Title

Software for continuous game experiments

Journal

Experimental Economics, 17(4)

ISSN

1386-4157

Authors

Pettit, James
Friedman, Daniel
Kephart, Curtis
et al.

Publication Date

2014-12-01

DOI

10.1007/s10683-013-9387-3

Peer reviewed

Exp Econ
DOI 10.1007/s10683-013-9387-3

ORIGINAL PAPER

Software for continuous game experiments

James Pettit · Daniel Friedman · Curtis Kephart · Ryan Oprea

Received: 11 December 2012 / Accepted: 23 December 2013

Abstract ConG is software for conducting economic experiments in continuous and discrete time. It allows experimenters with limited programming experience to create a variety of strategic environments featuring rich visual feedback in continuous time and over continuous action spaces, as well as in discrete time or over discrete action spaces. Simple, easily edited input files give the experimenter considerable flexibility in specifying the strategic environment and visual feedback. Source code is modular and allows researchers with programming skills to create novel strategic environments and displays.

Keywords Experimental economics · Continuous time · Software for laboratory experiments

JEL Classification C70 · C88 · C90

1 Introduction

How do people behave when they interact strategically? Social scientists have, for more than half a century, turned to game theory to answer such questions, but predic-

J. Pettit · D. Friedman · C. Kephart (✉)
Economics Department, University of California Santa Cruz, Santa Cruz, USA
e-mail: curtiskephart@gmail.com

J. Pettit
e-mail: James.L.Pettit@gmail.com

D. Friedman
e-mail: dan@ucsc.edu

R. Oprea
Department of Economics, University of California Santa Barbara, Santa Barbara, USA
e-mail: roprea@gmail.com

Published online: 08 January 2014

tions offered by game theory are often ambiguous (e.g., multiple equilibria), vague (e.g., contingent on details of timing that are hard to interpret), or implausible. Not surprisingly, many game theorists have turned to laboratory experiments to test predictions and refine the theory (e.g., Thrall et al. 1954; Rapoport and Orwant 1962; Kagel and Roth 1997; Goeree and Holt 2001).

So far, however, laboratory environments have been rather restrictive. Most are either one-shot encounters between two players, or very simple sequential play, or repetition of some stage game in discrete time. Most offer each player a very limited range of feedback on what other players are doing. Outside the lab, strategic interaction occurs in a much wider range of environments.

In this paper we introduce software that expands the range of environments for conducting experiments inspired by game theory. ConG (short for Continuous Games) is a suite of programs for running experiments with human subjects who interact strategically in real time. It uses graphics intensively to create a variety of visual environments, and allows subjects to continuously change and adapt their strategies. ConG permits several different ways to specify payoff functions, and to display a great deal of information in a compact manner. ConG can also run games in discrete time in a manner that closely parallels and can be easily compared to continuous time.

There are two major reasons to study strategic behavior in continuous time. First, although grid-like discrete time structures are appropriate in many strategic settings, a great deal of human interaction actually unfolds in asynchronous, continuous time environments. For example, consider work on the factory floor, telephone fund-raising drives, most team sports, academic seminars and collaborations, and internet pricing. Although discrete time is often treated in both theory and experiments as a reasonable approximation of continuous time, this is far from settled theoretically or behaviorally. For example, Simon and Stinchcombe (1989) point out that in many settings, asynchronous continuous interaction can fundamentally alter the character of strategic interaction, generating equilibria that are different from discrete time approximations. Friedman and Oprea (2012), using ConG, show that cooperation rates in prisoner's dilemmas rise from less than 50 % to over 90 % as interactions become continuous.

A second reason to study interaction in continuous time is that it allows researchers to speed up adjustment dynamics; continuous time allows researchers to approach long term interaction in relatively short blocks of clock time. For example, Oprea et al. (2012) are able to test some distinctive, long-term implications of evolutionary game theory in a series 20 periods using ConG, each period lasting 120 seconds.

There already exists excellent general purpose software for one-shot and discrete time game experiments, and also for continuous double auction markets, e.g., z-Tree (Fischbacher 2007), and MarketLink (Cox and Swarthout 2006). But thus far, studies of near continuous time have relied on ad hoc software designed to address a specific research question. ConG is more general purpose, and includes several distinctive features:

- ConG is truly asynchronous and event driven—it does not simply run many discrete periods quickly—and to subjects it feels like real-time interaction. (Of course, the software also includes a discrete time option.)

- ConG's configuration files enable experimenters to quickly implement a multitude of game environments. These files allow experimenters to specify the payoff function, display options, lags, etc., period by period, as explained below.
- ConG emphasizes real-time 2D and even 3D graphical displays to provide feedback that subjects can respond to quickly and effortlessly.
- In trials on low-cost machines with 12 automated subjects each changing action 40-60 times a second, the time lag until the results of an action were displayed on all machines rarely exceeded 200 milliseconds and was typically less than 100 milliseconds, faster than human reaction time.[1]
- ConG is free and licensed under a port of the FreeBSD license, subject to a citation requirement when the software is used for academic publications, allowing it to benefit from the advantages of producing software as a public good, Schwarz and Takhteyev (2010). Otherwise there is no restriction on access to, distribution of, or the ability to change ConG or its source code.

Section 2 highlights the key features of ConG and briefly explores what types of economic experiments are easily conducted using existing software as is, simply by editing the configuration file. Section 3 delves more deeply into ConG's underlying software architecture and development, including how additional programming may extend the software to novel experiment designs. The final subsection summarizes the software's limitations. Section 4 offers brief concluding remarks.

