

## README

This document describes the package of source code used for directional connectivity computations in “Directional connectivity in hydrology and ecology” by Larsen et al. To run this source code, Matlab is needed, together with the Matlab Image Processing toolbox and the MatlabBGL library. The latter is open source and can be downloaded at <http://dgleich.github.com/matlab-bgl/>.

**Disclaimer:** Unless noted otherwise, the codes below written by the USGS are public domain. See individual third-party library and package descriptions for intellectual property information, user agreements, and related information. Although the codes below have been used by the USGS, no warranty, expressed or implied, is made by the USGS as to the accuracy and functioning of such software and related material nor shall the fact of distribution constitute any such warranty, and no responsibility is assumed by the USGS in connection therewith.

### DCI Code Package:

`connectivity_at_angle.m`: Computes a specified directional connectivity index at a bearing different from that of the image boundary. Calls to functions to create the adjacency or distance matrix or to compute the directional connectivity index of choice are contained within this function.

`im2adjacency_skel.m`: Converts a binary image to undirected adjacency and distance matrices, using the skeleton network convention for defining links and nodes.

`im2adjacency_full.m`: Converts a binary image to undirected adjacency and distance matrices, using the full network convention for defining links and nodes.

`im2adjacency_skel_directed.m`: Converts a binary image to directed adjacency and distance matrices, using the skeleton network convention for defining links and nodes.

`im2adjacency_full_directed.m`: Converts a binary image to directed adjacency and distance matrices, using the full network convention for defining links and nodes.

`DCIu.m`: Calculates the unweighted directional connectivity index (DCI) from a distance matrix.

`DCIw.m`: Calculates the distance-weighted directional connectivity index (DCI) from a distance matrix.

`despur.m`: Called within `connectivity_at_angle.m`, this function corrects for spurs in the skeleton image that arise as artifacts after image rotation.

`indicator_semivariogram.m`: Calculates the indicator semivariogram along the horizontal dimension of the input matrix. The range of the semivariogram is the output, given in number of pixels. To convert it to a dimensional range, it needs to be multiplied by  $dx$ .

topoICS.m: Calculates the directional integral connectivity scale length, given in number of pixels. The direction along which it is computed is the horizontal dimension of the matrix (from left to right). To convert the integral connectivity scale to a dimensional length, it needs to be multiplied by  $dx$ .

```

function fval = connectivity_at_angle(theta, state, dx,dy, res, which_index)

%~~~~~
%connectivity_at_angle.m
%Written and updated June 2012 by Laurel Larsen, lglarsen@usgs.gov
%US Geological Survey, Reston, VA
%~~~~~

% This function rotates (and, if desired, resamples) the input image with
% respect to the computational grid in order to calculate directional
% connectivity at a bearing of interest. The output (fval) is the
% directional connectivity index convention of choice (weighted or
% unweighted) at the bearing theta. This function can be called repeatedly
% over a range of theta values to produce connectivity-orientation curves,
% which are then used to identify the orientation of landscape features.

% Inputs to the function are:

% theta: Bearing (in degrees clockwise from the axis running along
% dimension 1 of the source image) along which directional connectivity is
% to be computed.

% state: a 2D matrix of zeros and ones, in which ones represent the
% landscape patch of interest. The axis of interest along which directional
% connectivity is computed is dimension 1 of this matrix.

% dx: pixel length in cm (i.e., along dimension 1 of the variable "state").
% The way this code is presently set up, dx can only be equal to dy or be
% twice the value of dy.

% dy: pixel width in cm (i.e., along dimension 2 of the variable "state").

% res: Resolution multiplier fraction (either 1, 0.5, or 0.25, to leave the
% image resolution unchanged or coarsen it by a factor of 2 or 4,
% respectively).

% which_index: Specify 'DCIu' or 'DCIw'. The "u" designates an
% unweighted index; the "w" a weighted index as described in the
% documentation for those functions.
%~~~~~

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% %1. Set up

state = state'; %Transpose the input matrix for the computations below

fprintf('%s%d%s', 'Current value of theta is ', theta, '. ')

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% %2. If dx is twice as great as dy, create a new state matrix with twice
% as many pixels and dx equal to dy.

if dx~=dy
    newstate = NaN(size(state,1), size(state,2)*2);
    for jj = 1:size(state,2)
        newstate(:, 2*jj-1) = state(:,jj); newstate(:,2*jj) = state(:,jj);
    end
    state = newstate;
    dx = dy;
else
    newstate = state;
end

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% %3. Coarsen the resolution to enhance computational speed

% First, check to see whether an allowed resolution is specified.
if isempty(intersect(res, [1 0.5 0.25]))

