongly imposed after calculating the stiffness matrix and the load vector, see code below.

Plots

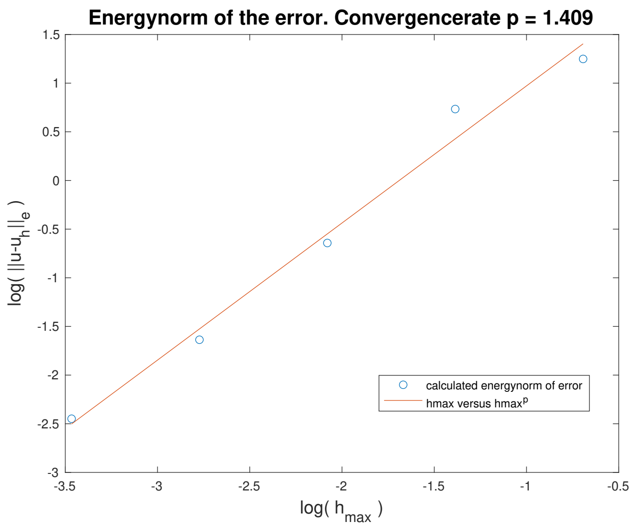

Figure 2: *Energynorm increases with the meshsize. The convergence rate tells us about when the error is unaffected by the size of the mesh. This is shown when the curve flattens. For these particular meshsizes, the curve does no flatten. So a finer mesh size can be used to achieve a better approximation of the solution.*

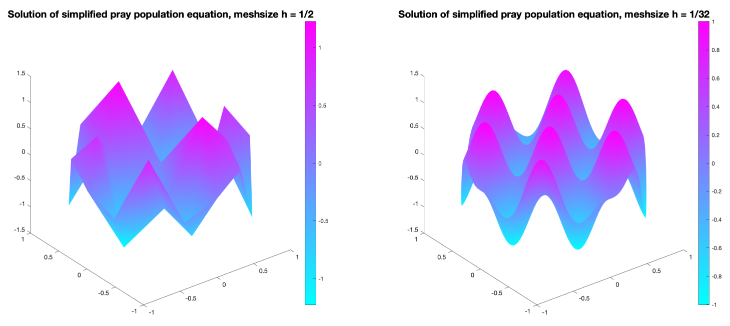

Figure 3: Plot of the solution u_h with a coarse mesh and a fine mesh

Problem B2: Constant predator population density

Task 1: Analytical derivations for GFEM and time discretization

Formulate a Galerkin finite element method using continuous piecewise linear approximation of 26. Discretize the time derivative using the Crank-Nicholson method and write down the corresponding linear algebra matrices and vectors.

$$\begin{cases} \partial_t u - u(1-u) + \frac{u}{u+\alpha} - \delta_1 \Delta u = 0 \quad (x,t) \quad \text{in} \quad \Omega \times (0,T] \\ \partial_t u = 0 \quad (x,t) \quad \text{on} \quad \partial\Omega \times (0,T] \\ u(x,0) = 1 + 20 \cdot \omega(x) \end{cases} \tag{26}$$

$\omega(x)$ it a uniformly randomized number between zero and one. The nonlinearity of this system has its complications. Fortunately it can be solved by a substitution and handled explicitly in the upcoming computational implementation. Substitute the nonlinear terms for S

$$S(u) = u_h(1-u_h) - \frac{u_h}{u_h + \alpha} \tag{27}$$

To obtain the Garlekin Finite Elements approximation of this DE, the weak form must be derived.

$$\int_\Omega \partial_t uvdx - \int_\Omega S(u)vdx - \int_\Omega \delta_1 \Delta uvdx = 0 \tag{28}$$

Weak formulation states that we need to find:

$$u \in V_0 = \{v(x,t) : \|v(\cdot,t)\| + \|\nabla v(\cdot,t)\| < \infty\} \tag{29}$$

such that

$$\int_\Omega \partial_t uvdx - \int_\Omega S(u)vdx + \int_\Omega \delta_1 \nabla u \cdot \nabla vdx = 0$$

$\forall v \in V_0$

(30)

which is the condition we need for finding u as the weak formulation states above, Because of the lower order of differentiation, from Δu to $\nabla u \cdot \nabla v$. Next up is to define the discrete version of the testspace V_0, as $V_{h,0} \subset V_0$, $dim(v_{h,0} < \infty)$. The Garlekin finite element formulation states that we need to find $u_h \in V_{h,0}$

$$V_{h,0} = \{v(x,t) : v(x,t) \in C^0(\tau_h), \forall t \in (0,T], v|_k \in P_1(k), \; \forall k \in \tau_h\} \tag{31}$$

k refers to the triangle elements in the mesh defined as

$$\tau_h = \{K\} \text{ , K elements, } \; h_k = diam(K) \text{ - meshsize} \tag{32}$$

Where the total set of all nodes $\{N\}$ on τ_h is the sum of all the internal nodes $\{N_h\}$ and its boundary nodes $\{N_b\}$

$$\{N\} = \{N_b\} + \{N_h\} : \text{all nodes on } \tau_h \tag{33}$$

As the finite test space and the mesh geometry is defined, we can start to construct the Garlekin Finite Element form ,which states that $u_h \in V_{h,0}$ is to be found for

$$\int_\Omega \partial_t u_h v dx - \int_\Omega S(u) v dx + \int_\Omega \delta_1 \nabla u_h \cdot \nabla v dx = 0, \quad \forall v \in V_{h,0} \tag{34}$$

Since $u_h \in V_{h,0}$, $\exists \{\xi\}_{N_j \in N_h}$ such that

$$u_h(x,t) = \sum_{N_j \in N_h} \xi_j(t)\phi_j(x) \tag{35}$$

Concerning the nonlinear term, we now know that $(u_h)_j = (\xi_j(t))$, meaning we can rewrite S_j as

$$S_j = \xi_j(1-\xi_j) - \frac{\xi_j}{\xi_j + \alpha} \tag{36}$$

Thus its linear interpolant $\Pi_h S(x,t)$, is

$$\Pi_h S(x,t) = \sum_{N_j \in N_h} \left[\xi_j(t)(1-\xi_j(t)) - \frac{\xi_j(t)}{\xi_j(t)+\alpha} \right] \phi_j(x) \tag{37}$$

bring in ϕ_j into the nonlinear terms such that all terms within the brackets will become $\xi_j(t)\phi_j(x) = u_{h,j}(x,t)$.

