Supplemental Materials

Counterbalancing for serial order carryover effects in experimental condition orders.

Psychological Methods

Joseph L Brooks

PROGRAMS FOR GENERATING SEQUENCES

Below is a Matlab/Octave program that can be used to create a serial-order counterbalanced sequence of conditions, i.e. ensuring that every condition is preceded equally often by every other condition (for first-order counterbalancing) or every nTuple of conditions of length n (for nth-order counterbalancing). The program uses the Kandel et al. 1996 method for finding an Euler path through the graph. This algorithm selects uniformly from all available sequences.

Matlab/Octave Function – carryoverCounterbalance.m

This function has been written to function equivalently in Matlab and Octave. Matlab (http://www.matlab.com) Mathworks, Natick, MA, USA) is a commercially-available technical computing environment. Some universities/departments have a site license for Matlab. Check with your IT administrator. Octave (http://www.gnu.org/software/octave/) is a freely-available numerical computing environment similar to Matlab.

This function works by entering the name of the .m file at the command line in Matlab or Octave (see detailed instructions in the *Matlab/Octave Function Code & Usage* section below). It was written in Matlab version 7.10.0.499 (R2010A) but should be compatible with most versions. No toolboxes are required. Make sure that the .m file is in the Matlab path before trying to run it. A counterbalanced condition sequence is output as an array of numbers. For access to the code open the .m file in Matlab or Octave editor. The subfunction *eulerPathKandelMethod_carryoverCounterbalance* (included in the same .m file) computes the Euler circuit through a directed graph given the adjacency matrix for that graph. This subfunction can be extracted and saved as a separate m-file if one wants to generate more complicated designs than those available with the

carryoverCounterbalance function, for instance, if one wants to create sequences in which some conditions occur more often than others.

Further information on the INPUT parameters:

- **numConds:** number of unique conditions. Must be a positive integer.

- **cbOrder:** depth/order of counterbalancing. Must be a positive integer. First-order counterbalancing ensures that every condition is preceded by every other condition equally often. nth-order counterbalancing ensures that every condition is preceded equally often by every n-length sequence (nTuple) of conditions. e.g. 2nd-order counterbalancing would ensure that condition 1 is preceded equally often by each set of 2 conditions, 1 1, 1 2, 2 1, 2 2.

- **reps:** The number of times each condition is preceded by every other condition (first-order) or every nTuple (nth-order). Must be a positive integer. This can be used to increase the total number of trials while maintaining counterbalancing.

- **omitSelfAdjacencies:** Determines whether the same condition can occur directly adjacent to another instance of that same condition. 0 = include condition self-adjacencies (e.g. 2 2 1 2 1 2); 1 = exclude condition self-adjacencies (e.g. 1 2 1 2 1 2)

Executable Program – carryoverCounterbalanceGUI

Download from: http://www.icn.ucl.ac.uk/counterbalance/

This is an executable version of the carryoverCounterbalance.m Matlab/Octave program that can be used without a Matlab license. It also includes a graphical user interface (GUI) that allows easy entry of the parameters. In order to run this executable file you will also need to install the Matlab Compiler Runtime (MCR) that is distributed for free along with this program (http://www.mathworks.co.uk/products/compiler/). One version is provided for Microsoft Windows-based machines (created on Windows XP SP3). Another version is provided for Apple Mac-based computers.

Windows: Run the executable file. This will start the installation of the MCR and extract the carryoverCounterbalanceGUI executable file into the directory where you ran the installation file. After the MCR installation process completes (by clicking through several “next” buttons), you can run the carryoverCounterbalanceGUI executable by double-clicking on it. After the program initializes, the program window will appear.

Mac: Run the .zip file. This will extract the contents to a folder. Run the MCRInstaller.dmg file. This should unpack the MCRInstaller.pkg file. Run this to install the MCR. After installation of the MCR, the carryoverCounterbalance file can be run by double-clicking it.

Enter parameters for the number of conditions, counterbalancing order, number of repetitions into the appropriate boxes, check or uncheck the box for omitting condition self-adjacencies in the sequence. These parameters are identical to the corresponding inputs for the Matlab/Octave function described above and subject to the same constraints. See description of inputs above for details. Details for each input parameter are also displayed when the mouse cursor is hovered over the input box for the given parameter. The sequence will appear in the text box in the lower half of the window. Depending on the number of conditions, orders of counterbalancing > 3 may take significant computation time and the program will not respond until it has completed generating the sequence.

