README.txt

The MATLAB script

run_simulation.m

produces the plots in Figures 9 and 11 of the paper.

The code has been tested on a MATLAB R2021a distribution. On a 4 GHz Quad-Core Intel Core i7 system with 16 GB 1867 MHz DDR3, it took 50 minutes till completion.

run_simulation.m

```
% This script runs the full simulation and produces the
% figures in the article "Stable Partial Cooperation in
% Managing Systems with Tipping Points"

% Direct tipping b = 0.6 ---------------------------------------------

par.b    =  0.60;  % sedimentation & outflow rate
par.r    =  0.03;  % discount rate
par.n    = 10;    % number of players
par.s0   =  0.00; % initial state for tipping game
par.smax = 100.0;

nsteps = 500;
c1     = 2.0;
dc     = c1/nsteps;
cval1   = dc:dc:c1;

size1 = compute_coalition_sizes(par,cval1);


% Direct tipping b = 0.52 --------------------------------------------

par.b = 0.52;

nsteps = 500;
c1     = 2.0;
dc     = c1/nsteps;
cval2   = dc:dc:c1;

compute_coalition_sizes(par,cval2);

% Inverse tipping ----------------------------------------------------

par.b  =  0.60;
par.s0 =  5.00; % initial state for inverse tipping game

nsteps = 500;
c1     = 7.0;
dc     = c1/nsteps;
cval3   = dc:dc:c1;

size3 = compute_coalition_sizes(par,cval3);
```

```
compute_coalition_sizes.m
--------------------------------------------------------------------------
% function [out, vmem, vnash] = compute_coalition_sizes(par0, cval)
%
% Compute stable coalition sizes of the dynamic tipping game over a given
% range of c parameter values
%
% INPUT
% structure par0 : system parameters
% double()  cval : array of c parameter values
%
% OUTPUT
% double(length(cval)) out
%                : vector of stable coalition sizes for given value of c
% double(length(cval),par0.n) vmem
%                : matrix of welfare of coalition members, for given value
%                of c and given coalition size
% double(length(cval),par0.n) vnash
%                : matrix of welfare of outsiders, for given value of c and
%                given coalition size

function [out, vmem, vnash] = compute_coalition_sizes(par0, cval)

n      = par0.n;
runname = strcat('coalition_size' ,...
    '_n=' ,sprintf('%2d'  ,par0.n) ,...
    '_b=' ,sprintf('%4.2f',par0.b) ,...
    '_r=' ,sprintf('%5.3f',par0.r) ,...
    '_s0=',sprintf('%4.2f',par0.s0));

% Initialisations
len      = length(cval);
vnash    = zeros(len,n);
vmem     = zeros(len,n);
vcoal    = zeros(len,1);
size     = zeros(len,1);
par      = repmat(par0, len, 1);
eqstring = strings(len, n);

% Main iteration over cval
parfor i=1:len
    par(i).c = cval(i); %#ok<PFOUS>
    fprintf('\nc = %10.6f\n', par(i).c)
% iteration of the effective number neff = n-k+1 of players:
    % n-k outsiders and 1 coalition
    for neff = 1:n
```

```
        [vnash(i,neff), eqstring(i,neff)] = ...
            value_ol_nash(par(i),neff); %#ok<PFOUS>
        vmem (i,neff) = vnash(i,neff) - log(n-neff+1)/par(i).r;
        k=n-neff+1;
        fprintf('k = %2d, vmem(%2d) = %7.2f, vnash(%2d) = %7.2f. %39s\n', ...
                  k, k, vmem(i,neff), k, vnash(i,neff), eqstring(i,neff) )
    end
end

% Computation of largest stable coalitions
disp(' ')
for i=1:length(cval)
    neff = 1;
    while ( (neff < n) && (vmem(i,neff) < vnash(i,neff+1) ) )
        neff = neff+1;
    end
    size(i)  = n-neff+1;
    vcoal(i) = vmem(i,neff);
    fprintf('c = %6.2f, stable coalition size = %d\n', cval(i), size(i))
end
out = size';

% Side effects: generate figures and data files
figure(1);clf;hold on;
axis([min(cval) max(cval) 0 n])
plot(cval,size,'.')

m  = 1;
v0 = vcoal(m);
v1 = vnash(m,n);
fprintf(...
    '\n c = %8.4f: vnash = %10.4f, vcoal = %10.4f\n', ...
    cval(m), v1, v0)
m  = len;
v0 = vcoal(m);
v1 = vnash(m,n);
fprintf(...
    '\n c = %8.4f: vnash = %10.4f, vcoal = %10.4f\n', ...
    cval(m), v1, v0)

print(strcat(runname,'.eps'),'-deps')
print(strcat(runname,'.pdf'),'-dpdf','-bestfit')
save (strcat(runname,'.dat'),'vmem','vnash','cval','out',...
    '-ascii','-tabs')

end
```