With both u_h and $S(x,t)$ solved for by FEM and linear interpolation, we can substitute back the sums into the GFEM formulation to obtain our stiffness and mass matrices.

$$\begin{aligned} \sum_{N_j \in N_h} \partial_t \xi_j(t) \int_\Omega \phi_j \phi_j dx + \sum_{N_j \in N_h} \xi_j(t) \int_\Omega \delta_1 \nabla \phi_j \nabla \phi_i dx = \\ \sum_{N_j \in N_h} \left[\xi_j(t)(1-\xi_j(t)) - \frac{\xi_j(t)}{\xi_j(t)+\alpha} \right] \int_\Omega \phi_j \phi_i dx \end{aligned} \tag{38}$$

The above equation can be recognized to contain the stiffness matrix, mass matrix and our solution $\vec{\xi}$. Note that the nonlinear terms can be considered as the load vector since they will be known for the current timestep, starting with the initial condition. With these matrices and vectors, the GFEM equation can be reformulated as a matrix equation

$$M\partial_t \vec{\xi}(t) + \delta_1 A \vec{\xi}(t) = \vec{b} \tag{39}$$

Now all that is left is to discretize the time derivative, which is done by the *Crank-Nicholson Method*, which states that nonlinear terms can be linearized at time n+1 by using the solution for the previous time n.

$$M\frac{\vec{\xi^{n+1}} + \vec{\xi^n}}{k_n} + \delta_1 A \frac{\vec{\xi}^{n+1} + \vec{\xi}^n}{2} = M\left(\vec{\xi}^n(1-\vec{\xi}^n) - \frac{\vec{\xi}^n}{\vec{\xi}^n + \alpha} \right) \tag{40}$$

Bring all $\vec{\xi}^{n+1}$ to LHS and $\vec{\xi}^n$ to the RHS

$$\left(M + \frac{1}{2}k_n\delta_1 A\right)\vec{\xi}^{n+1} = \left(M - \frac{1}{2}k_n\delta_1 A\right)\vec{\xi}^n - M\left(\vec{\xi}^n(1-\vec{\xi}^n) - \frac{\vec{\xi}^n}{\vec{\xi}^n+\alpha}\right) \tag{41}$$

Task 2: Plots & Matlab code for problem B2

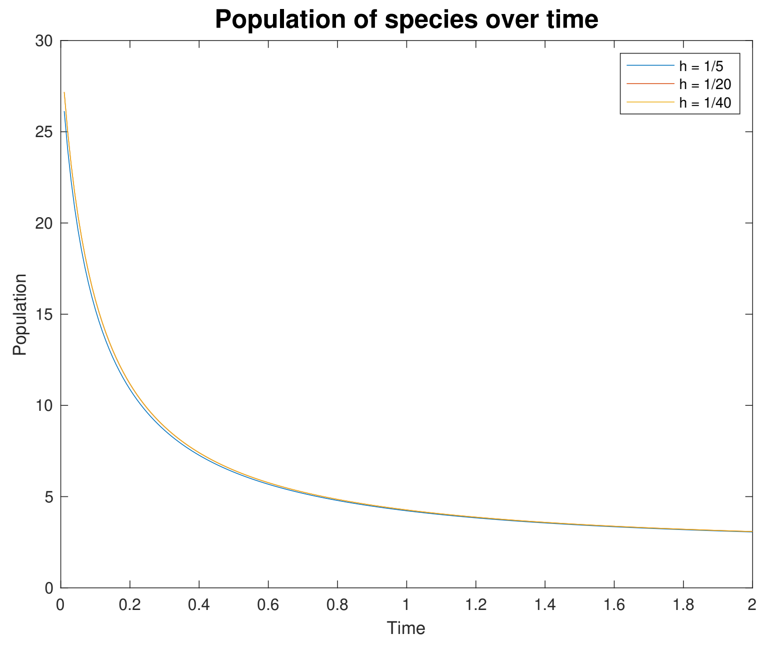

Figure 4: the decrease of the population is very similar for each mesh size

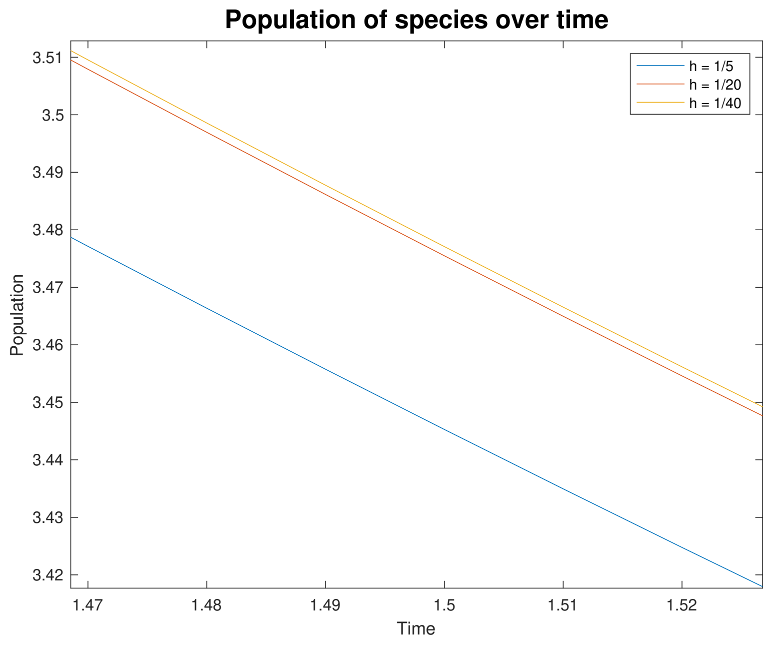

Figure 5: Here is a zoomed image showing the small impact the choice of mesh size has on the population over time. However in the beginning of the simulation, since the slope is steeper, a greater difference in population can be seen

Discretized solutions of the PDE

Note that since the initial conditions are randomly generated by Matlabs function *rand()*, the solutions for each mesh size is different. Furthermore it might be prefferable to see how the solution evolves over time, so i included plots for three time instances: at $t = 0$, where only the random initial condition is seen, and then for$t = 0.5$ and $t = 2$.