MATLAB/OCTAVE FUNCTION CODE & USAGE

Place the accompanying *carryoverCounterbalance.m* file in a directory that is in the Matlab or Octave path. The *path* is the set of directories where Matlab and Octave look for functions when they are called. You can add a directory to the path by typing (at the Matlab/Octave command line) *addpath(path)* where *path* is the name of the directory in which the *carryoverCounterbalance.m* file is located. For instance, if the file is located in *C:\Documents and Settings\jbrooks\My Documents\octave* then type: *addpath('C:\Documents and Settings\jbrooks\My Documents\octave\');*

The code in *carryoverCounterbalance.m* is annotated with comments on lines starting with the % symbol. If the accompanying .m file is not accessible, the code below can be copied into a blank text/.m file and saved as *carryoverCounterbalance.m*

Execute the file at the command line by typing *carryoverCounterbalance(#,#,#,#)* where # corresponds to the four parameters described in the text above. The output will be a counterbalanced sequence. To store the output sequence in a variable type *seq = carryoverCounterbalance(#,#,#,#)* where seq will be a variable containing the sequence and the # need to be replaced by the parameters described above.

Code begins here:

```
function condSequence =
carryoverCounterbalance(numConds,cbOrder,reps,omitSelfAdjacencies)
%
%When using this function/code in academic research, please cite:
%Brooks, J.L. (2012). Counterbalancing for serial order carryover effects in
%experimental condition orders. Psychological Methods.
%
%Creates a serial-order counterbalanced sequence of conditions, i.e. it ensures
%that every condition is preceded equally often by every other condition (first-order)
%or every nTuple of length n for nth-order counterbalancing. This can be
%done for any number of conditions. Uses Kandel et al. 1996 method to
%determine Euler path (see accessory functions included below).
%
%OUTPUT:
%-condSequence: the counterbalanced sequence. Length will depend on
%parameters. Output as an array of numbers
%
%INPUTS:
%-numConds: number of unique conditions. Must be a positive integer
%
%-cbOrder: depth/order of counterbalancing. Must be a positive integer.
%First-order counterbalancing ensures that every condition is preceded by
%every other condition equally often. nth-order counterbalancing ensures
%that every condition is preceded equally often by every n-length sequence
%(nTuple) of conditions. e.g. 2nd-order counterbalancing would ensure that
%condition 1 is preceded equally often by each set of 2 conditions,
%1 1, 1 2, 2 1, 2 2.
%
%-reps: The number of times each condition is preceded by every other
%condition (first-order) or every nTuple (nth-order). Must be a positive
%integer. This can be used to increase the total number of trials while
%maintaining counterbalancing.
%
%-omitSelfAdjacencies: 0 = include condition self-adjacencies (e.g. 2 2 1 2 1 1 2);
%1 = exclude condition self-adjacencies (e.g. 1 2 1 2 1 2)
%
% VERSION: 1.0.01.03.2012
% by Joseph Brooks, UCL Institute of Cognitive Neuroscience
% brooks.jl@gmail.com

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%CHECK INPUT
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
if reps < 1
    error('ERROR: reps parameter must be > 0');
end

if cbOrder < 1
    error('ERROR: cbOrder parameter must be > 0');
end

if numConds < 1
    error('ERROR: numConds parameter must be > 0');
end

if omitSelfAdjacencies < 0 || omitSelfAdjacencies > 1
    error('ERROR: omitSelfAdjacencies parameter must be 0 or 1');
end;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%CONSTRUCT ADJACENCY MATRIX FOR GRAPH WITH APPROPRIATE TEMPORAL
%%%RELATIONSHIPS
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

if cbOrder == 1
    %%%Construct adjacency matrix and condition list for cbOrder == 1
    %%%Fully-connected graph
    adjacencyMatrix = ones(numConds,numConds)*reps;
    nTuples = [1:numConds];

    if omitSelfAdjacencies