2 Basic functionality

Experimenters specify the period-by-period environment of a ConG session via a Comma Separated Variables (.csv) configuration file, easily editable with any spreadsheet software. This file defines the number of periods, period lengths, whether periods are to be run in discrete or continuous time, the payoff function and its parameterization, and variables specifying what subjects will see and how they interact in the game. Full details and documentation can be found at the software's home site: http://leeps.ucsc.edu/cong/.

Figure 1 shows a ConG screen for a symmetric 2×2 matrix game played in discrete time. The configuration file pictured in Fig. 2 shows that the experimenters have chosen prisoner's dilemma payoffs this period, and that the 60 second period is divided into 6 subperiods. For subsequent periods, additional lines of the configuration files could specify different period lengths, unpaid (practice) periods, different numbers of subperiods, or different payoff functions.

Figure 3 shows the user interface for the same game played in continuous time. (Figure 4 shows the configuration file except that generates this display.) Again, players may change their action at any time and as often as desired, but now any change in the action profile is immediately reflected in users' displays and in flow

[1] We ran the same exercise with z-Tree, and lags were an order of magnitude longer. Although an expert z-Tree programmer might be able to shorten these lags somewhat, ConG puts particular focus on rapid action experiments by providing built in low-latency interaction and highly graphical displays that allow for very fast absorption of information.

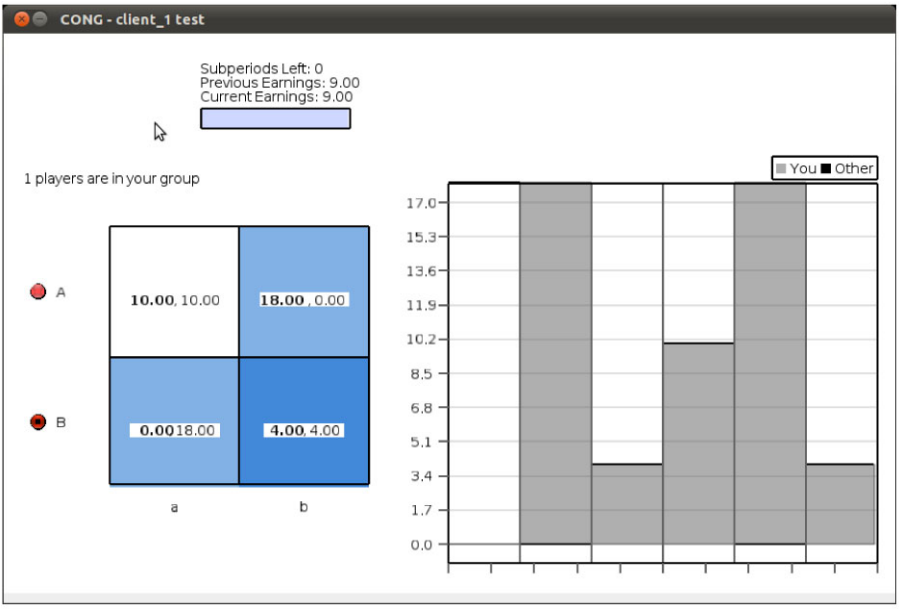

Fig. 1 User interface for a simultaneous-move discrete-time game with ConG software. Once the period starts, subjects may freely (and secretly) choose their current action (here "*B*") by clicking on one of the two radio buttons on the far left of the window, or by using an arrow key, and the corresponding matrix row is *shaded*. The matrix column corresponding to the counterpart player's action choice last period (here "*b*") is also *shaded*, so the cell with corresponding payoffs is *doubly shaded*. The time series chart to the right shows payoffs in the subperiods completed so far in the current period. These payoffs are determined only by the action profile obtaining at the end of each subperiod and actions in the current subperiod are shrouded until the end of the subperiod. To indicate how much time remains in each subperiod, a progress bar (the rectangle just below previous and current earnings table) fills from *white/empty* to *blue/full*. The progress bar is *fully blue* here, indicating the subperiod is over

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	period	paid	length	subperiods	percentChangePerSecond	groupSize	payoffFunction	name	min	max	Aa	Ab	Ba	Bb	selector
2	1	TRUE	60	6	NaN	1	matrix2x2	first	0	17	10	0	18	4	pure

Fig. 2 Configuration file corresponding to Fig. 1. Line 2 specifies the first period (column *A*) of a potentially multiple period session. Column *B* specifies that the period is paid, as opposed to an unpaid practice period. Columns *C* and *D* specify that the period lasts for 60 seconds and is broken down into 6 subperiods (each thus lasting 10 seconds). Column *E* is not applicable here, *F* specifies pairwise matching of players, and *G* invokes a payoff function in the form of a symmetric 2×2 matrix whose entries appear in columns *K*–*N*. Columns *I*–*J* define the vertical scale of the payoff chart, *H* allows for the experimenter to give period treatments different names, (allowing for ease of data analysis), and the "pure" in column *O* specifies the radio buttons subject use to choose their actions

payoffs. The computer response time between hardware interactions and display updates on equipment that satisfies minimum hardware requirements is typically less than 50 milliseconds, faster than human reaction time (Lipps et al. 2011). Based on our experience and the experience of subjects in our lab, this gives the sensation of continuous time, where changes in own and counterpart actions are realized instantly.