df.m

```
--------------------------------------------------------------------------------
% function y = df(s,b)
%
% Derivative of state response function
%
% INPUT
% double s : current state
% double b : sedimentation rate
%
% OUTPUT
% double y : state derivative of state response function

function y = df(s,b)

    y = b - 2 * s .* (1d0 + s.*s).^(-2);

end
```

f.m

```
------------------------------------------------------------------------
% function y = f(s,b)
%
% State response function
%
% INPUT
% double s : current state
% double b : sedimentation rate
%
% OUTPUT
% double y : uncontrolled state change rate

function y = f(s,b)

    s2 = s.*s; y = b*s - s2./(1d0+s2);

end
```

```
find_nash_steady_states.m
--------------------------------------------------------------------------
% function out = find_nash_steady_states(par,n)
%
% Finds the steady state values of the candidate Nash equilibria
%
% INPUT
% structure    par : system parameters
% integer      n   : number of players
%
% OUTPUT
% double()     out : array containing saddle steady states

function out = find_nash_steady_states(par,n)

% Step 1: find intervals containing roots of s -> g(s,par,n) by looking for
% sign changes of g
s = 0:0.001:40;
p = sign(g(s,par,n));
p = 2*(p>-0.5)-1;     % change p=0 to p=+1
l = length(p);
q = p(2:l)-p(1:l-1);
ieq = find(q);
eq = zeros(1,length(ieq));

% Step 2: use rootfinding to obtain high precision approximation of roots
options = optimset('Display','none','TolX',1e-14);
fun = @(s) g(s,par,n);
for i=1:length(ieq)
    s0 = [s(ieq(i)) s(ieq(i)+1)];
    eq(i) = fzero(fun,s0,options);
end

% Step 3: check number of roots and order roots by size
if (length(eq)==3)
    eq = [eq(1); eq(3)];
elseif (length(eq)~=1)
    % Generically there should be either 1 or 3 steady states. If that is
    % not the case, stop the computation and find out what is going on by
    % hand.
    disp('Wrong number of Nash candidate steady states')
    stop
end

out = sort(eq);
```

```
end

function y = g(s, par, n)

b = par.b; c = par.c; r = par.r;

y = -(r + df(s,b)) + 2 * (c/n) * s .* f(s,b);

end
```

```
find_nash_deviation_steady_state.m
--------------------------------------------------------------------------
% function [out,err] = find_nash_deviation_steady_states(par,n,eq)
%
% Finds the steady state values of a large deviation from the candidate
% Nash equilibrium
%
% INPUT
% structure    par : system parameters
% integer      n   : number of players
% double       eq  : Nash equilibrium steady state
%
% OUTPUT
% double       out : deviation saddle steady state
% integer      err : error code
%                 0 - computation succeeded
%                 1 -
%                 2 -

function [out,err] = find_nash_deviation_steady_state(par,n,eq)

err = 0;

seq = eq; Aeq = f(seq, par.b);

% Step 1: find intervals containing roots of s -> h(x,Aeq,par,n) by
% looking for sign changes of h
x = 0:0.001:10;
p = sign(h(x,Aeq,par,n));
p = 2*(p>-0.5)-1;   % change p=0 to p=+1
l = length(p);
q = p(2:l)-p(1:l-1);
ieq = find(q);
eq = zeros(1,length(ieq));

% Step 2: use rootfinding to obtain high precision approximation of roots
options = optimset('Display','none','TolX',1e-14);
fun = @(x) h(x,Aeq,par,n);
for i=1:length(ieq)
    x0 = x(ieq(i));
    eq(i) = fzero(fun,x0,options);
end

% Step 3: check number of roots and select the correct one
if (length(eq)==3)
     out = eq(3);
```

```
elseif (length(eq)==1)
     out = eq(1);
     err = 1;
else
    disp('ieq = ')
    disp(ieq)
    disp('eq = ')
    disp(eq)
    error('Wrong number of deviating steady states')
end

end

function out = h(s,A,par,n)

    b = par.b; c = par.c; r = par.r;

    out = (r + df(s,b))./(2*c*s) - A/n + A - f(s,b);

end
```

```
value_ol_nash.m
--------------------------------------------------------------------------------
% function [out, eqstring] = value_ol_nash(par,n)
%
% Computes welfare-maximal Nash equilibrium value in the n-player game
%
% INPUT
% structure par : problem parameters
% integer   n   : number of players
%
% OUTPUT
% real      out      : welfare of player in Nash equilibrium
% string    eqstring : verbal description equilibrium

function [out, eqstring] = value_ol_nash(par,n)

eq = find_nash_steady_states(par,n);

len = length(eq);
if len==1
    % Unique candidate equilibrium steady state
    out = value(par,n,eq);
    eqstring = 'unique';
else
    % Several candidate equilibrium steady states
    v   = zeros(len,1);
    err = zeros(len,1);
    for i=1:len
        % Compute candidate Nash values
        [v(i), err(i)] = value(par,n,eq(i));
        if err(i) == 2
            fprintf('c = %6.1f\nEquilibria: \n', par.c)
            disp(eq)
            error('Wrong eigenvalues')
        end
    end

    if err(2) ~= 0
        % Computation of eutrophic equilibrium failed
        out = v(1);
        eqstring = 'oligotrophic unique';
    elseif err(1) ~= 0
        % Computation of oligotrophic equilibrium failed
        out = v(2);
        eqstring = 'eutrophic unique';
    elseif v(2) > v(1)
```

```
        out = v(2);
        eqstring = 'eutrophic outranks oligotrophic';
    else
        % Oligotrophic outranks eutrophic:
        % test for Nash by computing the value of a large
        % deviation
        [eqdev, errdev] = find_nash_deviation_steady_state(par,n,eq(1));
        if errdev == 1
            % Computation of deviation failed
            out = v(1);
            eqstring = 'oligotrophic outranks eutrophic';
        else
            [w,errw] = value_deviation(par,n,eq(1),eqdev);
            if errw~=0 || (w<=v(1))
                out = v(1);
                eqstring = 'oligotrophic outranks eutrophic';
            else
                out = v(2);
                eqstring = 'oligotrophic not stable under deviation';
            end
        end
    end
end

end
```