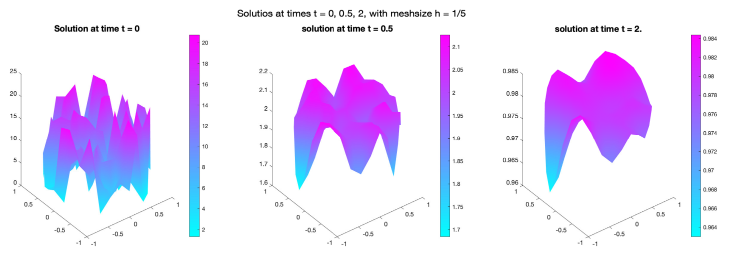

Figure 6: How the solution evolves over time for three time instances, for the mesh size h = 1/15

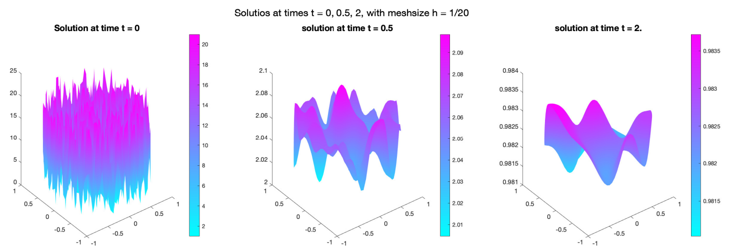

Figure 7: How the solution evolves over time for three time instances, for the mesh size h = 1/20

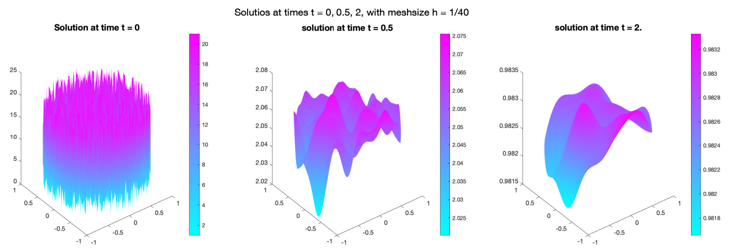

Figure 8: How the solution evolves over time for three time instances, for the mesh size h = 1/40

PART C - SYSTEM OF PDES

The final part of the project presents a system of PDE's, where the solution $\vec{u}$ lives in a vector function space. The system describes the interplay between two species, one which hunts and eats the other. It is referred to as the Predator-Prey model. The governing system of partial differential equations is

$$\partial_t u_1 - u_1(1-u_1) + \frac{u_1 u_2}{u_1+\alpha} - \delta_1 \Delta u_1 = 0 \tag{42}$$

$$\partial_t u_2 + \gamma u_2 - \beta \frac{u_1 u_2}{u_1+\alpha} - \delta_2 \Delta u_2 = 0 \tag{43}$$

$$\hat{n} \cdot \nabla u_1 = 0 \quad \text{on } \partial\Omega \times (0,T] \tag{44}$$

$$\hat{n} \cdot \nabla u_2 = 0 \quad \text{on } \partial\Omega \times (0,T] \tag{45}$$

$$u_1(\vec{x},0) = u_{1,0}(\vec{x}) \tag{46}$$

$$u_2(\vec{x},0) = u_{2,0}(\vec{x}) \tag{47}$$

where $u_{1,0}(\vec{x})$ and $u_{1,0}(\vec{x})$ are given functions. This is called the *strong form*, The PDE's contain a laplacian, and we need to bring that second order differentiation down to the first order, which will result in an expression corresponding to the *weak form*. Thus we need to find the solution $\vec{u}$ that lives in the vector function space W

$$\vec{u} = (u_1, u_2) \in W = \{\vec{v}(\vec{x},t) : \vec{v} \in H^1(\Omega) \times H^1(\Omega) \forall t \in (0,T]\} \tag{48}$$

such that

$$\begin{cases} \int_\Omega \partial_t u_1 v_1 d\vec{x} - \int_\Omega \left(u_1(1-u_1) + \frac{u_1u_2}{u_1+\alpha}\right) v_1 d\vec{x} + \int_\Omega \delta_1 \nabla u_1 \cdot \nabla v_1 d\vec{x} = 0 \\ \int_\Omega \partial_t u_2 v_2 d\vec{x} - \int_\Omega \beta \frac{u_1u_2}{u_1+\alpha} v_2 d\vec{x} - \int_\Omega \gamma u_2 v_2 d\vec{x} + \int_\Omega \delta_2 \nabla u_2 \cdot \nabla v_2 d\vec{x} = 0 \forall \vec{v} = (v_1, v_2) \in W \end{cases} \tag{49}$$

Now when the weak form is derived, we can formulate the Garlekin equation, which says we need to find the discretized solution $\vec{u}_h \in W_{h,0}$, $W_{h,0} \subset W$

$$W_{h,0} = \{\vec{v}(\vec{x},t) : \vec{v} \in C^0(\Omega) \forall t \in (0,T], \vec{v}|_k \in P_1(k), \forall k \in \tau_h\} \tag{50}$$

such that

$$\begin{cases} \int_\Omega \partial_t u_{1,h} v_1 d\vec{x} - \int_\Omega \left(u_{1,h}(1-u_{1,h}) + \frac{u_{1,h}u_{2,h}}{u_{1,h}+\alpha}\right) v_1 d\vec{x} + \int_\Omega \delta_1 \nabla u_{1,h} \cdot \nabla v_1 d\vec{x} = 0 \\ \int_\Omega \partial_t u_{2,h} v_2 d\vec{x} - \int_\Omega \beta \frac{u_{1,h}u_{2,h}}{u_{1,h}+\alpha} v_2 d\vec{x} - \int_\Omega \gamma u_{2,h} v_2 d\vec{x} + \int_\Omega \delta_2 \nabla u_{2,h} \cdot \nabla v_2 d\vec{x} = 0 \\ \forall \vec{v} = (v_1, v_2) \in W_h \end{cases} \tag{51}$$

since $\vec{u}_h \in W_{h,0}$ there exists a vector $\vec{\xi}$ such that $\vec{u}_h$ can be written as a linear combination of the bases of the vectorspace $W_{h,0}$. More formally this statement is written as

$$(u_1, u_2)^T = \vec{u}_h \in W_h \rightarrow \exists \{\vec{\xi}_j\}_{N_j \in N_h} \tag{52}$$

such that

$$(u_1, u_2)^T = \vec{u}_h = \sum_{N_j \in N_h} \vec{\xi}_j \phi_j \tag{53}$$

Now we have way to express the solution u_h in terms of a linear combination of the basis functions of the vector function space W_h. However this cannot be applied to the nonlinear terms. So a a different approach is needed. More precisely, we need to substitute the nonlinear terms.

