Electronic Supplementary Material (ESI) for RSC Advances.

Supplementary data

Process integration for microalgal lutein and biodiesel production with concomitant flue gas CO_2 sequestration: A biorefinery model for healthcare, energy and environment

R. Dineshkumar [a], Sukanta Kumar Dash [b], Ramkrishna Sen [a,*]

[a] Department of Biotechnology, Indian Institute of Technology Kharagpur, India

[b] Department of Mechanical Engineering, Indian Institute of Technology Kharagpur, India

[*] Corresponding author: E–mail address: rksen@yahoo.com; Tel: +91–3222–283752

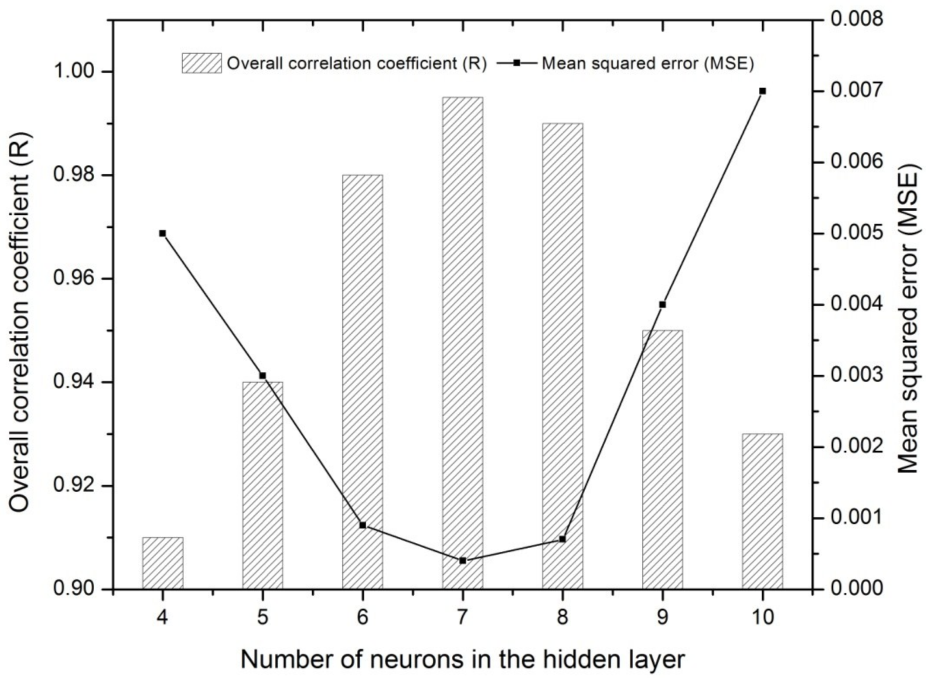

Fig.A.1. Effect of neuron number in the hidden layer on overall correlation coefficient and mean squared error