value.m

```
% function [val, err] = value(par,n,eq)
%
% Computes welfare-maximal Nash equilibrium value in the n-player game
%
% INPUT
% structure par : problem parameters
% integer   n   : number of players
% double    eq  : steady state value of candidate Nash equilibrium
%
% OUTPUT
% double    val : welfare of player in Nash equilibrium
% string    err : error code
%                 0 - computation succeeded
%                 1 - state trajectory bends back on itself
%                 2 - complex eigenvalues at steady state

function [val, err] = value(par,n,eq)

% Intialisations ----------------------------------------------------------
err = 0;

b=par.b;
c=par.c;
r=par.r;

seq = eq;
Aeq = f(seq,b);
veq = (log(Aeq/n) - c * seq.^2)/r;
xeq = [seq;Aeq;veq];

% Numerical approximation of Jacobian matrix vector field at steady state
h   = 1d-6;
e   = eye(3);
dvf = zeros(3);

for i=1:3
    x = xeq + h * e(:,i);
    dvf(:,i) = vf_nash(x,par,n)/h;
end

[ev,d]=eig(dvf);

i=1;
try
```

```
    while (d(i,i)>0)
        i=i+1;
    end
catch
% If the while loop throws an error, one eigenvalue is complex. This
% is an error
    fprintf('\n\nc = %6.1f, s_eq = %7.2f\n', par.c, seq)
    disp(d)
    err = 2;
end

if err == 2
    val = 0;
else
    if i<=3
% Normalisation to ensure that the first component of the
% eigenvectors is 1
        es = ev(:,i)/ev(1,i);
    else
% If i=4, all eigenvalues are positive, and the steady state is
% not a saddle. This should not happen.
        disp('value.m: something is wrong with the eigenvalues')
    end

    options = odeset('Events',@events,'RelTol',1e-7,'AbsTol',1e-9);
    if par.s0 < seq
% If the initial value is less than the steady state, integrate
% the stable manifold to the left, otherwise to the right
        x0 = xeq - h * es;
    else
        x0 = xeq + h * es;
        par.smax = par.s0;
    end

    odefun = @(t,x) (- vf_nash(x,par,n));
    tspan  = [0 1000];
    try
        [~,~,~,xe,ie] = ode45(odefun, tspan, x0, options);
        if ie==3
% ds/dt = 0: backbending trajectory, not a valid solution
            val = -1d10;
            err = 1;
        else
            val = xe(3);
        end
    catch ERR
```

```
        rethrow(ERR)
        error('Value: Problem integrating the Nash system')
    end
end

% function [value,isterminal,direction] = events(~,x)
%
% Throws event if s(t) leaves state interval or if ds/dt(t)=0

function [value,isterminal,direction] = events(~,x)

    y = vf_nash(x,par,n);

    value = [x(1); par.smax-x(1); y(1)];
    isterminal = [1; 1; 1];
    direction  = [0; 0; 0];

end
end
```

```
value_deviation.m
-------------------------------------------------------------------------
% function [out, err] = value_deviation(par,n,eq,eqdev)
%
% Computes value of a single player deviation from a candidate Nash
% equilibrium in the n-player game
%
% INPUT