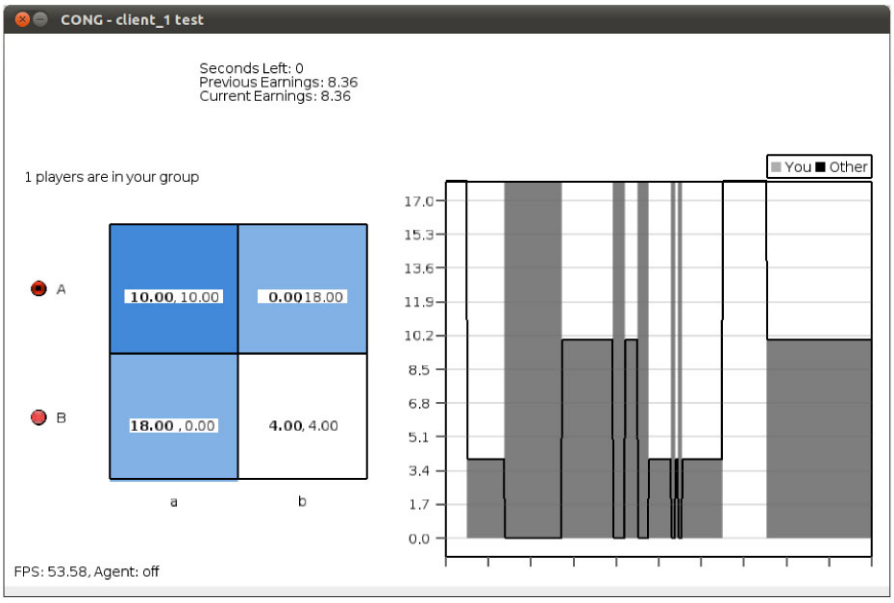

Fig. 3 User interface for a continuous-time game with ConG software. Action selection and shading are as in Fig. 1 (here the player is earning a payoff of 18 while the counterpart earns 0), but the payoffs here are continuous flows that immediately reflect changes in the action profile. The right-side time series chart shows the flow payoffs so far this period for both players. At the end of the period they are paid the integral of these flows, shown as the *gray area* for the player and as the *area* under the *black line* for the player's counterpart

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	period	paid	length	subperiods	percentChangePerSecond	groupSize	payoffFunction	min	max	Aa	Ab	Ba	Bb	selector	name
2	1	TRUE	60	0	NaN	1	matrix2x2	0	17	10	0	18	4	pure	second

Fig. 4 The configuration file corresponding to Fig. 3. Column *D* is set to 0 so this period is continuous time, and column *O* specifies a different name for output files

Figure 5 shows a player's screen for a continuous time population game with a "heatmap" display. (Figure 6 shows the configuration file except that generates this display.) Clicking on the heatmap allows players to choose from a continuous action set, here consisting of all mixes of two pure actions in a symmetric 2×2 payoff matrix.[2] The numbers at the corners of the heatmap show the matrix entries (in this example they form a hawk-dove game. The heatmap displays all potential flow payoffs, with warmer colors indicating higher values. Although the same information is conveyed in two other ways (hovering the cursor over an action profile pops up a text bubble reporting the numerical payoff at that profile, and a numerate player can linearly interpolate payoffs from the numbers shown at the heatmap's corners),

[2]In this example subjects are playing a "population game" (each subject is matched with the average choice of some subset of other players) though a quick change in the configuration file would change this to standard pair-wise matching.

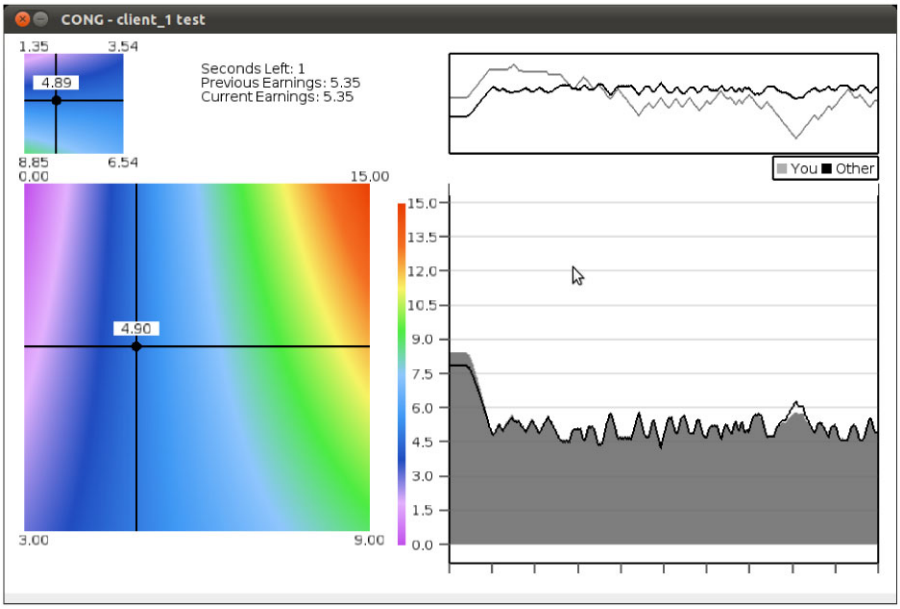

Fig. 5 Heatmap selector in a continuous time population game with a continuous action space. Players click (or use arrow keys) along the *vertical axis* of the large heatmap to select the desired action, which is displayed as a *horizontal black line* at the chosen level; the average of counterparts' current actions is indicated by the *vertical black line*. The intersection of these lines determines the current flow payoff for the player (currently 4.90). The thermometer scale to the right of the large heatmap translates the colors to numerical flow payoffs. Realized flow payoffs are graphed in the large time chart on the right. The small time chart above it shows player action choices over time while the small heatmap to the top left shows counterparts' heatmap, including their current flow payoffs