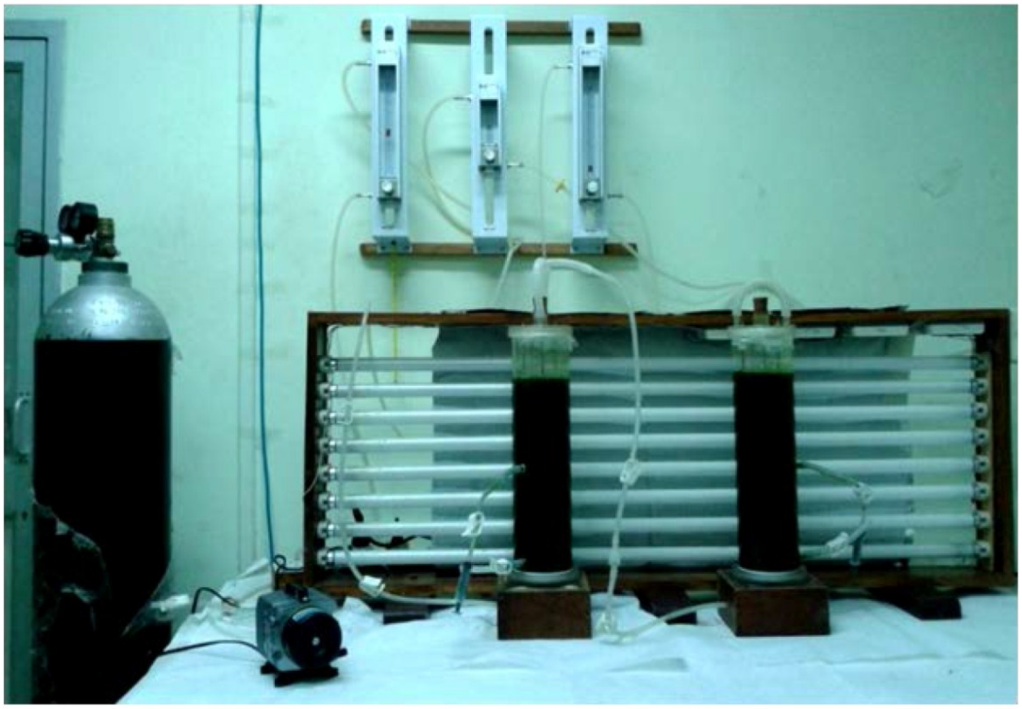

Fig.A.2. Experimental setup of CO_2 sequestration by *Chlorella minutissima* cultivated in a 2–L airlift photobioreactor under the optimized process conditions

Table A.1. Experimental range and levels of independent process variables

Process variables	Levels				
	-2	-1	0	1	2
Light intensity (μmol m^{-2} s^{-1})	50	100	175	250	300
CO_2 (%)	0.8	2.5	5	7.5	9.2
Flow rate (mL min^{-1})	395	600	900	1200	1404

Table A.2. The results of the validation experiments for the ANN model

Light intensity (μmol $m^{-2}s^{-1}$)	CO_2 (%)	Flow rate (mL min^{-1})	ANN-predicted lutein productivity (mg L^{-1} d^{-1})	Experimentally determined lutein productivity (mg L^{-1} d^{-1})	% Error between predicted and experimental values
200	1	1000	2.88	2.79 ± 0.04	3.2
150	4	900	3.19	3.28 ± 0.07	2.8
180	5	750	3.75	3.64 ± 0.05	3.0

Equation A.1

Response= –(6.10508) + (0.037626* Light) + (0.942398* CO_2) + (6.4016*10^{-03}* Flow rate) – (1.4533*10^{-03} Light * CO_2) + (4.889*10^{-06}* Light * Flow rate) + (1.266*10^{-04}* CO_2* Flow rate) – (6.2254*10^{-05}* Light^2) – (0.089687* CO_2^2) – (4.06765*10^{-06}* Flow rate^2)