```

```

    error('Crns:resChk', 'Pick a resolution fraction of 1, 0.5, or 0.25')
end

% If using a non-unity resolution fraction, resample the newstate matrix to
% half the number of pixels.
if res < 1
    newstate = NaN(floor(size(state)/2));
    for ii = 1:size(newstate,1)
        for jj = 1:size(newstate,2)
            newstate(ii,jj) = (state(2*ii-1, 2*jj-1)+state(2*ii, 2*jj-1)+state(2*ii-1, 2*jj) + state(2*ii, 2*jj))/4;
        end
    end
    state = newstate;

%If using a resolution fraction of 0.25, resample the state matrix again to
% half the number of pixels.
if res < 0.5
    newstate = NaN(floor(size(state)/2));
    for ii = 1:size(newstate,1)
        for jj = 1:size(newstate,2)
            newstate(ii,jj) = (state(2*ii-1, 2*jj-1)+state(2*ii, 2*jj-1)+state(2*ii-1, 2*jj) + state(2*ii, 2*jj))/4;
        end
    end
    state = newstate;
end

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 4. Rotate by theta degrees
state = imrotate(state, theta);

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 5. Crop images to a square that cuts off all "blank" space caused by
% rotating a rectangular image with respect to a rectangular computational
% grid.
state = state(floor((size(state, 1)-min(size(newstate))/sqrt(2))/2)+1:floor((size(state, 1)-min(size(newstate))/sqrt(2))/2) + floor(min(size(newstate))/sqrt(2)), floor((size(state, 2)-min(size(newstate))/sqrt(2))/2)+1:floor((size(state, 2)-min(size(newstate))/sqrt(2))/2)+floor(min(size(newstate))/sqrt(2))));

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% %6. Skeletonize the image
skel = bwmorph(1-state, 'skel', Inf); %.*imrotate(ones(size(newstate)), theta); %Take
the imrotate out if using crop approach
skel = despur(skel); %Gets rid of spurious vertical spurs due to artifacts of rotation
figure(7), imshow(skel), pause(0.5)

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% %7. Calculate and plot the directional connectivity index at that bearing
switch which_index
case 'DCIu'
    [distance, adjacency, pixelx pixely] = im2adjacency_skel(skel, dx, dy, 1);
    fval = DCIu(distance, dx, pixelx)
    figure(1), hold on, plot(theta, fval, 'md', 'MarkerFaceColor', 'm'), ylabel(
(which_index), xlabel('theta'), pause(0.5)
case 'DCIw'
    [distance, adjacency, pixelx pixely] = imadjacency_hi(skel, dx, dy, 1);
    fval = DCIw(distance, dx, pixelx)
    figure(2), hold on, plot(theta, fval, 'md', 'MarkerFaceColor', 'm'), ylabel(
(which_index), xlabel('theta'), pause(0.5)
end