% structure par   : problem parameters
% integer   n     : number of players
% double    eq    : candidate Nash equilibrium steady state
% double    eqdev : single player deviation steady state
%
% OUTPUT
% double    out : value of single player deviation
% string    err : error code
%                 0 - computation succeeded
%                 1 - solution error exceeds 1d-7
%                 2 - boundary value solver fails to converge

function [out,err] = value_deviation(par,n,eq,eqdev)

% Initialisations ---------------------------------------------------------

err = 0;
b = par.b; c = par.c; r = par.r; s0 = par.s0;

s     = eq;
sigma = eqdev;
A     = f(s,b);
ai    = (r + df(sigma,b))/(2*c*sigma);
veq   = (log(A/n) - c * s.^2)/r;
weq   = (log(ai)  - c * sigma.^2)/r;

xeq=[s;A;veq;sigma;ai;weq];

% Solution of boundary value problem --------------------------------------

warning('off','MATLAB:bvp5c:RelTolNotMet')

bcv = @(mu) ((1-mu)*[xeq(1); xeq(4); xeq(1); xeq(4); xeq(3); xeq(6)] ...
             +  mu *[    s0;     s0; xeq(1); xeq(4); xeq(3); xeq(6)]);
xv = bcv(0);

t      = 0:100:1000;
solinit = bvpinit(t, xeq);
```

```
odefun = @(t,x)   (vf_deviation(x,par,n));
bcfun  = @(xa,xb) [xa(1); xa(4); xb(1); xb(4); xb(3); xb(6)] - xv;
options = bvpset(...
        'NMax',1000,'RelTol',1e-7,'AbsTol',1e-9, ...
        'Vectorized','on','Stats','off');
sol     = bvp5c(odefun, bcfun, solinit, options);

% Homotopy in four steps from steady state to final solution

nsteps = 4; i=1;
while (i<=nsteps) && (err==0)
    xv = bcv(i/nsteps);
    bcfun  = @(xa,xb) [xa(1); xa(4); xb(1); xb(4); xb(3); xb(6)] - xv;

    try
        sol = bvp5c(odefun, bcfun, sol, options);
        if sol.stats.maxerr > 1d-7
            err = 1;
        end
        i = i + 1;
    catch
        err = 2;
    end
end

if err~=0
    % If four-step homotopy fails, try twenty-step homotopy
    err = 0; figure(2); clf; hold on

    xv = bcv(0);
    bcfun  = @(xa,xb) [xa(1); xa(4); xb(1); xb(4); xb(3); xb(6)] - xv;
    options = bvpset(...
        'NMax',2000,'RelTol',1e-7,'AbsTol',1e-9, ...
        'Vectorized','on','Stats','off');
    sol    = bvp5c(odefun, bcfun, solinit, options);


    mu = [0.2:0.2:0.6 0.61:0.01:1];
    nsteps = length(mu);
    i = 1;
    while (i<=nsteps) && (err==0)
        xv = bcv(mu(i));
        bcfun  = @(xa,xb) [xa(1); xa(4); xb(1); xb(4); xb(3); xb(6)] - xv;
        try
            sol = bvp5c(odefun, bcfun, sol, options);
```

```
            if sol.stats.maxerr > 1d-7
                fprintf('Fine iteration, step : %2d',i)
                fprintf(', maxerr = %6.2e\n',sol.stats.maxerr)
                err = 1;
            end
            i = i + 1;
        catch
            err = 2;
        end
    end
end

warning('on','MATLAB:bvp5c:RelTolNotMet')

if err==0
    out = sol.y(6,1);
else
    out = -1d-10;
end

end
```