PSO syntax:

```
function [xOpt,fval,exitflag,output,population,scores] = ...
    pso(fitnessfcn,nvars,Aineq,bineq,Aeq,beq,LB,UB,nonlcon,options)
% Find the minimum of a function using Particle Swarm Optimization.
%
% This is an implementation of Particle Swarm Optimization algorithm using
% the same syntax as the Genetic Algorithm Toolbox, with some additional
% options specific to PSO. Allows code-reusability when trying different
% population-based optimization algorithms. Certain GA-specific parameters
% such as cross-over and mutation functions will not be applicable to the
% PSO algorithm. Demo function included, with a small library of test
% functions. Requires Optimization Toolbox.
%
% In development, new features will be added regularly until this is made
% redundant by an official MATLAB PSO toolbox.
%
% S. Chen. Version 20100522.
% Available from http://www.mathworks.com/matlabcentral/fileexchange/25986
% Distributed under BSD license.
%
% Syntax:
% psodemo
% pso
% x = pso(fitnessfcn,nvars)
% x = pso(fitnessfcn,nvars,Aineq,bineq)
% x = pso(fitnessfcn,nvars,Aineq,bineq,Aeq,beq)
% x = pso(fitnessfcn,nvars,Aineq,bineq,Aeq,beq,LB,UB)
% x = pso(fitnessfcn,nvars,Aineq,bineq,Aeq,beq,LB,UB,nonlcon)
```

```
% x = pso(fitnessfcn,nvars,Aineq,bineq,Aeq,beq,LB,UB,nonlcon,options)
% x = pso(problem)
% [x, fval] = pso(...)
% [x, fval,exitflag] = pso(...)
% [x, fval,exitflag,output] = pso(...)
% [x, fval,exitflag,output,population] = pso(...)
% [x, fval,exitflag,output,population,scores] = pso(...)
%
% Description:
% psodemo
% Runs a demonstration of the PSO algorithm using test function specified
% by user.
%
% pso
% Runs a default demonstration using Rosenbrock's banana function.
%
% x = pso(fitnessfcn,nvars)
% Runs the particle swarm algorithm with no constraints and default
% options. fitnessfcn is a function handle, nvars is the number of
% parameters to be optimized, i.e. the dimensionality of the problem.
%
% x = pso(fitnessfcn,nvars,Aineq,bineq)
% Linear constraints, such that Aineq*x <= bineq. Soft boundaries only.
% Aineq is a matrix of size nconstraints x nvars, while b is a column
% vector of length nvars.
%
% x = pso(fitnessfcn,nvars,Aineq,bineq,Aeq,beq)
% Linear equality constraints, such that Aeq*x = beq. Soft boundaries only.
% If no inequality constraints exist, set Aineq and bineq to [].
%
% x = pso(fitnessfcn,nvars,Aineq,bineq,Aeq,beq,LB,UB)
% Lower and upper bounds definted as LB and UB respectively. Use empty
% arrays [] for A, b, Aeq, or beq if no linear constraints exist.
%
% x = pso(fitnessfcn,nvars,Aineq,bineq,Aeq,beq,LB,UB,nonlcon)
% Non-linear constraints. Nonlinear inequality constraints in the form c(x)
% <= 0 have now been implemented using 'soft' boundaries only. See the
% Optimization Toolbox documentation for the proper syntax for defining
% nonlinear constraints.
%
% x = pso(fitnessfcn,nvars,Aineq,bineq,Aeq,beq,LB,UB,nonlcon,options)
% Default algorithm parameters replaced with those defined in the options
% structure:
% Use >> options = psooptimset('Param1,'value1','Param2,'value2',...) to
% generate the options structure. Type >> psooptimset with no input or
% output arguments to display a list of available options and their
% default values.
%
% NOTE: the swarm will perform better if the PopInitRange option is defined
% so that it closely bounds the expected domain of the feasible region.
% This is automatically done for lower and upper bound constraints, but not
% for linear and nonlinear constraints.
%
% NOTE 2: If options.HybridFcn is to be defined, and if it is necessary to
% pass a non-default options structure to the hybrid function, then the
% options structure may be included in a cell array along with the pointer
% to the hybrid function. Example:
```

```
% >> % Let's say that we want to use fmincon to refine the result from PSO:
% >> hybridoptions = optimset(@fmincon) ;
% >> options.HybridFcn = {@fmincon, hybridoptions} ;
%
% x = pso(problem)
% Parameters imported from problem structure. Should work, but no error
% checking yet.
%
% [x, fval] = pso(...)
% Returns the fitness value of the best solution.
%