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
1	period	paid	length	subperiods	numGroups	matchType	percentChangePerSecond	payoffFunction	min	max	Aa	Ab	Ba	Bb	selector	mixed	matrixDisplay	showHeatmapLegend	name
2	1	TRUE	60	0	1	pair	1	matrix2x2	0	15	0	15	3	9	heatmap2d	TRUE	HeatmapBoth	TRUE	hd

Fig. 6 The configuration file corresponding to Fig. 5. Columns *E* and *F* specify a single population game in which each player is matched with the average choice of all other players. Column *G* sets the "speed limit" for action adjustment: with the current setting it will take 1 second to traverse the strategy spectrum, e.g, from the top of the heatmap to the bottom (a setting of 0 makes adjustment instant). Columns *K*–*N* specify the payoff matrix entries (here of the hawk-dove type), column *P* enables continuous strategy adjustment (if the entry were FALSE then subjects could select only corner actions). Columns *O*, *Q* and *R* respectively call for the display of the player's own heatmap, the counterpart heatmap, and the thermometer scale

the heatmap offers a very accessible overview of current and all counterfactual payoffs. Even colorblind subjects have reported that the heatmap (in conjunction with the text bubbles) is quite helpful. The configuration file permits the experimenter to suppress the counterpart's heatmap and/or the thermometer scale and/or the heatmap itself.

Figure 7 shows another interface option: the bubbles selector. (Figure 8 shows the configuration file except producing the bubbles display.) Subjects select their actions by clicking to move the black rectangle slider along the x-axis. The vertical height

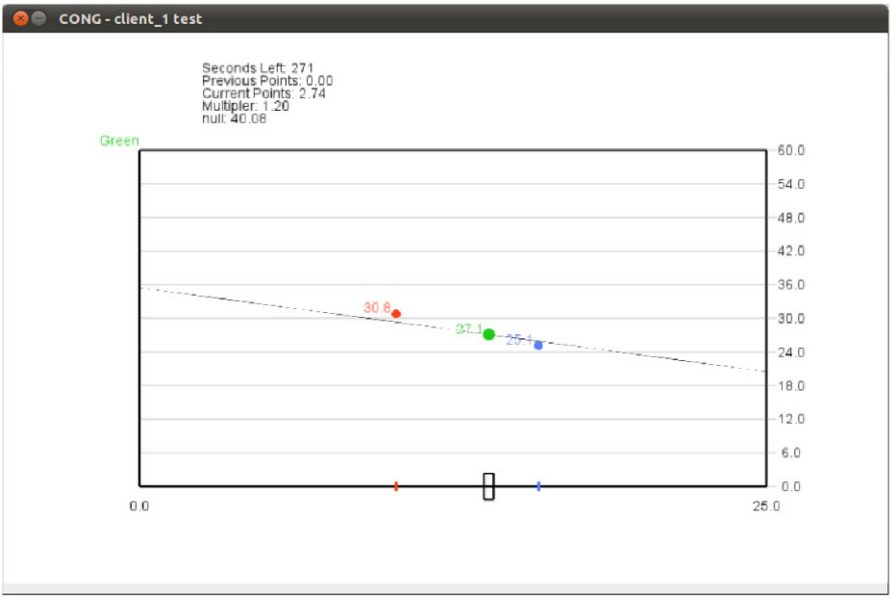

Fig. 7 Bubbles selector in a continuous times experiment. Subjects select their action via the black rectangle slider along the *bottom horizontal axis*. The *vertical height* of player's "bubbles" indicate current flow payoffs, with the "Current Earnings" field accumulating flow payoffs as the period advances. Other bubbles in the display indicate current actions (i.e. current horizontal positions) and payoffs (*vertical heights*) of counterpart players. The thin black "payoff landscape" line shows the payoff the player would earn at all possible actions choices, given counterparts' current profile

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
1	period	paid	length	subperiods	groupSize	matchType	payoffFunction	type	name	A	smin	smax	min	max	mixed	selector	initial	potential
2	6	TRUE	300	0	3	self	sum	public_goods	pf	1.2	0	25	0	60	TRUE	bubbles	0	TRUE

Fig. 8 The configuration file corresponding to Fig. 7. It specifies a 3 player public goods game with multiplier 1.2 during a 300 second paid period in continuous time, using the bubble selector. A FALSE entry in column *R* would suppress the payoff landscape line

of the player's "bubble" corresponds to his or her flow payoff. This selector is a very simple way to allow subjects to pick from a continuous action space. The figure shows a public goods game, but the bubbles selector can be used for any continuous action game (i.e. Cournot games, Bertrand games etc.). We need to cite the working paper, see citation info here: http://ideas.repec.org/p/cdl/ucsbec/qt5404914p.html.

A number of additional features may be implemented via edits to ConG's.csv configuration file. Players may be grouped and matched period-by-period in intricate ways, creating one-population games, two-population games and any other arbitrary counterpart matching or block randomization. Players may be permitted to chat with one another via a freeform chat window. When a player clicks to a new action target in a continuous time game, a configuration file setting determines whether that action is realized instantaneously or the adjustment occurs at a specified finite rate (the "speed limit"). Customization is also possible for features beyond those specified in our standard configuration file. Different sorts of graphic displays and differ-

ent sorts of payoff functions, for example, can be readily programmed in Java using ConG's extensibility feature. These points are discussed below and more extensively on ConG's documentation website.