```

```
function [distance adjacency pixelx pixely] = im2adjacency_skel(state, dx, dy, %  
despurred)
```

```
%~~~~~  
%im2adjacency_hi.m  
%Written and updated June 2012 by Laurel Larsen, lglarsen@usgs.gov  
%US Geological Survey, Reston, VA  
%~~~~~
```

```
%Convert a binary image to an adjacency matrix, defining links and nodes  
%using the skeleton network convention, whereby every "on" pixel (i.e.,  
% with a value of 1) in the skeleton image is treated as a node. The output  
% matrix 'distance' contains the undirected distances between linked  
% nodes. The code is currently set up to compute the distance matrix from  
% physical distances. A future version will contain a modification for  
% readily computing functional distance matrices. 'Adjacency' is a logical  
% matrix with ones on the diagonals and ones indicating that two nodes are  
% linked. "Pixely" is a vector of y-coordinates of the nodes; "pixelx" is a  
% vector of x-coordinates of the nodes. Input variable definitions are as  
% follows:
```

```
% state: a 2D matrix of zeros and ones, in which ones represent either the  
% landscape patch of interest (if there is no fourth argument or the fourth  
% argument is not equal to one) or the image skeleton (if "despurred" = 1).  
% If the base image is rotated first within the function  
% "connectivity_at_angle", use the image skeleton. Otherwise, use the  
% binary image for this variable. The axis of interest along which  
% directional connectivity is computed is dimension 1 of this matrix.
```

```
%dx: pixel length in cm (i.e., along dimension 1 of the variable "state").
```

```
%dy: pixel width in cm (i.e., along dimension 2 of the variable "state").
```

```
%despurred: Do NOT include this as an argument UNLESS the base image has  
%been rotated with respect to the computational grid (to compute  
%connectivity at a particular bearing). The argument is 1 if rotation has  
%occurred, as implemented in connectivity_at_angle.m.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
%1. Skeletonize the image or read in the skeleton image if provided as  
%variable "state".
```

```
if nargin < 4, despurred = NaN; end %Assign NaN to despurred if no argument for that  
variable was given.
```

```
if despurred ~= 1  
    skel = bwmorph(state', 'skel', Inf); %Requires Matlab's image processing toolbox  
else  
    skel = state';  
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
%2. Locate the nodes of the skeleton image and their neighbors
```

```
%a. Find the number of 8-connected neighbors of each pixel.
```

```
padded = zeros(size(skel)+[2 2]);  
padded(2:size(padded,1)-1, 2:size(padded,2)-1) = skel;  
neighbors = cat(3, padded(1:size(padded,1)-2, 1:size(padded,2)-2), padded(1:size(padded,1)-2, 2:size(padded,2)-1), padded(1:size(padded,1)-2, 3:size(padded,2)), padded(2:size(padded,1)-1, 1:size(padded,2)-2), padded(2:size(padded,1)-1, 3:size(padded,2)), padded(3:size(padded,1), 1:size(padded,2)-2), padded(3:size(padded,1), 2:size(padded,2)-1), padded(3:size(padded,1), 3:size(padded,2)));  
n_connected = sum(neighbors,3);
```

```
%b. All skeleton pixels are defined as nodes.  
node = skel;
```

```
%c. Save pixel coordinates of the nodes  
ind = find(node);  
[pixely pixelx] = ind2sub(size(skel), ind);
```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%3. Calculate link distances
dd = sqrt(dx^2+dy^2); %diagonal pixel distance
startlink = [];
endlink = [];
distlink = [];
for ii = 1:length(ind) %Do this for all nodes
    searchfromy = pixely(ii); searchfromx = pixelx(ii); %Coordinates of the pixel being
searched from
    n_traced = 0; %number of links traced so far
    which_neighbors = find(neighbors(searchfromy, searchfromx,:));
    while n_traced < n_connected(searchfromy, searchfromx) %Trace a new path to the next
downwind node unless all paths have already been traced.
        n_traced = n_traced+1;
        dist = 0;
        which_neighbor = which_neighbors(n_traced);
        newy = searchfromy; newx = searchfromx;
        keep_searching = 1;
        while keep_searching
            switch which_neighbor %Figure out distance of this component as the pixel
diagonal, length, or width
                case{1}
                    newy = newy-1; newx = newx-1;
                    dist = dist+dd;
                case{2}
                    newy = newy-1;
                    dist = dist+dy;
                case{3}
                    newy = newy-1; newx = newx+1;
                    dist = dist+dd;
                case{4}
                    newx = newx-1;
                    dist = dist+dx;
                case{5}
                    newx = newx+1;
                    dist = dist+dx;
                case{6}
                    newy = newy+1; newx = newx-1;
                    dist = dist+dd;
                case{7}
                    newy = newy+1;
                    dist = dist+dy;
                case{8}
                    newy = newy+1; newx = newx+1;
                    dist = dist+dd;
            end
            endpoint = find(ind == sub2ind(size(skel), newy, newx));
            keep_searching = 0;
            endlink = [endlink; endpoint]; %Array of link ending index
            startlink = [startlink; ii]; %Array of link starting index
            distlink = [distlink; dist]; %Array of link distances
        end
    end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%4. Output: Store adjacency and distance matrices as memory-saving sparse
%data structures
distance = sparse(startlink, endlink, distlink, ii, ii);
adjacency = sparse([startlink;(1:ii)'], [endlink; (1:ii)'], 1,ii,ii);

```

```

function [distance adjacency pixelx pixely] = im2adjacency_full(state, dx, dy)

%~~~~~
%im2adjacency_full.m
%Written and updated June 2012 by Laurel Larsen, lglarsen@usgs.gov
%US Geological Survey, Reston, VA
%~~~~~

%Convert a binary image to an adjacency matrix, defining links and nodes
%using the "full" convention, whereby a node is any "on" pixel (i.e., with
% a value of 1 in the "state" matrix. The output matrix 'distance'
% contains the undirected distances between linked nodes. The code is
% currently set up to compute the distance matrix from structural
% distances. A future version will contain a modification for readily
% computing functional adjacency matrices. 'Adjacency' is a logical matrix
% with ones on the diagonals and ones indicating that two nodes are linked.
% "Pixely" is a vector of y-coordinates of the nodes; "pixelx" is a vector
% of x-coordinates of the nodes. Input variable definitions are as follows:

% state: a 2D matrix of zeros and ones, in which ones represent the
% landscape patch of interest. The axis of interest along which directional
% connectivity is computed is dimension 1 of this matrix.