$$S = u_1(1 - u_1) + \frac{u_1 u_2}{u_1 + \alpha} \tag{54}$$

$$T = \beta \frac{u_1 u_2}{u_1 + \alpha} \tag{55}$$

we can interpolate these terms in the vector function space $W_{h,0}$

$$\Pi_h S \in W_{h,0} \rightarrow \Pi_h S(\vec{x}, t) = \sum_{N_j \in N_h} S_j(t) \phi_j(\vec{x}) \tag{56}$$

$$\Pi_h T \in W_{h,0} \rightarrow \Pi_h T(\vec{x}, t) = \sum_{N_j \in N_h} T_j(t) \phi_j(\vec{x}) \tag{57}$$

and can therefore the substitution terms be expressed as follows

$$S_j(t) = \xi_{1,j}(1 - \xi_{1,j}) + \frac{\xi_{1,j}\xi_{2,j}}{\xi_{1,j} + \alpha} \rightarrow \Pi_h S = \sum_{N_j \in N_h} \left(\xi_{1,j}(1 - \xi_{1,j}) + \frac{\xi_{1,j}\xi_{2,j}}{\xi_{1,j} + \alpha} \right) \tag{58}$$

$$T_j(t) = \beta \frac{\xi_{2,j}\xi_{1,j}}{\xi_{2,j} + \alpha} \rightarrow \Pi_h T = \sum_{N_j \in N_h} \left(\beta \frac{\xi_{2,j}\xi_{2,j}}{\xi_{2,j} + \alpha} \right) \tag{59}$$

Now we can substitute the linear and nonlinear terms into equation 51

$$\sum_{N_j \in N_h} \partial_t \xi_{1,j} \int_\Omega \phi_i \phi_i d\vec{x} - \sum_{N_j \in N_h} \left(\xi_{1,j}(1 - \xi_{1,j}) + \frac{\xi_{1,j}\xi_{2,j}}{\xi_{1,j} + \alpha} \right) \int_\Omega \phi_i \phi_j d\vec{x} + \sum_{N_j \in N_h} \xi_{i,j} \int_\Omega \delta_1 \nabla \phi_i \cdot \nabla \phi_j d\vec{x} = 0 \tag{60}$$

$$\sum_{N_j \in N_h} \partial_t \xi_{2,j} \int_\Omega \phi_i \phi_i d\vec{x} - \sum_{N_j \in N_h} \beta \frac{\xi_{1,j}\xi_{2,j}}{\xi_{1,j} + \alpha} \int_\Omega \phi_i \phi_j d\vec{x} - \sum_{N_j \in N_h} \gamma \xi_{2,j} \int_\Omega \phi_i \phi_j d\vec{x} + \sum_{N_j \in N_h} \xi_{2,j} \int_\Omega \delta_2 \nabla \phi_i \cdot \nabla \phi_j d\vec{x} = 0 \tag{61}$$

We recognize the stiffness matrices A and the mass matrices M

$$\sum_{N_j \in N_h} \partial_t \vec{\xi}_1 M - \sum_{N_j \in N_h} \left(\vec{\xi}_1(1 - \vec{\xi}_1) + \frac{\vec{\xi}_1 \vec{\xi}_2}{\vec{\xi}_1 + \alpha} \right) M + \sum_{N_j \in N_h} \xi_{i,j} \delta_1 A = 0 \tag{62}$$

$$\sum_{N_j \in N_h} \partial_t \vec{\xi}_2 M - \sum_{N_j \in N_h} \beta \frac{\vec{\xi}_1 \vec{\xi}_2}{\vec{\xi}_1 + \alpha} M - \sum_{N_j \in N_h} \gamma \vec{\xi}_2 M + \sum_{N_j \in N_h} \vec{\xi}_2 \delta_2 A = 0 \tag{63}$$

Now we have expressed the problem in such a way that we can assemble the matrices needed and solve for $\vec{\xi}_1$ and $\vec{\xi}_2$. This problem however, is a time dependent one, therefore we need a way to predict the next solution depending on the current one. There are several ways to discretize these time derivatives, some stable, but only for select conditions, while others are unconditionally stable. The *Crank Nicholson Scheme* is such a a method. The scheme approximates the solution as the mean between two time steps n and $n+1$: $\vec{\xi} = \frac{\vec{\xi}^n + \vec{\xi}^{n+1}}{2}$, and the time derivative as the linear slope between the solution at two time steps $\partial\vec{\xi} = \frac{\vec{\xi}^{n+1} - \vec{\xi}^n}{\Delta t}$, where Δt is the length of the time step. Thereafter one brings all terms corresponding to the time $n+1$ to the LHS and terms of time n to the RHS.

$$\left(M + \frac{1}{2}\delta_1 k_n A\right)\vec{\xi}_1^{n+1} = \left(M - \frac{1}{2}\delta_1 k_n A\right)\vec{\xi}_1^n + M\left(\vec{\xi}_1^n(1 - \vec{\xi}_1^n) + \frac{\vec{\xi}_1^n \vec{\xi}_2^n}{\vec{\xi}_1^n + \alpha}\right)$$
$$\vec{\xi}_1(0) = \vec{u}_{1,0} \quad (64)$$

$$\left(M + \frac{1}{2}\delta_2 k_n A - \frac{1}{2}\gamma k_n M\right)\vec{\xi}_2^{n+1} = \left(M - \frac{1}{2}\delta_2 k_n A + \frac{1}{2}\gamma k_n M\right)\vec{\xi}_2^n + M\beta\frac{\vec{\xi}_1^n \vec{\xi}_2^n}{\vec{\xi}_1^n + \alpha}$$
$$\vec{\xi}_2(0) = \vec{u}_{2,0} \quad (65)$$

Now we can begin implementing the theory in Fenics. For code, see appendix.