3 ConG development and architecture

Over the past two decades, LEEPS lab at UC Santa Cruz has developed software libraries that ease the creation of new programs for conducting laboratory experiments. Over the years as programs designed for specific experiments were put together on an ad-hoc basis, it became apparent that a number of utilities are useful to all experiments programmed and implemented in the lab. These include utilities for general real-time networked client-server applications, utilities that enable networked experiments with "soft" real-time latency (in practice typically less than 50 milliseconds round-trip), and libraries that offer access to high quality graphics using the OpenGL and Processing libraries. OpenGL enables 2D and 3D graphics rendering using a graphics co-processor and thus smooth animation. This collection was quite flexible and accessible to programmers. For example, in Winter 2008, two undergraduate students at UC Santa Cruz with heavy course loads elsewhere took less than four weeks to design, code, and run a continuous time game for a game theory class project.

Funded by NSF grant SES-0925039 ("Continuous Games in the Laboratory," 2009–2012), LEEPS lab has used these utilities to construct ConG, a suite of programs that offers a general framework by which a control machine is networked to client machines for the purposes of an economic experiment. Using this suite, an experiment is created by defining the graphic display and characterizing the game to be played together with treatment specific details.

Figure 9 visualizes the process by which the client and server control programs continuously update each other and respond to user input. At initiation of ConG, the software asks for the number of subjects in the session, and allows the experimenter to load a.csv configuration file (discussed in more detail below). This file establishes the payoff function, action selection interface, how subjects are grouped and matched, how each period is timed, and other details that fully characterize the experiment to be run.

Once the server connects to each client, subjects are asked for their name—information linked to the subject's final payoff account—and client windows create the display environment chosen for the first period.

When the experimenter clicks to start period one, the server instructs each client window to unlock the selector, freeing subjects to edit their current action profile. The display begins charting current and past actions, payoffs and anything else selected in the configuration file. The period clock counts down: in discrete time periods, time is designated by a "Subperiods Left" counter; in continuous time periods the clock counts the number of seconds left.

The user interface allows players to keep or change their current action choices, and provides useful information about previous, current, and potential action choices and payoffs. Each combination of configuration file "selector", "payoffFunction" and other fields will produce a different display environment and interface for experiment subjects.

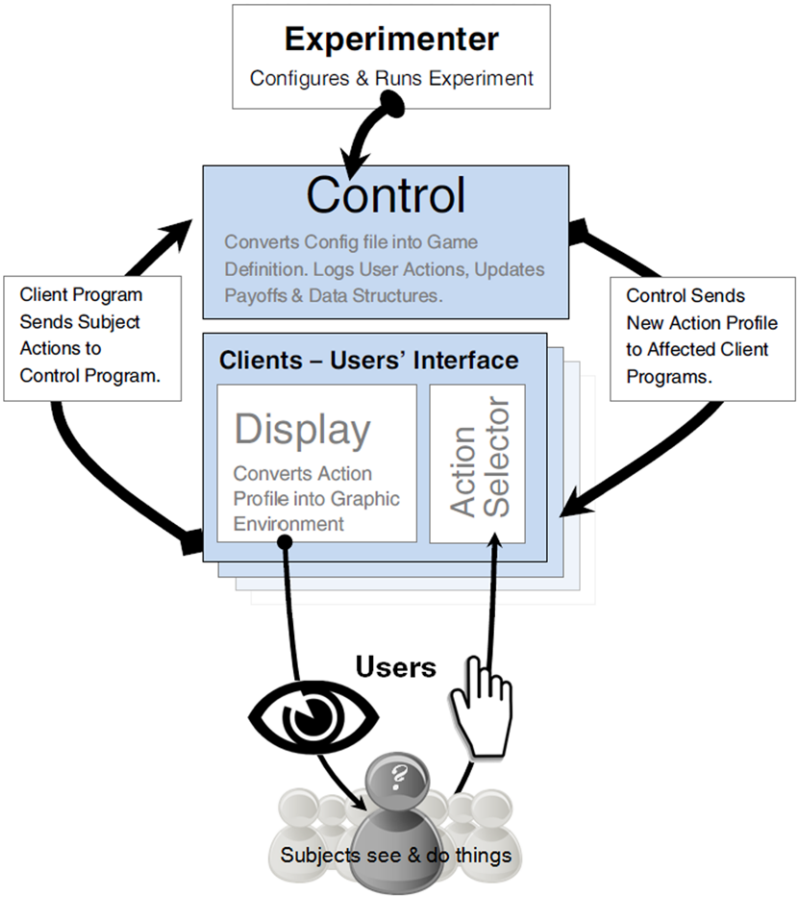

Fig. 9 An abstract visual representation of a ConG experiment

After the period has begun, whenever a subject clicks on the selector to make a change, a monitor process (a thread running alongside the client, watching for any subject activity) notifies the server that the action vector has been modified. The server picks up this notification, queues it, logs it, calculates the duration of the previous action profile, multiplies this by each relevant player's flow payoffs for that profile, and adds it to each player's accumulated payoff accounts. (The relevant players are determined by the grouping and matching procedure. For instance, in an 8 subject session of a public goods game with 4 players per group, only half will be affected by any individual player's change of action.) The server then sends subjects in the affected group the new action vector, which the client programs incorporate into their graphic displays.

This loop is continuously repeated throughout the period's duration: the client threads send subject action changes to the server and, after a number of calculations and changes to data structures, client programs update users' screens. Given minimal hardware specifications, this loop is timed at about 50 milliseconds (0.05 seconds).

