Supplementary Material

for the paper entitled

"*On the degradation of forest ecosystems by extreme events: Statistical Model Checking of a hybrid model*"

by Guillaume Cantin, Benoît Delahaye and Beatriz M. Funatsu

```
/*****************************************************************************************

        Supplementary material for the paper entiled

        "On the degradation of forest ecosystems by extreme climatic events:
        statistic verification of a hybrid model"

        by Guillaume Cantin, Beatriz M. Funatsu, Benoît Delahaye.

        ---------------------------------------------------------------------------------

        The present file is a computation code written in the FreeFem++ language,
        for simulating the hybrid model (H) reproducing the dynamics of the forest ecosystem
        under the perturbation of extreme climatic events.
        Therefore, it should be executed with FreeFem++.

        For more information about FreeFem++, see https://freefem.org/.

        ---------------------------------------------------------------------------------

        The computation code is based on the Strang splitting method,
        with a Runge-Kutta scheme for the time discretization and
        a finite elements scheme for the space discretization.
        For more information about the Strang splitting method,
        see Strang, On the construction and comparison of difference schemes (1968).
        For more information on the Runge-Kutta scheme and the finite elements scheme,
        see Allaire, Numerical analysis and optimization:
        an introduction to mathematical modelling and numerical simulation (2007).

*****************************************************************************************/


// Parameters of the extreme climatic events
real p = 0.2;           // Probability of an extreme event
real intensity = 0.5;   // Intensity of an extreme event
int step = 1;           // Time step between two possible extreme events

// Simulation of a Bernoulli variable
func int Bernoulli(real p)
{
        real DE;
        DE = randreal1();
        return (DE<p);
}



// Parameters of the reaction-diffusion system
// a, b, c, r, f, h are taken from Antonovsky et al. (1990)
real a = 0.006;
real b = 0.247;
real c = 0.01;          // gamma(v) = a*(v-b)^2+c
real r = 0.134;
real f = 0.017;
real h = 0.04;

real alpha = 0.5;
```

```
real beta = 0.5;
real delta = 0.268;     // delta is chosen so that beta*delta = r

// Stationary solution
real D = (f*alpha*delta - h*(c+f))/(a*h);       // D ~= 4.9917
real uplus = h*(b+sqrt(D))/f;                   // u+ ~= 5.8381
real vplus = b+sqrt(D);                         // v+ ~= 2.4812
real wplus = alpha*(b+sqrt(D))/beta;            // w+ ~= 2.4812
real h1 = f*alpha*delta/(a*b*b + c + f);        // h1 ~= 0.0832

// Diffusion coefficient of seeds
real d = 50.0;

// Time discretization parameters
real dt = 0.1;
real dtau = dt/2.0;

// Domain Omega (disk of radius L)
real L = 50.0;
border C(t=0,2*pi){x=(L/2)*(cos(t))+(L/2); y=(L/2)*(sin(t))+(L/2);}
mesh Th = buildmesh(C(30));
fespace Vh(Th, P1);

// Reaction part of the model
func real[int] fct(real[int] X, real t)
{
        // X[0] -> u
        // X[1] -> v
        // X[2] -> w

        real[int] dX(3);
        dX = 1.0;
        dX[0] = beta*delta*X[2] - (a*(X[1]-b)*(X[1]-b)+c)*X[0] - f*X[0];
        dX[1] = f*X[0] - h*X[1];
        dX[2] = -beta*X[2] + alpha*X[1];
        return dX;
}

// Runge-Kutta method
func real[int] RK4(real[int] u, real t)
{
        real[int] u2(3), u3(3), u4(3);
        real[int] p1(3), p2(3), p3(3), p4(3);
        real t2, tnew;
        real[int] unew(3);
        unew = 1.0;

        p1 = fct(u, t);
        t2 = t + 0.5*dt;
        u2 = u + 0.5*dt*p1;
        p2 = fct(u2, t2);
        u3 = u + 0.5*dt*p2;
        p3 = fct(u3, t2);
        tnew = t + dt;
        u4 = u + dt*p3;
        p4 = fct(u4, tnew);
        unew = 1.0*p1;
        unew = unew + 2.0*p2;
        unew = unew + 2.0*p3;
        unew = unew + 1.0*p4;
        unew = u + dt*1./6.*unew;
        return unew;
}

// Unknowns of the problem
Vh unew, uold, vnew, vold, wnew, wold, phi;

// Trace of the simulation

// Distance to the persistence equilibrium P+
func real trace1(){
```

```
        int MAXINT = 1000;
        real calcul = 0.0;
        for (int i=0; i<Th.nt; i++){
                for (int j=0; j<3; j++){
                        calcul = calcul + (uold[][Vh(i,j)] - uplus)*(uold[][Vh(i,j)] - uplus);
                        calcul = calcul + (vold[][Vh(i,j)] - vplus)*(vold[][Vh(i,j)] - vplus);
                        calcul = calcul + (wold[][Vh(i,j)] - wplus)*(wold[][Vh(i,j)] - wplus);
                }
        }
        calcul = sqrt(calcul);
        return calcul;
}

// Distance to the extinction equilibrium 0
func real trace2(){
        int MAXINT = 1000;
        real calcul = 0.0;
        for (int i=0; i<Th.nt; i++){
                for (int j=0; j<3; j++){
                        calcul = calcul + (uold[][Vh(i,j)])*(uold[][Vh(i,j)]);