        adjacencyMatrix = adjacencyMatrix - adjacencyMatrix.*eye(size(adjacencyMatrix));
    end;
else
    %%%CONSTRUCT N-TUPLE CORRESPONDING TO EACH NODE WHEN cbOrder > 1
    nTuples = nTuples_brooks(numConds,cbOrder);

    %If indicated in input parameters, remove nodes with self-adjacencies by putting
zeros in
    %corresponding columns and rows of adjacency matrix
    if omitSelfAdjacencies
        nTuples = nTuples_removeSelfAdjacencies_brooks(nTuples);
    end

    %%%Construct adjacency matrix by connecting only those nTuples
    %%%which share the (n-1)-length-beginning/(n-1)-length-end of their sequences
    adjacencyMatrix = zeros(size(nTuples,1),size(nTuples,1));

    for tminus1 = 1:size(adjacencyMatrix,1)
        for t = 1:size(adjacencyMatrix,2)
            if nTuples(tminus1,2:size(nTuples,2)) == nTuples(t,1:size(nTuples,2)-1)
                adjacencyMatrix(tminus1,t) = 1;
            end
        end
    end

    %%%Duplicate edges based on number of reps requested
    adjacencyMatrix = adjacencyMatrix*reps;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%FIND EULER PATH THROUGH GRAPH SPECIFIED BY ADJACENCY MATRIX
```

```
%%%Uses Kandel et al 1996 method for Euler Circuit
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
nodeSequence = eulerPathKandelMethod_carryoverCounterbalance(adjacencyMatrix);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%CONSTRUCT CONDITION SEQUENCE FROM NODE SEQUENCE.
%%%-For first-order counterbalancing node sequence = cond sequence
%%%-For higher-order counterbalancing, overlapping parts of sequence
%%%must be considered and only one additional condition from the
%%%nTuple will be added for all nodes beyond the first
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
if cbOrder == 1
    condSequence = nodeSequence;
else
    condSequence = nTuples(nodeSequence(1),:);

    for i = 2:length(nodeSequence)
        condSequence = [condSequence nTuples(nodeSequence(i),size(nTuples,2))];
    end
end;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%ACCESSORY FUNCTIONS%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [seq] = eulerPathKandelMethod_carryoverCounterbalance(adjacencyMatrix)
% Finds an Euler circuit through a graph, G, and returns sequence of nodes
% associated with that path. Based on method by Kandel, et al. (1996).
% The circuit/path created is uniformly drawn from the set of all possible
% Eulerian circuits of G.
%
% USAGE:
% [seq] = eulerPathKandelMethod(adjacencyMatrix)
% OUTPUT: seq is a ordered sequence of nodes corresponding to the circuit
% through G.
%
% INPUT: adjacency matrix
%        Option1: If entering a single integer, n, then program assumes a
%        fully-connected digraph with n nodes and 1 instance of the edge
%        between each node.
%        Option2: If entering a matrix then the matrix will be
%        treated as the adjacency matrix of the graph, G. Must be an nxn matrix for
%        a graph with n nodes Dimension1 = represents the
%        node sending the directed edge, e.g. G(1,2) is an entry
%        representing a directed edge from node 1 to node 2.
%
%        - adjacency matrix must be consistent with an Eulerian graph,
%        i.e., all nodes have in-degree = out-degree.
%        Semi-Eulerian graphs are not valid for this program.
%        - all entries in the adjacency matrix must be >= 0. Values
%        larger than one indicate duplicated instances of an edge between
%        the nodes.
%
% References:
% Kandel, D., Matias, Y., Unger, R., & Winkler, P. (1996). Shuffling
% biological sequences. Discrete Applied Mathematics, 71(1-3), 171-185:
%
% VERSION: 1.0.01.03.2012
% by Joseph Brooks, UCL Institute of Cognitive Neuroscience
% brooks.jl@gmail.com
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% CHECK INPUT & GIVE ERROR MESSAGES IF NOT APPROPRIATE
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%If input is integer then create adjacency matrix for fully connected
%%%digraph with one instance of each edge