That is, it takes about 50 milliseconds from the moment a user clicks a new action until all users see the impact on-screen. Since even the fastest human reaction time is significantly longer than this, it seems to subjects interacting via ConG that the computer accepts and incorporates all clicks instantaneously.

3.1 Graphic display

ConG leverages the Processing library (processing.org) to build graphical widgets for displaying data and selecting from possible actions in the game. Each widget is given a portion of the screen that it may draw to. All widgets have access to the current game state, including the following: (1) the points earned by the subject in the current period, updated at least once per second, (2) the total points earned by the subject in all periods, (3) a time-indexed list of the actions chosen by all subjects in the group, and (4) the current payoff function which is a Java function that takes the current actions played and returns a floating point number as a flow payoff.

Widgets also have access to mouse and keyboard input, and can use these to send action updates to the server. Currently, all widgets draw in 2D, but the library translates this to a plane in 3D, that is rendered using the 3D co-processor when present. This offloads most of the rendering to hardware, and gives frame-rates of around 30 frames per second even on low-end hardware (for example, the LEEPS lab currently uses $300 machines with a built-in 3D processor). On mid-range hardware, 60+ fps is common and easily achievable. In addition, extension to displaying 3D models of data is straightforward, requiring only a new widget that draws in 3D.

Here is an example of game-state action-profile data ConG uses to draw users' graphic display.

```
{{time:  t1, action{1: [0.5, 0.5], 2: [0.2, 0.8]}},
 {time:  t2, action{1: [0.75, 0.25], 2: [0.2, 0.8]}},
 {time:  tN, action{1: [0.8, 0.2], 2: [0.2, 0.8]}}}
```

Subjects in this game have control over two axes of action choice. At time t_1, subject 1 was playing the action profile [0.5, 0.5]. At time t_2, he or she switched to [0.75, 0.25] while subject 2's action choice remained unchanged. At time t_3, subject 1 has a strategy of [0.8, 0.2], while subject 2's action is again unchanged at [0.2, 0.8].

3.2 Configuration file fields

Config file fields define values of specific parameters used inside the program, permitting variation between periods. Experimenters about to create new configuration files, and programmers considering new extensions which require additional config file fields, may benefit from a closer look at the structure and syntax of configuration files. Here we give a brief overview of salient features; more details can be found on the ConG documentation website.

Time Each session consists of a number of periods, each of which is specified in a line of the configuration file. As noted earlier, the specification includes the length of each period in seconds, and whether or not the period is to be run in discrete

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1	period	paid	length	subperiods	groupSize	matchType	payoffFunction	min	max	Aa	Ab	Ba	Bb	mixed	selector	matrixDisplay	showHeatmapLegend
2	1	FALSE	20	0	1	pair	matrix2x2	0	20	10	0	18	4	FALSE	pure		
3	2	TRUE	20	0	1	pair	matrix2x2	0	20	10	0	18	4	FALSE	pure		
4	3	TRUE	20	1	1	pair	matrix2x2	0	20	10	0	18	4	FALSE	pure		
5	4	TRUE	20	6	1	pair	matrix2x2	0	20	10	0	18	4	FALSE	pure		
6	5	TRUE	20	0	1	pair	matrix2x2	0	20	10	0	18	4	TRUE	heatmap2d	HeatmapSingle	TRUE
7	6	TRUE	20	0	1	pair	matrix2x2	0	20	10	0	18	4	TRUE	heatmap2d	Corners	TRUE
8	7	TRUE	20	0	1	pair	matrix2x2	0	20	10	0	18	4	TRUE	heatmap2d	HeatmapBoth	TRUE
9	8	TRUE	60	0	6	pair	matrix2x2	0	15	0	15	3	9	TRUE	heatmap2d	HeatmapBoth	TRUE
10	9	TRUE	60	0	6	pair	matrix2x2	0	15	0	15	3	9	TRUE	heatmap2d	HeatmapBoth	TRUE
11	10	TRUE	60	0	12	self	matrix2x2	0	15	0	15	3	9	TRUE	heatmap2d	HeatmapBoth	TRUE
12	11	TRUE	60	0	6	pair	matrix2x2	0	15	0	15	3	9	TRUE	heatmap2d	HeatmapSingle	TRUE
13	12	TRUE	60	0	6	pair	matrix2x2	0	15	0	15	3	9	TRUE	heatmap2d	HeatmapBoth	TRUE
14	13	TRUE	60	0	6	pair	matrix2x2	0	15	0	15	3	9	TRUE	heatmap2d	corners	TRUE

Fig. 10 An example ConG.csv configuration file designed for twelve human subjects. Periods 1–7 run prisoner's dilemma with a number of treatment variations. Periods 8–13 run a hawk-dove game

time (as in periods 3 and 4 in Fig. 10) or continuous time (in which case subperiods parameter is set to 0), and in discrete time the number of subperiods the period is to be evenly divided into. In discrete time players simultaneously make a single action choice in each subperiod: at the end of the subperiod, ConG collects and logs all subjects actions and updates subject displays with the new set of own and counterpart actions to be applied for the subsequent subperiod. This is, in effect, a finitely repeated simultaneous move game, one-shot in period 3 and repeated 6 times in period 4 of Fig. 10.

ConG software treats continuous time architecturally differently than discrete: current actions are changed asynchronously, whenever a user clicks to a new action choice. As noted earlier, ConG has a reaction lag of sorts, the delay between when the user makes a change and when the computer reveals the effect of that change to all users graphically, but with minimum hardware requirements met, that reaction lag is far below any human's ability to perceive it. Thus subjects in a ConG experiment perceive play as continuous, as in a video game.