% dx: pixel length in cm (i.e., along dimension 1 of the variable "state").

% dy: pixel width in cm (i.e., along dimension 2 of the variable "state").

% ~~~~~
% %1. Assign nodes to "on" state pixels
node = state';

% ~~~~~
%2. Locate the nodes and their neighbors

%a. Find the number of 8-connected neighbors of each node.
padded = zeros(size(node)+[2 2]);
padded(2:size(padded,1)-1, 2:size(padded,2)-1) = node;
neighbors = cat(3, padded(1:size(padded,1)-2, 1:size(padded,2)-2), padded(1:size(padded,1)-2, 3:size(padded,2)), padded(2:size(padded,1)-1, 1:size(padded,2)-2), padded(2:size(padded,1)-1, 3:size(padded,2)), padded(3:size(padded,1), 1:size(padded,2)-2), padded(3:size(padded,1), 2:size(padded,2)-1), padded(3:size(padded,1), 3:size(padded,2)));
n_connected = sum(neighbors,3);

%b. Save pixel coordinates of the nodes
ind = find(node);
[pixely pixelx] = ind2sub(size(node), ind);

% ~~~~~
%3. Calculate link distances
dd = sqrt(dx^2+dy^2); %diagonal pixel distance
startlink = [];
endlink = [];
distlink = [];
for ii = 1:length(ind) %Do this for all nodes
    searchfromy = pixely(ii); searchfromx = pixelx(ii); %Coordinates of the pixel being
searched from
    n_traced = 0; %number of links traced so far
    which_neighbors = find(neighbors(searchfromy, searchfromx,:));
    while n_traced < n_connected(searchfromy, searchfromx) %Trace a new path to the next
downwind node unless all paths have already been traced.
        n_traced = n_traced+1;
        dist = 0;
        which_neighbor = which_neighbors(n_traced);
        newy = searchfromy; newx = searchfromx;
        keep_searching = 1;
        while keep_searching
            switch which_neighbor %Figure out distance of this component as the pixel

```

```

diagonal, length, or width
    case{1}
        newy = newy-1; newx = newx-1;
        dist = dist+dd;
    case{2}
        newy = newy-1;
        dist = dist+dy;
    case{3}
        newy = newy-1; newx = newx+1;
        dist = dist+dd;
    case{4}
        newx = newx-1;
        dist = dist+dx;
    case{5}
        newx = newx+1;
        dist = dist+dx;
    case{6}
        newy = newy+1; newx = newx-1;
        dist = dist+dd;
    case{7}
        newy = newy+1;
        dist = dist+dy;
    case{8}
        newy = newy+1; newx = newx+1;
        dist = dist+dd;
    end
    endpoint = find(ind == sub2ind(size(node), newy, newx));
    keep_searching = 0;
    endlink = [endlink; endpoint]; %Array of link ending index
    startlink = [startlink; ii]; %Array of link starting index
    distlink = [distlink; dist]; %Array of link distances
end
end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%4. Output: Store adjacency and linkage matrices as memory-saving sparse
%data structures
distance = sparse(startlink, endlink, distlink, ii, ii);
adjacency = sparse([startlink;(1:ii)'], [endlink; (1:ii)'], 1,ii,ii);

```



```

function [distance adjacency pixelx pixely] = im2adjacency_skel_directed(state, dx, dy, %
despurred)

%~~~~~
%im2adjacency_skel_directed.m
%Written and updated June 2012 by Laurel Larsen, lglarsen@usgs.gov
%US Geological Survey, Reston, VA
%~~~~~

%Convert a binary image to an adjacency matrix, defining links and nodes
%using the skeleton network convention, whereby every "on" pixel (i.e.,
% with a value of 1) in the skeleton image is treated as a node. The output
% matrix 'distance' contains the directed distances between linked
% nodes. The code is currently set up to compute the distance matrix from
% physical distances. A future version will contain a modification for
% readily computing functional distance matrices. 'Adjacency' is a logical
% matrix with ones on the diagonals and ones indicating that two nodes are
% linked. "Pixely" is a vector of y-coordinates of the nodes; "pixelx" is a
% vector of x-coordinates of the nodes. Input variable definitions are as
% follows:

% state: a 2D matrix of zeros and ones, in which ones represent either the
% landscape patch of interest (if there is no fourth argument or the fourth
% argument is not equal to one) or the image skeleton (if "despurred" = 1).
% If the base image is rotated first within the function
% "connectivity_at_angle", use the image skeleton. Otherwise, use the
% binary image for this variable. The axis of interest along which
% directional connectivity is computed is dimension 1 of this matrix.