if size(adjacencyMatrix,1) == 1
    adjacencyMatrix = ones(adjacencyMatrix,adjacencyMatrix);
end

%%%Is matrix square?
if size(adjacencyMatrix,1) ~= size(adjacencyMatrix,2)
    error('ERROR: Adjacency matrix is not square');
end;

%%%Is matrix 2D?
if length(size(adjacencyMatrix)) > 2
    error('ERROR: Adjacency matrix should have only 2 dimensions');
end;

%%% Is graph Eulerian? Error if not. Error if semi-Eulerian
for i = 1:size(adjacencyMatrix,1)
    if sum(adjacencyMatrix(:,i)) ~= sum(adjacencyMatrix(i,:))
        error('ERROR: non_Eulerian graph specfied. In-degree must equal out-degree at
every vertex');
    end;
end

%%% Does each vertex have at least one edge?
for i = 1:size(adjacencyMatrix,1)
    if sum(adjacencyMatrix(:,i)) == 0 && sum(adjacencyMatrix(i,:)) == 0
        error('ERROR: Disconnected graph...at least one vertex has no edges.');
    end;
end

%%%Are all values >= 0
if min(min(adjacencyMatrix)) < 0
    error('ERROR: Adjacency matrix contains value < 0');
end;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% START COMPUTATION
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%Uniformly draw arborescence from the set of all possible for the input graph
%%%arb output is adjacency matrix for an arborescence of the input
arb = arborescenceKandel_carryoverCounterbalance(adjacencyMatrix);

%The following matrix indicates which out edges are NOT included in the
%arborescence by subtracting the arborescence from the adjacency matrix.
%These edges need to be randomly permuted in the outEdges list
remainingEdges = adjacencyMatrix - arb;

%%%For each node create a list of outgoing edges and randomly order them
%%%except for any edge that is part of the arborescence. Any outgoing edges
%%%that are in the arborescence should be last in the order
outEdgesOrder = cell(size(adjacencyMatrix,1),1);
```

```
for o = 1:size(adjacencyMatrix,1)
    %%%In the cell corresponding to node n, list all out edges from this
    %%%node including duplicate out edges as muliple instances in the list.
    for i = 1:size(adjacencyMatrix,2)
        for r = 1:remainingEdges(o,i)
            outEdgesOrder{o} = [outEdgesOrder{o} i];
        end
    end

    %%%Randomly shuffle the out edges for this node
    outEdgesOrder{o} = outEdgesOrder{o}(randperm(length(outEdgesOrder{o})));

    %%%Add arborescence edge to end of list if it exists
    if sum(arb(o,:)) > 0
        outEdgesOrder{o} = [outEdgesOrder{o} find(arb(o,:) == 1)];
    end
end

%%%Set first node in sequence to the root of the arboresence (node
%%%toward which all edges in arb are pointing)
seq = find(sum(arb,2) == 0);

%%% Generate sequence of nodes by starting at the root node of the arb
%%% matrix and take the out edge from that node with the lowest number.
%%% Add the target node of this out edge to the sequence and Remove the out
%%% edge from the list for the source node. Repeat treating the target node
%%% as the new source node until all edges have been traversed.
while sum(cellfun('length',outEdgesOrder)) > 0
    seq = [seq outEdgesOrder{seq(end)}(1)];

    if length(outEdgesOrder{seq(end-1)}) > 1
        outEdgesOrder{seq(end-1)} = outEdgesOrder{seq(end-1)}(2:end);
    else
        outEdgesOrder{seq(end-1)} = [];
    end
end




function arb = arborescenceKandel_carryoverCounterbalance(adjacencyMatrix)

%%% FIND UNIFORMLY-DRAWN ARBORESCENCE for the graph specified by the adjacency matrix
%%% in the input. Do this by performing a backward random walk
%%% on the graph...based on Kandel et al. (1996)
%%%
%%% INPUT: Adjacency Matrix of the graph. A square matrix of values >= 0
%%% where a positive integer in Aij indicates an edge from vertex i to
%%% vertex j. The size of each dimension of the matrix is equal to the
%%% number of vertices of the graph
%%%
%%% OUTPUT: Adjacency Matrix of the arborescence.
%%%
%%% KANDEL ALGORITHM
%%% (1) start at random node
%%% (2) randomly choose an edge to cross (loops which connect back to the same node can
be ignored)
%%% (3) if this edge leads to a node that has not been visited then