FENICS IMPLEMENTATION OF THE DIFFUSIVE LOTKA-VOLTERRA EQUATIONS

Given Initial Condition

The Fenics program plots the solution for u and v (prey and predator), given some initial conditions, which can for example be set as follows:

$$\begin{cases} u_0 = \frac{4}{15} - 2 \cdot 10 - 7(x_1 - 0.1x_2 - 255)(x_1 - 0.1x_2 - 675) \\ v_0 = \frac{22}{45} - 3 \cdot 10^{-5}(x_1 - 450) - 1.2 \cdot 10^{-4}(x_2 - 150) \end{cases} \tag{66}$$

Due to the initial conditions which seems to separate the prey and predators by what can be interpreted by 90 degrees (see figure 9), a spiraling motion grows into existence, where the predators are consuming the prey and thus reproducing in those areas (reaction terms). The prey, in absence of any predators will have a net positive, and thus grow in the direction away from the predators. Both populations are also spreading out into the space containing fewer of their own species (diffusion).

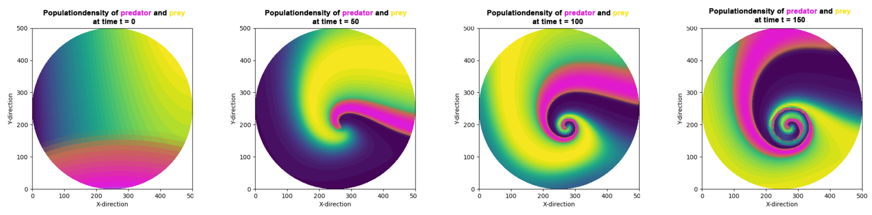

Figure 9: Yellow/green denotes prey and pink/orange denoted the predators. The initial condition is set up in such a way that a spiraling shape appears

After a 1000 time steps, the spiral pattern has deteriorated(see figure 10) due to the decreasing distance between the spiral arms, and the eventual interaction between the "arms". It has now evolved into a finer and more uniform pattern, with some now smaller spiraling patterns. The large populations has split up into smaller families (neglecting territorial behavior, if one now would consider foxes and rabbits). The populations seem to stabilize, meaning one does not outgrow the other.

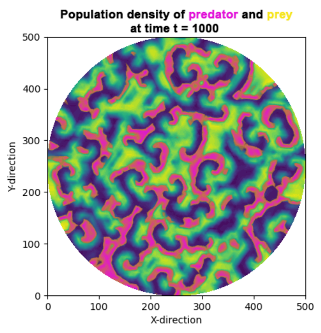

Figure 10: *Yellow/green denotes prey and pink/orange denoted the predators. After a certain time the spirals react with each other and a finer pattern emerges. See how the pink predators are on the edges surrounding the prey, moving towards them.*

Random Initial Conditions

If one imposes a random initial condition instead of the former, more defined initial condition, the solution converges faster to the same pattern that can bee seen with the previous IC. The explanation lies in how much the populations are separated initially. They are separated by quite a lot for the first IC's and not as much by the second randomized IC's. See figure 9, 11 and 12 for comparison. Another noteworthy properly of the simulation is the size of the pattern that emerges after a longer period of time, see for time $t = 1000$ of figure 11. The size seems to depend on δ_1 and δ_2, which determines how much the population will diffuse.

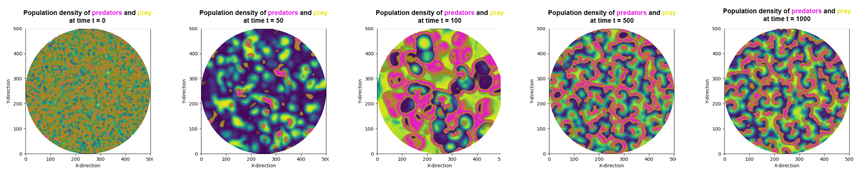

Figure 11: *Yellow/green denotes prey and pink/orange denoted the predators. Due to the uniformly random initial conditions, the evolution of the system converges faster toward the fine pattern that can be seen on the last frame.*

As for the size of each population, It oscillates heavily for the first initial condition, by the fact that they are separated spatially and the initial size of the population differs significantly (94K of the predators and 51K preys). See figure 12. One can also reflect upon the phase shift of the high frequency oscillations. Studying the data table, one can conclude that the phase is approximately $\tau = 4.5$ time units. Finer precision can be achieved, if the temporal resolution of $\Delta t = 0.5$ is adjusted.