%dx: pixel length in cm (i.e., along dimension 1 of the variable "state").

%dy: pixel width in cm (i.e., along dimension 2 of the variable "state").

%despurred: Do NOT include this as an argument UNLESS the base image has
%been rotated with respect to the computational grid (to compute
%connectivity at a particular bearing). The argument is 1 if rotation has
%occurred, as implemented in connectivity_at_angle.m.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%1. Skeletonize the image or read in the skeleton image if provided as
%variable "state".
if nargin < 4, despurred = NaN; end %Assign NaN to despurred if no argument for that %
variable was given.
if despurred ~= 1
    skel = bwmorph(state', 'skel', Inf); %Requires Matlab's image processing toolbox
else
    skel = state';
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%2. Locate the nodes of the skeleton image

%a. Find the number of 8-connected neighbors of each pixel.
padded = zeros(size(skel)+[2 2]);
padded(2:size(padded,1)-1, 2:size(padded,2)-1) = skel;
neighbors = cat(3, zeros(size(skel)), padded(1:size(padded,1)-2, 2:size(padded,2)-1), %
padded(1:size(padded,1)-2, 3:size(padded,2)), zeros(size(skel)), padded(2:size(padded,1)-1, %
-1, 3:size(padded,2)), zeros(size(skel)), padded(3:size(padded,1), 2:size(padded,2)-1), %
padded(3:size(padded,1), 3:size(padded,2)));
n_connected = sum(neighbors,3);

%b. All skeleton pixels are defined as nodes.
node = skel;

%c. Save pixel coordinates of the nodes
ind = find(node);
[pixelx pixely] = ind2sub(size(skel), ind);

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%3. Calculate link distances
dd = sqrt(dx^2+dy^2); %diagonal pixel distance
startlink = [];
endlink = [];
distlink = [];
for ii = 1:length(ind)
    searchfromy = pixely(ii); searchfromx = pixelx(ii); %Coordinates of the pixel being
searched from
    n_traced = 0; %number of links traced so far
    which_neighbors = find(neighbors(searchfromy, searchfromx,:));
    while n_traced < n_connected(searchfromy, searchfromx) %Trace a new path to the next
downwind node unless all paths have already been traced.
        n_traced = n_traced+1;
        dist = 0;
        which_neighbor = which_neighbors(n_traced);
        newy = searchfromy; newx = searchfromx;
        keep_searching = 1;
        while keep_searching
            switch which_neighbor %Figure out distance of this component as the pixel
diagonal, length, or width
                case{2}
                    newy = newy-1;
                    dist = dist+dy;
                case{3}
                    newy = newy-1; newx = newx+1;
                    dist = dist+dd;
                case{5}
                    newx = newx+1;
                    dist = dist+dx;
                case{7}
                    newy = newy+1;
                    dist = dist+dy;
                case{8}
                    newy = newy+1; newx = newx+1;
                    dist = dist+dd;
            end
            endpoint = find(ind == sub2ind(size(skel), newy, newx));
            keep_searching = 0;
            endlink = [endlink; endpoint]; %Array of link ending index
            startlink = [startlink; ii]; %Array of link starting index
            distlink = [distlink; dist]; %Array of link distances
        end
    end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%4. Output: Store adjacency and distance matrices as memory-saving sparse
%data structures
distance = sparse(startlink, endlink, distlink, ii, ii);
adjacency = sparse([startlink;(1:ii)'], [endlink; (1:ii)'], 1,ii,ii);

```

```

function [distance adjacency pixelx pixely] = im2adjacency_full_directed(state, dx, dy)

%~~~~~
%im2adjacency_full_directed.m
%Written and updated June 2012 by Laurel Larsen, lglarsen@usgs.gov
%US Geological Survey, Reston, VA
%~~~~~