%%% this edge is part of the arborescence. For the edge's origin node,
%%% it should be marked as the last edge to be traversed out of the
%%% node when creating the sequence.
%%% in outEdgeOrder matrix to make it the last traversed, i.e. size-of-matrix+1
```

```
%%% (4) select an edge out and move to another node.
%%% (5) repeated 2-4 until each node has been visited at least once.
%%%
%%% VERSION: 1.0.01.03.2012
%%% by Joseph Brooks, UCL Institute of Cognitive Neuroscience
%%% brooks.jl@gmail.com

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%CHECK INPUT%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%Is matrix square?
if size(adjacencyMatrix,1) ~= size(adjacencyMatrix,2)
    error('ERROR: Adjacency matrix is not square');
end;

%%%Is matrix 2D?
if length(size(adjacencyMatrix)) > 2
    error('ERROR: Adjacency matrix should have only 2 dimensions');
end;

%%% Does each vertex have at least one edge?
for i = 1:size(adjacencyMatrix,1)
    if sum(adjacencyMatrix(:,i)) == 0 && sum(adjacencyMatrix(i,:)) == 0
        error('ERROR: At least one vertex is disconnected (i.e. has no edges)');
    end;
end

%%%Are all values >= 0
if min(min(adjacencyMatrix)) < 0
    error('ERROR: Adjacency matrix contains value < 0');
end;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%COMPUTE ARBORESCENCE%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%Create empty adjacency matrix to represent output arborescence
arb = zeros(size(adjacencyMatrix));

%%%Simplify input adjacency matrix by removing duplicate edges. The result
%%%will treat duplicate edges as equivalent.
adjacencyMatrix = sign(adjacencyMatrix);

%%%Choose a random starting vertex and add it to the list of nodes visited
currentNode = randi(size(adjacencyMatrix,1));
nodesVisited = [currentNode];

while length(unique(nodesVisited)) < size(adjacencyMatrix,1)
    %%%Find all edges INTO the current node...edges into the node are
    %%%considered because this is a BACKWARD walk
    availableSourceNodes = find(adjacencyMatrix(:,currentNode) > 0);

    %%%Randomly choose one of the edges into of the node and designate as
    %%%source
    selectedSource = availableSourceNodes(randi(length(availableSourceNodes)));

    %%%If this is the first visit to the source node, then mark the edge as
    %%%part of the arborescence
    if sum([nodesVisited == selectedSource]) == 0
        arb(selectedSource,currentNode) = 1;
```

```
    end

    %%%Add target node to list of visited nodes
    nodesVisited = [nodesVisited,selectedSource];

    currentNode = selectedSource;
end



function result = nTuples_brooks(numItems,n)
%
% Create all possible nTuples of length n from a list of items of length =
% numItems
%
% OUTPUT:
% -result: matrix where each row corresponds to an nTuple of length n. Size
% of matrix will be numItems^n x n
%
% INPUT:
% -numItems: this is the number of items that are possible in each position
% of each nTuple.
% -n: this is the length of each nTuple.
%
% EXAMPLE: For all of the nTuples of length 2 of 3 items, use nTuples(3,2).
% The result of this computation is:
%      1     1
%      1     2
%      1     3
%      2     1
%      2     2
%      2     3
%      3     1
%      3     2
%      3     3
%
% VERSION: 1.0.05.03.2012
% by Joseph Brooks, UCL Institute of Cognitive Neuroscience
% brooks.jl@gmail.com

result = zeros(numItems^n,n);

for v = 1:numItems^n
    for i = 1:n
        result(v,i) = mod(ceil(v/numItems^(i-1)),numItems)+1;
    end
end



function result = nTuples_removeSelfAdjacencies_brooks(nTuplesList)
%
% VERSION: 1.0.05.03.2012
% by Joseph Brooks, UCL Institute of Cognitive Neuroscience
% brooks.jl@gmail.com

result = [];

for i = 1:size(nTuplesList,1)
    containsAdjacency = false;

    for n = 1:size(nTuplesList,2)-1
        if nTuplesList(i,n) == nTuplesList(i,n+1)
```

```
            containsAdjacency = true;
        end;
    end;

    if ~containsAdjacency
        result(end+1,:) = nTuplesList(i,:);
    end
end
```