                        calcul = calcul + (vold[][Vh(i,j)])*(vold[][Vh(i,j)]);
                        calcul = calcul + (wold[][Vh(i,j)])*(wold[][Vh(i,j)]);
                }
        }
        calcul = sqrt(calcul);
        return calcul;
}

// Diffusion part of the model
problem Diffusionu(unew, phi) = int2d(Th)( unew*phi + 0.0*dtau*(dx(unew)* dx(phi) + dy(unew)* dy(phi)))
                        + int2d(Th) (- uold*phi );
problem Diffusionv(vnew, phi) = int2d(Th)( vnew*phi + 0.0*dtau*(dx(vnew)* dx(phi) + dy(vnew)* dy(phi)))
                        + int2d(Th) (- vold*phi );
problem Diffusionw(wnew, phi) = int2d(Th)( wnew*phi + d*dtau*(dx(wnew)* dx(phi) + dy(wnew)* dy(phi)))
                        + int2d(Th) (- wold*phi );

real t = 0.0;

// Initial condition (small perturbation of the persistence equilibrium)
func real ci(real x, real y)
{
        real res;
        res = 0.1/(1+0.1*(x-(3*L/4))*(x-(3*L/4))+0.1*(y-(L/2))*(y-(L/2)));
        return res;
}

// Splitting scheme (Strang method)
for (int nbExec=0; nbExec<1200; nbExec++){

                // Initial condition
                uold = uplus + ci(x,y);
                vold = vplus + ci(x,y);
                wold = wplus + ci(x,y);

                for (int m=0; m<100/dt; m++)
                {
                        // Diffusion for half-step
                        Diffusionu;
                        uold = unew;
                        Diffusionv;
                        vold = vnew;
                        Diffusionw;
                        wold = wnew;
                        t = t + dtau;

                        // Reaction for one step
                        func real[int] reaction(real x, real y)
                                {
                                        real[int] U(3);
                                        U[0] = uold;
                                        U[1] = vold;
```

```
                                U[2] = wold;
                                U = RK4(U, t);
                                return U;
                        }

                uold = reaction(x, y)[0];
                vold = reaction(x, y)[1];
                wold = reaction(x, y)[2];
                t = t + dt;

                // Diffusion for half-step
                Diffusionu;
                uold = unew;
                Diffusionv;
                vold = vnew;
                Diffusionw;
                wold = wnew;
                t = t + dtau;

                // Localized extreme event
                if (m%step == 0){
                        if (Bernoulli(p)==1){
                                // If an extreme event occurs:
                                real xee = -1.0+2.0*randreal1();
                                real yee = -1.0+2.0*randreal1();
                                xee = xee/sqrt(2);
                                yee = yee/sqrt(2);
                                xee = (L/2.0) + (L/2.0)*xee;
                                yee = (L/2.0) + (L/2.0)*yee;
                                real ree = 3.0;

                                // New initial conditions
                                func real uee(real x, real y){
                                        real res;
                                        if ((x-xee)*(x-xee)+(y-yee)*(y-yee)<ree){
                                                res = uold(x, y)*(1-intensity);
                                        }
                                        else{
                                                res = uold(x,y);
                                        }
                                        return res;
                                }
                                func real vee(real x, real y){
                                        real res;
                                        if ((x-xee)*(x-xee)+(y-yee)*(y-yee)<ree){
                                                res = vold(x, y)*(1-intensity);
                                        }
                                        else{
                                                res = vold(x,y);
                                        }
                                        return res;
                                }
                                func real wee(real x, real y){
                                        real res;
                                        if ((x-xee)*(x-xee)+(y-yee)*(y-yee)<ree){
                                                res = wold(x, y)*(1-intensity);
                                        }
                                        else{
                                                res = wold(x,y);
                                        }
                                        return res;
                                }
                                uold = uee(x, y);
                                vold = vee(x, y);
                                wold = wee(x, y);
                        }
                }
                cout << "(" << trace1() << ", " << trace2() << ")" << endl;
        }
}
cout << "END OF THE PROGRAM" << endl;
```