%Convert a binary image to an adjacency matrix, defining links and nodes
%using the "full" convention, whereby a node is any "on" pixel (i.e., with
% a value of 1 in the "state" matrix. The output matrix 'distance'
% contains the directed distances between linked nodes. The code is
% currently set up to compute the distance matrix from structural
% distances. A future version will contain a modification for readily
% computing functional adjacency matrices. 'Adjacency' is a logical matrix
% with ones on the diagonals and ones indicating that two nodes are linked.
% "Pixely" is a vector of y-coordinates of the nodes; "pixelx" is a vector
% of x-coordinates of the nodes. Input variable definitions are as follows:

% state: a 2D matrix of zeros and ones, in which ones represent the
% landscape patch of interest. The axis of interest along which directional
% connectivity is computed is dimension 1 of this matrix.

% dx: pixel length in cm (i.e., along dimension 1 of the variable "state").

% dy: pixel width in cm (i.e., along dimension 2 of the variable "state").

% ~~~~~
% %1. Assign nodes to "on" state pixels
node = state';

% ~~~~~
%2. Locate the nodes and their neighbors

%a. Find the number of 8-connected neighbors of each node.
padded = zeros(size(node)+[2 2]);
padded(2:size(padded,1)-1, 2:size(padded,2)-1) = node;
neighbors = cat(3, zeros(size(skel)), padded(1:size(padded,1)-2, 2:size(padded,2)-1),  
padded(1:size(padded,1)-2, 3:size(padded,2)), zeros(size(skel)), padded(2:size(padded,1),  
-1, 3:size(padded,2)), zeros(size(skel)), padded(3:size(padded,1), 2:size(padded,2)-1),  
padded(3:size(padded,1), 3:size(padded,2)));
n_connected = sum(neighbors,3);

%b. Save pixel coordinates of the nodes
ind = find(node);
[pixelx pixely] = ind2sub(size(skel), ind);

% ~~~~~
%3. Calculate link distances
dd = sqrt(dx^2+dy^2); %diagonal pixel distance
startlink = [];
endlink = [];
distlink = [];
for ii = 1:length(ind) %Do this for all nodes
    searchfromy = pixely(ii); searchfromx = pixelx(ii); %Coordinates of the pixel being  
searched from
    n_traced = 0; %number of links traced so far
    which_neighbors = find(neighbors(searchfromy, searchfromx,:));
    while n_traced < n_connected(searchfromy, searchfromx) %Trace a new path to the next  
downwind node unless all paths have already been traced.
        n_traced = n_traced+1;
        dist = 0;
        which_neighbor = which_neighbors(n_traced);
        newy = searchfromy; newx = searchfromx;
        keep_searching = 1;
        while keep_searching
            switch which_neighbor %Figure out distance of this component as the pixel  
diagonal, length, or width

```

```
        case{2}
            newy = newy-1;
            dist = dist+dy;
        case{3}
            newy = newy-1; newx = newx+1;
            dist = dist+dd;
        case{5}
            newx = newx+1;
            dist = dist+dx;
        case{7}
            newy = newy+1;
            dist = dist+dy;
        case{8}
            newy = newy+1; newx = newx+1;
            dist = dist+dd;
    end
    endpoint = find(ind == sub2ind(size(skel), newy, newx));
    keep_searching = 0;
    endlink = [endlink; endpoint]; %Array of link ending index
    startlink = [startlink; ii]; %Array of link starting index
    distlink = [distlink; dist]; %Array of link distances
end
end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%4. Output: Store adjacency and linkage matrices as memory-saving sparse
%data structures
distance = sparse(startlink, endlink, distlink, ii, ii);
adjacency = sparse([startlink;(1:ii)'], [endlink; (1:ii)'], 1,ii,ii);
```

```

function DCI = DCIu(distance, dx, pixelx)

%-----
%DCIu.m
%Written and updated June 2012 by Laurel Larsen, lglarsen@usgs.gov
%US Geological Survey, Reston, VA
%-----