Table S1

4th-order counterbalanced sequence is simultaneously counterbalanced for 3rd-order effects

Trial Number	4th-Order Counterbalanced Sequence	For each condition (T = 1 or 2) at Trial T, preceding 3-tuples of conditions (trials T-1,T-2,T-3, respectively)															
		Trial T = 1								Trial T = 2							
		112	121	111	122	212	221	211	222	112	121	111	122	212	221	211	222
1	2																
2	1																
3	2																
4	2																
5	2														1		
6	2																1
7	2																1
8	1								1								
9	2												1				
10	1					1											
11	1		1														
12	2									1							
13	1							1									
14	1		1														
15	1	1															
16	2											1					
17	2															1	
18	2														1		
19	1								1								
20	1				1												
21	1	1															
22	1			1													
23	1			1													
24	2											1					
25	1							1									
26	2										1						
27	1					1											
28	2										1						
29	2													1			
30	1						1										
31	1				1												
32	2									1							
33	2															1	
34	1						1										
35	2												1				
36	2													1			
	Total Appearances of each 2-tuple	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**

Note: A 4rd-order counterbalanced sequence (second column) is one in which each condition is preceded equally often by every 4-tuple of conditions. This table shows that in this 4th-order counterbalanced sequence (for 2 conditions) each condition (1 or 2) is preceded equally often by all of the 3-tuples and thus is also 3rd-order counterbalanced. The first n=4 trials must be removed from analysis for this to be the case where n is the highest degree of counterbalancing in the sequence.

Table S2

4th-order counterbalanced sequence is simultaneously counterbalanced for 2nd-order effects

Trial Number	4th-Order Counterbalanced Sequence	For each condition (T = 1 or 2) at Trial T, preceding 2-tuples of conditions (trials T-1,T-2 respectively)							
		Trial T = 1				Trial T = 2			
		11	12	21	22	11	12	21	22
1	2								
2	1								
3	2								
4	2								
5	2								1
6	2								1
7	2								1
8	1				1				
9	2						1		
10	1			1					
11	1		1						
12	2					1			
13	1			1					
14	1		1						
15	1	1							
16	2					1			
17	2							1	
18	2								1
19	1				1				
20	1		1						
21	1	1							
22	1	1							
23	1	1							
24	2					1			
25	1			1					
26	2						1		
27	1			1					
28	2						1		
29	2							1	
30	1				1				
31	1		1						
32	2					1			
33	2							1	
34	1				1				
35	2						1		
36	2							1	
	Total Appearances of each 2-tuple	**4**	**4**	**4**	**4**	**4**	**4**	**4**	**4**

Note: A 4rd-order counterbalanced sequence (second column) is one in which each condition is preceded equally often by every 4-tuple of conditions. This table shows that in this 4th-order counterbalanced sequence (for 2 conditions) each condition (1 or 2) is preceded equally often by all of the 2-tuples and thus is also 2nd-order counterbalanced. The first n=4 trials must be removed from analysis for this to be the case where n is the highest degree of counterbalancing in the sequence.

Table S3

4th-order counterbalanced sequence is simultaneously counterbalanced for 1st-order effects

Trial Number	4th-Order Counterbalanced Sequence	For each condition (T = 1 or 2) at Trial T, preceding conditions (trials T-1)			
		Trial T = 1		Trial T = 2	
		1	2	1	2
1	2				
2	1				
3	2				
4	2				
5	2				1
6	2				1
7	2				1
8	1		1		
9	2			1	
10	1		1		
11	1	1			
12	2			1	
13	1		1		
14	1	1			
15	1	1			
16	2			1	
17	2				1
18	2				1
19	1		1		
20	1	1			
21	1	1			
22	1	1			
23	1	1			
24	2			1	
25	1		1		
26	2			1	
27	1		1		
28	2			1	
29	2				1
30	1		1		
31	1	1			
32	2			1	
33	2				1
34	1		1		
35	2			1	
36	2				1
	Total Appearances of each 2-tuple	**8**	**8**	**8**	**8**

Note: A 4rd-order counterbalanced sequence (second column) is one in which each condition is preceded equally often by every 4-tuple of conditions. This table shows that in this 4th-order counterbalanced sequence (for 2 conditions) each condition (1 or 2) is preceded equally often by every condition (including the same condition) and thus is also 1st-order counterbalanced. The first n=4 trials must be removed from analysis for this to be the case where n is the highest degree of counterbalancing in the sequence.