APPENDIX

Part A: 1D FEM solver in Matlab

Main script

```
clc;
```

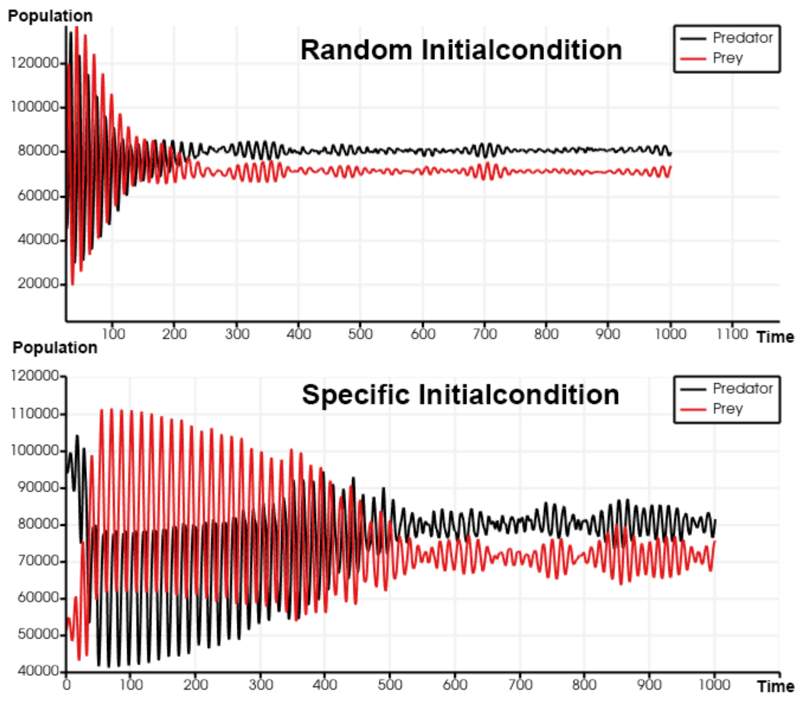

Figure 12: *When initially the size of the populations are similar, they converge faster to a stable state. In contrast, the populations*

```
clear all;
close all;

global delta;
delta = 0.01;

a = -1; % left end point of interval
b = 1; % right
N = 12; %...so wachayall want?
h = 1/N;
x = a:h:b; % node coords

eta = ones(N,1);      % allocate element residuals
TOL = 1e-3;
MAX = 1e4;
alpha = 0.9;

% Will never reach 10000 nodes before reaching tolerance requirement of
% 1e-3



while N<MAX && sum(eta.^2)>TOL

% solution, assemble!
% stiffness matrix assembly
A = my_stiffness_matrix_assembler(x);

% load vector assembly
B = my_load_vector_assembler(x);

% mass matrix assembly
M = my_mass_matrix_assembler(x);

% solve system of equations
```

```
xi = A\B;

% discrete laplace(u_h) approximation
Lxi = -M\A*xi;

%%%%%%%%%%%%%%%%%%%%
% compute residual %
%%%%%%%%%%%%%%%%%%%%
eta = zeros(N,1);     % allocate element residuals
for i = 1:length(x)-1 % loop over elements
    h = x(i+1) - x(i); % element length
    a1 = f(x(i))+Lxi(i); % temporary variables
    b1 = f(x(i+1))+Lxi(i+1);
    t = (a1^2+b1^2)*h/2; % integrate f 2. Trapezoidal rule
    eta(i) = sqrt(h*h*t); % element residual
end


%%%%%%%%%%%%%%%%%%%%%%%%%%
% refine select elements %
%%%%%%%%%%%%%%%%%%%%%%%%%%
for i = 1:N
    if(eta(i)^2>alpha*max(eta.^2))          % if residual is large
        x = [x (x(i+1)+x(i))/2];    % insert another node
    end
end
N = length(x);       % the size of the vector N increases
x = sort(x);         % however its inserted at the end, so we need to sort the vector

end


% solution, assemble!one last time!

% stiffness matrix assembly
A = my_stiffness_matrix_assembler(x);

% load vector assembly
B = my_load_vector_assembler(x);

% mass matrix assembly
M = my_mass_matrix_assembler(x);

% solve system of equations
xi = A\B;

% discrete laplace(u_h) approximation
Lxi = -M\A*xi;

%%%%%%%%%%%%%%%%%%%%
% compute residual %
%%%%%%%%%%%%%%%%%%%%
eta = zeros(N,1);     % allocate element residuals
for i = 1:length(x)-1 % loop over elements
    h = x(i+1) - x(i); % element length
    a1 = f(x(i))+Lxi(i); % temporary variables
    b1 = f(x(i+1))+Lxi(i+1);
    t = (a1^2+b1^2)*h/2; % integrate f 2. Trapezoidal rule
    eta(i) = sqrt(h*h*t); % element residual
end

% evaluate the given function
for i=1:length(x)
    func(i,1) = f(x(i));
end

% residual
```

```
R = func+Lxi;

%plot everything
figure;
subplot(2,2,1)
plot(x,xi)
xlabel('x domain')
ylabel('Approximation u_h')
title('FEM solution u_h');

subplot(2,2,2)
plot(x,R);
xlabel('x domain')
ylabel('Residual R')
title('Residual R = g+u_h'''', g(x) = delta^{-1} f(x)')

subplot(2,2,3)
plot(x,eta);
xlabel('x domain')
ylabel('Eta')
title('Error indicator eta, with g(x) = delta^{-1} f(x)')

subplot(2,2,4)
plot(x(2:end),[1./diff(x)])
xlabel('x domain')
ylabel('Meassure of node density')
title('mesh size distribution')
```

Function definition

```
function func = f(x)

global delta;
% discrete f(x), LHS of DE
funcdef = @(x) [(abs(x)<=0.1) (abs(0.2-abs(x))<=0.1) (abs(0.4-abs(x))<=0.1) (abs(0.6-abs(x))<=0.1

func = funcdef(x);
end
```

1D Stiffness matrix assembler

```
function A=my_stiffness_matrix_assembler(x)

    N = length(x)-1;
    A = zeros(N+1, N+1);     % allocate stiffnes matrix
    for i = 1:N % loop over elements
        h = x(i+1) - x(i); % element length
        n = [i i+1]; % nodes
        A(n,n) = A(n,n) + [1 -1; -1 1]/h; % assemble element stiffness
    end

    A(1,1) = 1;
    A(1,2) = 0;
    A(N+1,N+1)= 1;
    A(N+1,N)=0;

end
```

1D Load vector assembler

```
function B=my_load_vector_assembler(x)
```

```
    N = length(x)-1;
    B = zeros(N+1, 1);      % allocate load vector
    for i = 1:N
            h = x(i+1) - x(i);
            n = [i i+1];
            B(n) = B(n) + [f(x(i)); f(x(i+1))]*h/2;
    end
    B(1) = 0;   % Apply BC strongly
    B(end) = 0; % Apply BC strongly
end
```

1D Mass matrix assembler

```
function M=my_mass_matrix_assembler(x)
    N = length(x)-1;
    M = zeros(N+1,N+1);     % allocate mass matrix
    for i = 1:N % loop over subintervals
        h = x(i+1) - x(i); % interval length

        M(i,i) = M(i,i) + h/3; % add h/3 to M(i,i)
        M(i,i+1) = M(i,i+1) + h/6;
        M(i+1,i) = M(i+1,i) + h/6;
        M(i+1,i+1) = M(i+1,i+1) + h/3;
    end

end
```