% [x, fval,exitflag] = pso(...)
% Returns information on outcome of pso run. Should match exitflag values
% for GA where applicable, for code reuseability.
%
% [x, fval,exitflag,output] = pso(...)
% The structure output contains more detailed information about the PSO
% run. Should match output fields of GA, where applicable.
%
% [x, fval,exitflag,output,population] = pso(...)
% A matrix population of size PopulationSize x nvars, with the locations of
% particles across the rows.
%
% [x, fval,exitflag,output,population,scores] = pso(...)
% Final scores of the particles in population.
%
% See also:
% PSODEMO, PSOOPTIMSET.

if ~nargin % Rosenbrock's banana function by default, as a demonstration
    fitnessfcn = @(x)100*(x(2)-x(1)^2)^2+(1-x(1))^2 ;
    nvars = 2 ;
    LB = [-1.5,-2] ;
    UB = [2,2] ;
    options.PopInitRange = [[-2;4],[-1;2]] ;
    options.PlotFcns = {@psoplotbestf,@psoplotswarmsurf} ;
    options.Generations = 200 ;
    options.DemoMode = 'on' ;
    options.KnownMin = [1 1] ;
elseif isstruct(fitnessfcn)
    nvars = fitnessfcn.nvars ;
    Aineq = fitnessfcn.Aineq ;
    bineq = fitnessfcn.bineq ;
    Aeq = fitnessfcn.Aeq ;
    beq = fitnessfcn.beq ;
    LB = fitnessfcn.LB ;
    UB = fitnessfcn.UB ;
    nonlcon = fitnessfcn.nonlcon ;
    if ischar(nonlcon) && ~isempty(nonlcon)
        nonlcon = str2func(nonlcon) ;
    end
    options = fitnessfcn.options ;
   % options.PlotFcns = {@psoplotbestf,@psoplotswarmsurf} ;
    fitnessfcn = fitnessfcn.fitnessfcn ;
elseif nargin < 2
    msg = 'PSO requires at least two input arguments' ;
    error('%s, or a problem structure. Type >> help pso for details',...
```

```matlab
        msg)
end % if ~nargin

if ~exist('options','var') % Set default options
    options = struct ;
end % if ~exist

options = psooptimset(options) ;

options.Verbosity = 1 ; % For options.Display == 'final' (default)
if strncmpi(options.Display,'off',3)
    options.Verbosity = 0 ;
elseif strncmpi(options.Display,'iter',4)
    options.Verbosity = 2 ;
elseif strncmpi(options.Display,'diag',4)
    options.Verbosity = 3 ;
end

if ~exist('Aineq','var'), Aineq = [] ; end
if ~exist('bineq','var'), bineq = [] ; end
if ~exist('Aeq','var'), Aeq = [] ; end
if ~exist('beq','var'), beq = [] ; end
if ~exist('LB','var'), LB = [] ; end
if ~exist('UB','var'), UB = [] ; end
if ~exist('nonlcon','var'), nonlcon = [] ; end
% Check for constraints and bit string population type
if strncmpi(options.PopulationType,'bitstring',2)
    if ~isempty([Aineq,bineq]) || ~isempty([Aeq,beq]) || ...
            ~isempty(nonlcon) || ~isempty([LB,UB])
        Aineq = [] ; bineq = [] ; Aeq = [] ; beq = [] ; nonlcon = [] ;
        LB = [] ; UB = [] ;
        msg = sprintf('Warning: Constraints will be ignored') ;
        msg = sprintf('%s for options.PopulationType ''bitstring''',msg) ;
        disp(msg)
    end
end
% Change this when nonlcon gets fully implemented:
if ~isempty(nonlcon) && strcmpi(options.ConstrBoundary,'reflect')
    msg = 'Non-linear constraints don''t have ''reflect'' boundaries' ;
    warning('pso:main:nonlcon',...
        '%s implemented. Changing options.ConstrBoundary to ''soft''.',...
        msg)
    options.ConstrBoundary = 'soft' ;
end

% Is options.PopInitRange reconcilable with LB and UB constraints?
% -------------------------------------------------------------------------
% Resize PopInitRange in case it was given as one range for all dimensions
if size(options.PopInitRange,2) == 1 && nvars > 1
    options.PopInitRange = repmat(options.PopInitRange,1,nvars) ;
end

% Check initial population with respect to bound constraints
% Is this really desirable? Maybe there are some situations where the user
% specifically does not want an uniform inital population covering all of
% LB and UB?
if ~isempty(LB) || ~isempty(UB)
```

```matlab
    options.LinearConstr.type = 'boundconstraints' ;
    options.PopInitRange = ...
        psocheckpopulationinitrange(options.PopInitRange,LB,UB) ;
end
% ------------------------------------------------------------------------

% Check validity of VelocityLimit
if all(~isfinite(options.VelocityLimit))
    options.VelocityLimit = [] ;