Payoff function A payoff function is a mapping of players' current (and possibly previous) action choices to a single flow payoff number. Flow payoffs are accumulated over each period and are added over all paid periods to determine subject's total points. ConG allows individual payoff functions that depend on the individual's own group's action choices and the action choices of the matched group. Payoff functions have four inputs: (1) subject, i, the player receiving the payoff, (2) the set of current action vectors in the subject's group, g_i, of which one component may be subject i's own current action vector, (3) the set of current action vectors in the matched group, $m(g_i)$, which could be g_i, and (4) time settings that aggregate flow payoffs. The payoff function, Π_i, maps the first three inputs to a single floating-point number, which (after being multiplied by the length of time since the last relevant change in action vectors) is added to the player's accumulated earnings.

Continuous and discrete time flow payoffs are handled similarly, but distinctly. In continuous time, flow payoffs are calculated and applied over the time interval that subjects action choices remain unchanged. That is, in continuous time payoffs are event driven, payoffs are only recalculated when action choices that might affect current flow payoffs have changed. In discrete time, actions are fixed in each subperiod so flow payoffs are applied for the length of the subperiod grid.

Simple edits to the configuration file support arbitrary symmetric 2×2 matrix games[3] (Figs. 1, 3, and 5 above) and several aggregative games. In aggregative games, the payoffs are a function of own actions and the sum (or some other function) of counterpart player actions. Ready made examples already programmed into ConG include the public goods game featured in Fig. 7 above, as well as threshold coordination games and a few families of Cournot games.

Matching and grouping Each subject is assigned an integer i in the set $\{1, 2, \ldots, n\}$, which is partitioned into r groups $\{g_1, g_2, \ldots, g_r\}$. Subjects in group g_i are matched with all subjects in their match group, $m(g_i)$, (e.g. if group one is composed of subjects two and five then $g_1 = \{2, 5\}$, and if group one is match with group three then $m(g_1) = g_3$. The payoff function then collects the current action choices of these players and, for each player, maps these to a single floating-point number representing current flow payoffs.

In the simplest prisoner's dilemma game—a two-player single-population game in which individual payoffs are a function of own action and matched player actions—a subject's match group is their own group, i.e. $m(g_1) = \{1, 2\} = g_1$. In the hawk-dove population experiment described in the previous section, the two-population matching protocol specifies each individual's payoffs as a function of her own actions and the mean of a matched group's actions, where the matched group is disjoint from the individual's own group. E.g. with four players we might have, $g_1 = \{1, 2\}$, $g_2 = \{3, 4\}$, with $m(g_1) = g_2$ and $m(g_2) = g_1$.

ConG's configuration file allows an experimenter to define how subjects are placed into groups, and how those groups are to be matched with one another—allowing for intricate grouping and matching of participants, creating the potential for standard pairwise games as well as many types of population games. For example, in periods 1–7 the configuration file in Fig. 10 creates a two player prisoner's dilemma game in which an individual player is matched with another individual in standard pairwise fashion (note groupSize = 1 and matchType = pair). In one-population games, as in period 10 in Fig. 10, subjects are matched with members of their own twelve-person group (note the groupSize = 12 and matchType = self). In ConG's two-population matching protocol (periods 8, 9, and 11–13: note the larger groupSize = 6 and matchType = pair), subjects are in one six-person group and matched with a six-person group composed of other subjects.

When it is feasible to reassign subjects to groups in a new period, the default in ConG is to randomize assignments independently each period. As noted, more complex deterministic (or predetermined random) assignments can be implemented via the config file. The basic output file (Ticks) records the assignment each period.

ConG accommodates rather complex groupings, matchings and siloing (i.e. subsets of session participants that do not interact with one another) via subject-group assignment in the configuration file. A number of arbitrary matching and grouping

[3] An experimenter may implement a game with non-symmetric payoffs via subject-by-subject config file assignment. In the config file, below the period-by-period game definition, rows may be used to define each subjects' settings for each period. Non-symmetric payoffs result when different payoffs are assigned to subjects in the same matched group. For an example of subject-by-subject assignment, see the Arbitrary Grouping and Matching section of the ConG documentation website.

examples, along with supporting configuration files, are available in the Grouping and Matching section of the ConG documentation website.

Selector, user interface A user interface gives subjects a way to enter their chosen action, and will display feedback about payoffs and counterpart actions. Popular software packages usually require subjects to use their keyboard to type in numbers, and displays are often text-intensive. ConG emphasizes graphic displays rather than text, and mouse clicks, radio buttons and slider drags for choosing actions. Except in chat room options, players rarely use the keyboard.

In ConG, players select from a set of possible actions via a "selector." The radio button selector, as in Fig. 1, restricts actions to a specified finite discrete set of pure strategies. Selectors for continuous action spaces include both the heatmap and bubbles displays discussed in Sect. 2. The heatmap selector (seen in Fig. 5), combines a mixed current-strategy slider selector for own actions displayed on the vertical axis of the heatmap, with a counterpart strategies on the horizontal axis. The heatmap itself displays the payoffs for each possible action profile. The bubbles selector produces the display seen in Fig. 7; subjects choose actions via a slider along the bottom horizontal axis of the payoff chart.

Certain payoff functions work well, or only work at all, with certain selectors. For example the configuration file payoff function setting "2×2 matrix" only functions with the radio buttons and heatmap selectors; it would have to be adapted to work with the bubbles selector. Likewise, the configuration file payoff function setting "sum" works well with the bubbles selector.