Part B: 2D PDE FEM Solver

Main script

```
close all; clear all; clc;
% define domain
geometry = @circleg ;
hmax = [5 20 40].^(-1); % REMOVE FIRST ELEMENT IN FINAL

% define constants
delta = 0.01;       % physical meaning?
alpha = 4;          % physical meaning?
k = 0.01;           % temporal resolution
T = 2;              % timelimit

% define vectors
populations=zeros(length(hmax),T/k+1); %store population rate for each h
a=1;    %indexing for populationrates, increases by one for each h
time = [0:k:T];     % time vector



% spin it!
for h=hmax

    % Initialize mesh
    [p,e,t]  = initmesh (geometry , 'hmax' , h);

    % begin a figure for the solutions
    figure
    sgtitle(['Solutios at times t = 0, 0.5, 2, with meshsize h = 1/', num2str(h^(-1))],'fontsize'

    u_h = zeros(length(p),1); % create empty vector for all solutions

    % impose initial condition
    for i=1:length(p)
```

```
        u_h(i) = InitialCondition();
    end

    % Initial condition solution plot
    subplot(1,3,1)
    pdeplot(p,e,t, 'xydata', u_h, 'zdata', u_h)
    title('Solution at time t = 0','fontsize',14)

    % Matrices, Assemble!
    [A,M,b] = matrixAssembler(p,t,u_h); % u_h set from initial condition above

    % CRANK-NICHOLSON TIME DISCRETIZATION
    LHS = M/k+0.5*delta*A;     % matrix computation for the upcoming timestep u_{h}^{n+1}
    RHS = M/k-0.5*delta*A;     % matrix copmutation for the current timestep u_{h}^{n}

    population = [];% create empty vector for storage of population
    for i=time

        u_h = LHS\(RHS*u_h-M*S(u_h));  % solve for the upcoming u_{u,n+1},
                                % using the matrices and the previous solution u_{h,n}


        % population
        population = [population  Population(p,e,t,u_h)];
%         pdeplot(p,e,t, 'xydata', u_h, 'zdata', u_h)
%         pause(0.001);

        % solution plot halfway at t = 0.5
              if(i==0.5)
            subplot(1,3,2)
            pdeplot(p,e,t, 'xydata', u_h, 'zdata', u_h)
            title('solution at time t = 0.5','fontsize',14)
        end
    end

    %solution plot at ending, t=2
        subplot(1,3,3)
        pdeplot(p,e,t, 'xydata', u_h, 'zdata', u_h)
        title('solution at time t = 2.','fontsize',14)
        pause;

     populations(a,:)=population;    % store all populationvalues
                                        % for all meshsizes
     a = a+1;    % increace indexing
end

% differentiate the population vector to obtain the rate
% populationrates = zeros(length(hmax),length(populations));
% for j=2:length(population)
%     populationrates(:,j) = (populations(:,j)-populations(:,j-1))/k;
% end

% Task is to only plot the population for the two first meshes
figure;
plot(time(2:end), populations(1,2:end),time(2:end), populations(2,2:end),time(2:end), populations
legend({'h = 1/5','h = 1/20', 'h = 1/40'});
title('Population of species over time','fontsize',16)
xlabel('Time')
ylabel('Population')

% sgtitle('Population of species over time','fontsize',16)
% subplot(3,1,1)
% plot(time(2:end), populations(1,2:end))
% title(['Meshsize h = 1/',num2str(hmax(1)^(-1))],'fontsize',14)
% xlabel('time in seconds','fontsize',14)
% ylabel('Population','fontsize',14)
% subplot(3,1,2)
% plot(time(2:end), populations(2,2:end))
% title(['Meshsize h = 1/', num2str(hmax(2)^(-1))],'fontsize',14)
% xlabel('time in seconds','fontsize',14)
% ylabel('Population','fontsize',14)
% subplot(3,1,3)
% plot(time(2:end), populations(3,2:end))
% title(['Meshsize h = 1/', num2str(hmax(3)^(-1))],'fontsize',14)
% xlabel('time in seconds','fontsize',14)
% ylabel('population,','fontsize',14)
```

Initial Conditions

```
function [u] = InitialCondition()
    u = 1+20*rand();
end
```

Matrix Assembler

```
function [A,M,b] = matrixAssembler(p,t,u)

    % call all assemblers
    A = stiffnessMatrixAssembler2D(p,t);
    M = massMatrixAssembler2D(p,t);
    b = loadVectorAssembler2D(p,t,u);
end
```

2D Stiffness Matrix Assembler

```
function A = stiffnessMatrixAssembler2D(p,t)
    np = size(p,2); % number of nodes
    nt = size(t,2); % number of elements

    A = zeros(np,np); % allocate stiffness matrix

    for i = 1:nt    % loop over elements

        nodes = t(1:3,i);   % local matrix to global matrix mapping

        x = p(1,nodes);     % node x-coordinates
        y = p(2,nodes);     % node y-coordinates

        [area,b,c] = HatGradients(x,y); % compute gradients in x&y direction
                                        % and the area of traingle K.

        localA = (b*b'+c*c').*area;     % local element stiffness matrix


        A(nodes,nodes) = A(nodes,nodes) + localA;   % add local matrix stiffnesses
                                                    % to global matrix A
    end
end
```

2D Mass Matrix Assembler

```
function M = massMatrixAssembler2D(p,t)
    np = size(p,2); % number of nodes
    nt = size(t,2); % number of elements

    M = zeros(np,np); % allocate mass matrix

    for i = 1:nt % loop over elements

        nodes = t(1:3,i);   % local matrix to global matrix mapping

        x = p(1,nodes);     % node x-coordinates
        y = p(2,nodes);     % node y-coordinates

        area = polyarea(x,y);   % triangle area with matlabs built-in fucntion


        localM =    [2 1 1;     % local element mass matrix
```

```
                      1 2 1;
                      1 1 2]/12*area;
        M(nodes,nodes) = M(nodes,nodes)+ localM;     % add local element mass matrices
                                                     % to the global matrix M
    end
end
```

2D Load Vector Assembler

```
function b = loadVectorAssembler2D(p,t,u)
    np = size(p,2); % number of nodes
    nt = size(t,2); % number of elements

    b = zeros(np,1);% allocate load vector

    for i = 1:nt    % loop over all elements
        nodes = t(1:3,i);   % local vector to global vector mapping

        x = p(1,nodes);     % node x-coordinates
        y = p(2,nodes);     % node y-coordinates

        area = polyarea(x,y);   % triangle area with matlabs built-in fucntion

        localb = S(u(nodes))/3*area;    % local element load vector

        b(nodes) = b(nodes) + localb;   % add local element load vector
                                        % to global load vector b
     end
end
```

nonlinear substitution S

```
function s = S(u)
    % substitution to handle the nonlinear terms
    alpha = 4;
    s = -u.*(1-u)+(u./(u+alpha));
end
```