vf_nash.m

```
% function y = vf_nash(x,par,n)
%
% Computes right hand side of Nash equilibrium differential
% equations
%
% INPUT
% double(3) x   : state vector, consisting of state, total
%                 emissions, and value
% structure par : problem parameters
% integer   n   : number of players
%
% OUTPUT
% y             : time derivative of state vector under Nash equilibrium
%                 dynamics

function y = vf_nash(x,par,n)

b = par.b;
c = par.c;
r = par.r;

    s = x(1);
E = x(2);
v = x(3);

    y = [ E - f(s,b); ...
          (2 * (c/n) * s .* E -(r + df(s,b))) .* E; ...
          r * v - log(E/n) + c * s.^2];
end
```

vf_deviation.m

```
% function y = vf_deviation(x,par,n)
%
% Computes right hand side of Nash equilibrium deviation differential
% equations
%
% INPUT
% double(3) x   : state vector, consisting of state, total
%                 emissions, and value
% structure par : problem parameters
% integer   n   : number of players
%
% OUTPUT
% y             : time derivative of state vector under Nash equilibrium
%                 deviation dynamics

function y = vf_deviation(x,par,n)

    b = par.b;
    c = par.c;
    r = par.r;

    s     = x(1,:);
    E     = x(2,:);
    v     = x(3,:);
    sigma = x(4,:);
    ei    = x(5,:);
    w     = x(6,:);
    e     = E/n;

    y = [E - f(s,b); ...
        (2 * (c/n) * s .* E -(r + df(s,b))) .* E; ...
        r * v - log(e) + c * s.^2; ...
        (ei - e) + E - f(sigma,b); ...
        (2 * c * sigma .* ei -(r + df(sigma,b))) .* ei; ...
        r * w - log(ei) + c * sigma.^2];
end
```