elseif isscalar(options.VelocityLimit)
    options.VelocityLimit = repmat(options.VelocityLimit,1,nvars) ;
elseif ~isempty(length(options.VelocityLimit)) && ...
        ~isequal(length(options.VelocityLimit),nvars)
    msg = 'options.VelocityLimit must be either a positive scalar' ;
    error('%s, or a vector of size 1xnvars.',msg)
end % if isscalar
options.VelocityLimit = abs(options.VelocityLimit) ;

% Generate swarm initial state (this line must not be moved)
if strncmpi(options.PopulationType,'double',2)
    state = psocreationuniform(options,nvars) ;
elseif strncmpi(options.PopulationType,'bi',2)
    state = psocreationbinary(options,nvars) ;
end

% Check initial population with respect to linear and nonlinear constraints
% ------------------------------------------------------------------------
if ~isempty(Aeq) || ~isempty(Aineq) || ~isempty(nonlcon)
    options.LinearConstr.type = 'linearconstraints' ;
    if ~isempty(nonlcon)
        options.LinearConstr.type = 'nonlinearconstraints' ;
    end
    if strcmpi(options.ConstrBoundary,'reflect')
        options.ConstrBoundary = 'soft' ;
        msg = sprintf('Constraint boundary behavior ''reflect''') ;
        msg = sprintf('%s is not yet supported for linear constraints.',...
            msg) ;
        msg = sprintf('%s Switching boundary behavior to ''soft''.',msg) ;
        warning('pso:mainfcn:constraintbounds',...
            '%s',msg)
    end
    [state,options] = psocheckinitialpopulation(state,...
        Aineq,bineq,Aeq,beq,...
        LB,UB,...
        nonlcon,...
        options) ;
end
% ------------------------------------------------------------------------

n = options.PopulationSize ;
itr = options.Generations ;

if ~isempty(options.PlotFcns)
    close(findobj('Tag','Swarm Plots','Type','figure'))
    state.hfigure = figure('NumberTitle','off',...
        'Name','PSO Progress',...
        'Tag','Swarm Plots') ;
```

```
end % if ~isempty

if options.Verbosity > 0, fprintf('\nSwarming...'), end
exitflag = 0 ; % Default exitflag, for max iterations reached.
flag = 'init' ;

% Iterate swarm
state.fitnessfcn = fitnessfcn ;
state.LastImprovement = 1 ;
state.ParticleInertia = 0.9 ; % Initial inertia
% alpha = 0 ;
for k = 1:itr
    state.Score = inf*ones(n,1) ; % Reset fitness vector
    state.Generation = k ;
    state.OutOfBounds = false(options.PopulationSize,1) ;

    % Check bounds before proceeding
    % ---------------------------------------------------------------------
    if ~all([isempty([Aineq,bineq]), isempty([Aeq,beq]), ...
            isempty([LB;UB]), isempty(nonlcon)])
        state = psocheckbounds(options,state,Aineq,bineq,Aeq,beq,...
            LB,UB,nonlcon) ;
    end % if ~isempty
    % ---------------------------------------------------------------------

    % Evaluate fitness, update the local bests
    % ---------------------------------------------------------------------
    if strcmpi(options.Vectorized,'off')
        for i = setdiff(1:n,find(state.OutOfBounds))
            state.Score(i) = fitnessfcn(state.Population(i,:)) ;
        end % for i
    else % Vectorized fitness function
        state.Score(setdiff(1:n,find(state.OutOfBounds))) = ...
            fitnessfcn(state.Population(setdiff(1:n,...
            find(state.OutOfBounds)),:)) ;
    end % if strcmpi

    betterindex = state.Score < state.fLocalBests ;
    state.fLocalBests(betterindex) = state.Score(betterindex) ;
    state.xLocalBests(betterindex,:) = ...
        state.Population(betterindex,:) ;
    % ---------------------------------------------------------------------

    % Update the global best and its fitness, then check for termination
    % ---------------------------------------------------------------------
    [minfitness, minfitnessindex] = min(state.Score) ;

%     alpha = alpha + (1/k) * ...
%         ((1/n)*sum((state.Velocities*state.Velocities')^2) ./ ...
%         ((1/n)*sum(state.Velocities*state.Velocities')).^2) ;
%     tempchk = alpha <= 1.6 ;
 %   fprintf('\nFitness = %f',state.fGlobalBest);
    if minfitness < state.fGlobalBest
        state.fGlobalBest(k) = minfitness ;