% This function calculates the unweighted directional connectivity index,
% in which connectivity at all scales (down to the length of a pixel)
% contributes equally to the index. Inputs are a distance matrix and
% array of node locations from im2adjacency_skel.m or im2adjacency_full.m,
% depending on whether the "full" or "skel" convention for defining links and
% nodes is employed. The output is the directional connectivity index
% (DCI), scaled between 0 and 1.
%
% This function requires the MatlabBGL library (Gleich, 2011) to be
% installed. This open-source library can be downloaded at
% http://dgleich.github.com/matlab-bgl/
%-----

dx = dx/100; %Converts to meters so that we're not dealing with such large numbers.
distance = distance/100; %Converts to meters so that we're not dealing with such large
numbers.
R = max(pixelx); %The number of rows in the original image
start_nodes = find(pixelx~=R); %Nodes in the last row of the image cannot be
starting/source nodes.
num = 0; %Initialize numerator of summation
den = 0; %Initialize denominator of summation
for ii = 1:length(start_nodes)
    d = shortest_paths(distance, start_nodes(ii)); %A vector of the shortest path
between starting/source node and all other nodes
    r = pixelx(start_nodes(ii)); %The row of the starting/source node
    for jj = r+1:R
        end_nodes = find(pixelx == jj);
        if ~isempty(end_nodes)
            dij = min(d(end_nodes)); %Shortest distance (structural or functional)
between starting/source node and any node in the next row
            num = num+dx*(jj-r)/dij;
            den = den+1;
        end
    end
end
DCI = num/den;

```

```

function DCI = DCIw(distance, dx, pixelx)

%-----
%DCIw.m
%Written and updated June 2012 by Laurel Larsen, lglarsen@usgs.gov
%US Geological Survey, Reston, VA
%-----

% This function calculates the weighted directional connectivity index,
% in which connectivity at large scales is weighted preferentially.
% Weighting is proportional to the projected downwind distance between each
% pair of nodes. Inputs are a distance matrix and array of node locations
% from im2adjacency_skel.m or im2adjacency_full.m, depending on whether the
% "full" or "skel" convention for defining links and nodes is employed. The
% output is the directional connectivity index (DCI), scaled between 0 and
% 1.
%
% This function requires the MatlabBGL library (Gleich, 2011) to be
% installed. This open-source library can be downloaded at
% http://dgleich.github.com/matlab-bgl/
%-----

dx = dx/100; %Converts to meters so that we're not dealing with such large numbers.
distance = distance/100; %Converts to meters so that we're not dealing with such large
numbers.
R = max(pixelx); %The number of rows in the original image
start_nodes = find(pixelx~=R); %Nodes in the last row of the image cannot be
starting/source nodes.
num = 0; %Initialize numerator of summation
den = 0; %Initialize denominator of summation
for ii = 1:length(start_nodes)
    d = shortest_paths(distance, start_nodes(ii)); %A vector of the shortest path
between starting/source node and all other nodes
    r = pixelx(start_nodes(ii)); %The row of the starting/source node
    for jj = r+1:R
        end_nodes = find(pixelx == jj);
        if ~isempty(end_nodes)
            dij = min(d(end_nodes)); %Shortest distance (structural or functional)
between starting/source node and any node in the next row
            num = num+dx^2*(jj-r)^2/dij;
            den = den+dx*(jj-r);
        end
    end
end
DCI = num/den;

```

```

function A = despur(A)

%-----
%despur.m
%Written and updated June 2012 by Laurel Larsen, lglarsen@usgs.gov
%US Geological Survey, Reston, VA
%-----

% When Matlab rotates an image and then creates a skeleton from it,
% vertical spurs appear as an artifact. This function removes those spurs
% from the skeleton image (A), returning the skeleton without the spurs.

nrows = size(A,1); %Number of rows in the skeleton image.
ncols = size(A,2); %Number of columns in the skeleton image.

go_on = 1;

while go_on == 1
    B = [A(nrows,:); A; A(1,:)]; B = [B(:,ncols) B(:,1)]; %Rearrange data matrix to
    perform matrix functions without using a for loop.
    nneighbors = B(1:nrows, 1:ncols)+B(2:nrows+1, 1:ncols)+B(3:nrows+2, 1:ncols)+B(1:
nrows, 2:ncols+1)+B(3:nrows+2, 2:ncols+1)+B(1:nrows, 3:ncols+2)+B(2:nrows+1, 3:ncols+2)+
+B(3:nrows+2, 3:ncols+2); %A matrix containing information about the number of neighbors
each pixel has.
    verticneighbors = B(1:nrows, 2:ncols+1) + B(3:nrows+2, 2:ncols+1); %A matrix
containing information about the number of vertical neighbors each pixel has.
    these_indices = find(nneighbors == 1 & verticneighbors == 1 & A == 1); %Indices
containing spurs.
    if isempty(these_indices)
        go_on = 0;
    else
        A(these_indices) = 0; %Remove the spurs by setting them equal to 0.
    end
end

```

```

function [gamma, range] = indicator_semivariogram(state)