Population integration function

```
function population = Population(p,e,t,u)
    population = 0;         % initialize population count
    for i=1:size(t,2)
        nodes = t(1:3,i);   % local to global mapping
        x = p(1,nodes);     % nodes x-coordinates
        y = p(2,nodes);     % nodes y-coordinates

        area = polyarea(x,y);   % triangle area with matlabs built-in function

        % increasing population by discrete integration ocer triangle by trapezodial rule
        population = population + area*(u(nodes(1)) + u(nodes(2)) + u(nodes(3)))/3;
    end
end
```

Part C: 2D system of PDE FEM solver in Python

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Dec 16 10:35:53 2020

@author: joelbjervig
"""

import matplotlib.pyplot as plt
from fenics import *
import random

# Create mesh and define function space
mesh = Mesh("circle.xml")
#x = SpatialCoordinate(mesh)

# Construct the finite element space polynomial of degree 1
V = VectorFunctionSpace(mesh, 'P', 1)

# Define parameters
T = 1000
dt = 0.5
alpha = 0.4
beta = 2
gamma = 0.8
delta1 = 1
delta2 = 1
theta = 0.5


# Class representing the intial conditions TASK 1
class InitialConditions1(UserExpression):
    def eval (self , values , x):
        values[0] = (4/15)-2*pow(10,-7)*(x[0]-0.1*x[1]-255)*(x[0]-0.1*x[1]-675)
        values[1] = (22/45)-3*pow(10,-5)*(x[0]-450)-1.2*pow(10,-4)*(x[1]-150)

    def value_shape(self):
        return (2 ,)

# Class representing the intial conditions TASK 2
class InitialConditions2(UserExpression):
    def eval (self , values , x):
        r = random.uniform(0,1)
        values[0] = 0.5*(1-r)
        values[1] = 0.25+0.5*r

    def value_shape(self):
        return (2 ,)

# test and trial functions
u, v = TrialFunction(V), TestFunction(V)

# Define initial condition chosse IC1 or IC2 depending on task
u_initial = Function(V)
u_initial = InitialConditions2(degree=2)

# interpolate initial condition
u0 = Function(V)
u0 = project(u_initial, V) # or project u initial onto V space as project(u_initial)


# Create bilinear and linear forms. first term from backward euler
a0 = inner(u[0],v[0])*dx - dt*theta*inner(u[0],v[0])*dx + dt*theta*delta1*inner(grad(u[0]),grad(v
a1 = inner(u[1],v[1])*dx + dt*theta*gamma*inner(u[1],v[1])*dx + dt*theta*delta2*inner(grad(u[1]),
# a0 = u[0]*v[0]*dx - dt*u[0]*v[0]*dx + theta*delta1*dt*inner(grad(u[0]),grad(v[0]))*dx
# a1 = u[1]*v[1]*dx - dt*gamma*u[1]*v[1]*dx + theta*delta2*dt*inner(grad(u[1]),grad(v[1]))*dx

L0 = inner(u0[0],v[0])*dx + dt*(1-theta)*inner(u0[0],v[0])*dx - dt*(1-theta)*delta1*inner(grad(u0
L1 = inner(u0[1],v[1])*dx - dt*(1-theta)*gamma*inner(u0[1],v[1])*dx - dt*(1-theta)*delta2*inner(g


# L0 = u0[0]*v[0]*dx - theta*delta1*dt*inner(grad(u0[0]),grad(v[0]))*dx + ((-u0[0]**2)+(u0[0]*u0[
# L1 = u0[1]*v[1]*dx - theta*delta2*dt*inner(grad(u0[1]),grad(v[1]))*dx + ((u0[0]*u0[1])/(u0[0]+a

# L0 = u0[0]*v[0]*dx - theta*delta1*dt*inner(grad(u0[0]),grad(v[0]))*dx + u0[0]*(1-u0[0])*v[0]*dx
# L1 = u0[1]*v[1]*dx - theta*delta2*dt*inner(grad(u0[1]),grad(v[1]))*dx + u0[0]*u0[1]*v[1]*dx

a = a0+a1
L = L0+L1
# set up boundary conditions
# Because this is a homogeneous Neumann problem, no boundary conditions
# are specified.
# bc = []

# assemble stiffness matrix
A = assemble(a)


# Set output file
out_file = File("results/poisson1.pvd","compressed")
# open txt file to write population in
population_file = open("results/preyPredPop.txt","w+")

# Set initial condition
u = Function(V)
u.assign(u0) #u = u0


u_initial = Function(V)
u_initial.assign(u0)

t_save = 0;
num_samples = 1000

t = 0.0 # initial time
# times to plot at
t0 = 0
t1 = 50
t2 = 100
t3 = 150
t4 = 500
t5 = 1000

# remember to apply the initial condition corresponding to each task.
task1 = True  # True = run task 1
task12 = False # True = run task 1.2: second instance of task 1
task2 = False  # True = run task 2


# initial solution, basically plotting IC
out_file<<(u,t)
# categories
population_file.write("Time, Prey, Predator\n")

# Timestepping
while t <= T:

    # u0 = u
    u0.assign(u)
```

```
# assemble mattrix and vector
b = assemble(L)

# solve linear system with LU factorization
solve(A,u.vector(),b,"lu", "default")

t_save += dt
# only save a certain amount of samples, might get big otherwise
if t_save > T/num_samples or t >= T-dt:
    print("saving solution")

    # save file
    out_file << (u,t)

    t_save = 0

# PRINTING
if ( (task1==True and (t == t0 or t == t1 or t == t2 or t == t3)) or ((task12 == True) and (t
    # plot solution prey
    plot(u[0])
    plt.title('Population density of prey at time t = %i' % t)
    plt.xlabel('X-direction')
    plt.ylabel('Y-direction')
    plt.show()
    # plot solution predators
    plot(u[1])
    plt.title('Population density of predators at time t = %i' % t)
    plt.xlabel('X-direction')
    plt.ylabel('Y-direction')
    plt.show()

# move to next timestep and adjust boundary condition.
t = t + dt

# compute the functional
population_u = assemble(u[0] * dx)
population_v = assemble(u[1] * dx)


# print population for prey and predator
print("",t ,", ",population_u,", ",population_v)
tempstring = "" + str(t) + ", " + str(population_u) + ", " + str(population_v) + "\n"
population_file.write(tempstring)
```