        state.xGlobalBest = state.Population(minfitnessindex,:) ;
        state.LastImprovement = k ;
        imprvchk = k > options.StallGenLimit && ...
```

```
            (state.fGlobalBest(k - options.StallGenLimit) - ...
                state.fGlobalBest(k)) / (k - options.StallGenLimit) < ...
                options.TolFun ;
        if imprvchk
            exitflag = 1 ;
            flag = 'done' ;
        elseif state.fGlobalBest(k) < options.FitnessLimit
            exitflag = 2 ;
            flag = 'done' ;
        end % if k
    else % No improvement from last iteration
        state.fGlobalBest(k) = state.fGlobalBest(k-1) ;
    end % if minfitness
%  fprintf('\n%d   Fitness = %f',k, state.fGlobalBest);
mygen(k) = state.Generation;
myfitness(k)= state.fGlobalBest(k);

  stallchk = k - state.LastImprovement >= options.StallGenLimit ;
  if stallchk
      % No improvement for StallGenLimit generations
      exitflag = 3 ;
      flag = 'done' ;
  end
  % ----------------------------------------------------------------------

  % Update flags, state and plots before updating positions
  % ----------------------------------------------------------------------
  if k == 2
      flag = 'iter' ;
  elseif k == itr
      flag = 'done' ;
      exitflag = 0 ;
  end

  if ~isempty(options.PlotFcns) && ~mod(k,options.PlotInterval)
      % Exit gracefully if user has closed the figure
      if isempty(findobj('Tag','Swarm Plots','Type','figure'))
          exitflag = -1 ;
          break
      end % if isempty
      % Find a good size for subplot array
      rows = floor(sqrt(length(options.PlotFcns))) ;
      cols = ceil(length(options.PlotFcns) / rows) ;
      % Cycle through plotting functions
      if strcmpi(flag,'init')
          haxes = zeros(length(options.PlotFcns),1) ;
      end % if strcmpi
      for i = 1:length(options.PlotFcns)
          if strcmpi(flag,'init')
              haxes(i) = subplot(rows,cols,i,...
                  'Parent',state.hfigure) ;
              set(gca,'NextPlot','replacechildren')
          else
              subplot(haxes(i))
          end % if strcmpi
          state = options.PlotFcns{i}(options,state,flag) ;
      end % for i
```

```matlab
        drawnow
    end % if ~isempty

    if ~isempty(options.OutputFcns) && ~mod(k,options.PlotInterval)
        for i = 1:length(options.Output)
            state = options.OutputFcns{i}(options,state,flag) ;
        end % for i
    end % if ~isempty

    if strcmpi(flag,'done')
        break
    end % if strcmpi
    % ---------------------------------------------------------------------

    % Update the particle velocities and positions
    state = psoiterate(state,options) ;
end % for k

% Assign output variables and generate output
% -----------------------------------------------------------------------
xOpt = state.xGlobalBest ;
fval = state.fGlobalBest(k) ; % Best fitness value
% Final population: (hopefully very close to each other)
population = state.Population ;
scores = state.Score ; % Final scores (NOT local bests)
output.generations = k ; % Number of iterations performed
clear state
scatter(mygen, myfitness, 20,'b','d');
output.message = psogenerateoutputmessage(options,output,exitflag) ;
if options.Verbosity > 0, fprintf('\n\n%s\n',output.message) ; end
% -----------------------------------------------------------------------

% Check for hybrid function, run if necessary
% -----------------------------------------------------------------------
if ~isempty(options.HybridFcn) && exitflag ~= -1
    [xOpt,fval] = psorunhybridfcn(fitnessfcn,xOpt,Aineq,bineq,...
        Aeq,beq,LB,UB,nonlcon,options) ;
end
% -----------------------------------------------------------------------

% Wrap up
% -----------------------------------------------------------------------
if options.Verbosity > 0
    if exitflag == -1
        fprintf('\nBest point found: %s\n\n',mat2str(xOpt,5))
    else
        fprintf('\nFinal best point: %s\n\n',mat2str(xOpt,5))
    end
end % if options.Verbosity

if ~nargout, clear all, end
% -----------------------------------------------------------------------
```