%-----
%indicator_semivariogram.m
%Written and updated June 2012 by Laurel Larsen, lglarsen@usgs.gov
%US Geological Survey, Reston, VA
%-----

%This function calculates indicator semivariograms along the horizontal
%dimension of the matrix. The range is given in number of pixels and should
%be multiplied by dx.
%Flow in the state matrix is assumed to be from left to right.

n_rows = size(state,1); %Number of rows in state
n_col = size(state,2); %Number of columns in state
h = 1:n_col-1; %separation bins
N = zeros(size(h)); %Number of pairs summed in each separation bin
gamma = zeros(size(h));
for c = 1:n_col-1
    gamma(1:n_col-c) = gamma(1:n_col-c)+sum(((state(:, c+1:size(state,2)) - repmat(state(:,c), 1, n_col-c)).*repmat(state(:,c), 1, n_col-c)).^2, 1); %An array of the
semivariogram summation corresponding to column c for h = 1, 2, ...n
    N(1:n_col-c) = N(1:n_col-c) + sum(repmat(state(:,c),1,n_col-c), 1); %Number of
sample pairs in each lag distance bin.
end
gamma = gamma./(2*N); %Indicator semivariogram

figure, plot(h, gamma, 'k-')
xstop = find(diff(gamma)<0, 1, 'first'); %Comment this row or the next to select the
convention for defining which points to use for fitting the theoretical semivariogram.
% xstop = round(input('Maximum value of h for range-fitting is?\n'));
beta = nlinfit(h(1:xstop), gamma(1:xstop), @rangefitter, [0 0.25 3*xstop/4]); %Fit the
theoretical semivariogram
hold on
yhat = beta(1)+beta(2)*(1-exp(-h(1:xstop)./beta(3))); %The theoretical semivariogram
plot(h(1:xstop), yhat, 'k-')
range = beta(3); %Output semivariogram range.

%-----
%rangefitter.m
%Copyright 2012 by Laurel Larsen, lglarsen@usgs.gov
%US Geological Survey, Reston, VA
%-----
function yhat = rangefitter(beta, h)

yhat = beta(1)+beta(2)*(1-exp(-h./beta(3)));

```



```

function [h,p,ICS] = topoICS(state, adjacency)

%~~~~~
%topoICS.m
%Written and updated June 2012 by Laurel Larsen, lglarsen@usgs.gov
%US Geological Survey, Reston, VA
%~~~~~

%Gives the array of separation distances (h) and probabilities that two
%ones in the downslope direction are connected (p).
%Flow in the state matrix is assumed to be from left to right.
%It is assumed that the reachability matrix (RM) input will be a
%directed matrix so that water cannot flow upstream to go downstream.

RM = all_shortest_paths(adjacency, struct('inf', 9999)); %Come up with a "reachability
matrix" (A matrix containing 1 if node pairs are directly or indirectly connected and 0
if they are not connected.)
RM = sparse(RM-9999); %Store the matrix in sparse format
RM(RM<0) = 1;
n_rows = size(state,1); %Number of rows in state
n_col = size(state,2); %Number of columns in state
h = 1:n_col-1; %separation bins
n_ones = zeros(size(h)); %Number of pairs of ones in each downslope separation bin
n_connected = zeros(size(h)); %Number of connected pairs of ones in each downslope
separation bin
state_index = reshape(cumsum(reshape(state, numel(state), 1)), n_rows, n_col); %index of
each node in the RM

for c = 1:n_col-1
    pairs = repmat(state(:,c), 1, n_col-c).*state(:, c+1:n_col); %1 for pairs of 1s
    n_ones(1:n_col-c) = n_ones(1:n_col-c)+sum(pairs, 1); %Number of pairs of ones in
each h bin
    start_i = repmat(state_index(:,c), 1, n_col-c).*pairs; %Gives the index of the
upstream node for each pair. Zeros elsewhere.
    end_i = state_index(:,c+1:n_col).*pairs; %Gives the index of the downstream member
of each pair of nodes
    connected = RM(sub2ind(size(RM), start_i(start_i>0), end_i(end_i>0)));
    connected_mat = start_i; %Initialize matrix of connected pairs
    connected_mat(start_i>0) = connected; %1 where pairs of ones are connected
    n_connected(1:n_col-c) = n_connected(1:n_col-c)+sum(connected_mat, 1); %Number of
connected pairs in each h bin
end
p = n_connected./n_ones; %Probability that a pair of 1s is connected in the downslope
direction for each h bin.
figure, plot(h,p, 'k-d')
ICS = sum(p);

```