3.3 Data collection and output

Throughout each continuous time period, the ConG server generates "tick events" once every N milliseconds. Each tick event contains a time stamp, the current vector of flow payoffs, the current action profile and other pertinent information generated by the experiment. A separate logging program receives tick events and updates a.csv output file.

Should a catastrophic event occur that ends the experiment abruptly, the.csv tick file will lose only a few of the most recent events, saving as much data as possible. The current version of ConG logs tick events ten times a second ($N = 100$). Thus, very little data will be lost even if some unforeseen event interrupts the experiment.

In discrete time periods, the same information (with subperiod number replacing the time stamp) is generated once each subperiod.

To reiterate, payoffs in continuous time are event-driven and calculated asynchronously, i.e., each time the game state is changed by a subject selecting a new action, the payoffs for the previous time interval over which the game state was constant are calculated and added to affected subjects' payoffs. However, ConG does not presently *record* data asynchronously, but rather at regular intervals called ticks. The default tick interval is 0.1 second and the default rate for updating the output file is every second (every tenth tick).

Users who prefer more or less frequent sampling can adjust these tick settings, but should be aware that much faster updates can sometimes overstrain server resources.

Indeed, it is technically feasible (using ConG's extensibility features) to create event-driven asynchronous output files. We chose not to do so in the default settings for two reasons. First, in some tests we found that subjects clustered rapid strategy changes that could create noticeable processing lags and increase the possibility that the program would crash. Second, the statistical inference techniques used to analyze ConG experiments thus far have generally not required event-driven data.

3.4 Extending ConG

Experimenters with limited computer skills can implement a wide variety of laboratory games simply by judicious editing of the configuration file. Experimenters who have Java programming skills (or who can hire a programmer) can create new payoff functions and novel visual display environments that significantly extend the scope of the software. The source code is modular and open-ended, so researchers can design and run new visually-intensive and/or continuous games without having to bear heavy development costs.

Some background on the software libraries and architecture will be helpful for experimenters thinking about extending ConG. Casual readers, or those who intend to use only new configuration files for existing capacities, can skip the rest of this subsection with no loss of continuity.

3.4.1 Extensibility discussion via an example

We use a non-trivial example to illustrate extensibility: an implementation of Hotelling's model of spatial competition that requires a relatively complicated payoff function as well as modified graphics. In the familiar "linear city" game, players compete over a bounded, finite, single-dimension spectrum of differentiation with fixed prices. Flow payoffs are a function of player market share, i.e., the portion of the action space the player's chosen "location" is nearest. Since existing configuration file settings do not allow for the Hotelling model, a new file is needed to define what players see, how they interact, and how payoffs are defined.

ConG uses a java interface named `PayoffScriptInterface` to allow experimenters to extend the software. In the java programming language, an interface encapsulates abstract methods and constants. ConG's extensibility file is a java class that implements that interface. `PayoffScriptInterface` must contain the method `getPayoff` (where the experiment's payoff function is defined) and the method `draw` (which defines what subjects see on screen). This therefore facilitates the implementation of arbitrary payoff function or graphic environment not already possible via the configuration file.

Relative to programming an experiment in Java from scratch, ConG's extensibility feature offers vast saving of coding and debugging. A Java programmer can plug into a nicely structured system that deals with networking, data logging, control interface and other helpful (or essential) features for running a laboratory experiment. Programmers do not need advanced skills in Java to create only a new payoff function or a new visual feedback display.

The Hotelling extension relies on a display very similar to the bubbles display seen in Fig. 7. Players choose their locations via a horizontal slider at the bottom of the

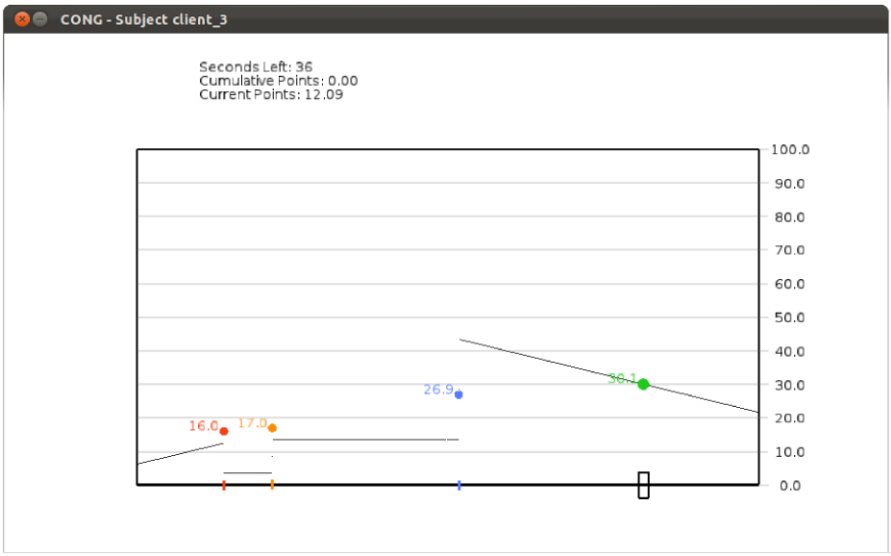

Fig. 11 Screenshot from a 4-player Hotelling game in ConG. This window is from the point of view of the player that is furthest right edge of action space. The *thin discontinuous line* running across the action space reflects the potential flow payoffs this player could earn at all other locations, holding his or her counterparts' locations constant

screen, which defines their location. Figure 11 is a screenshot of the Hotelling game, and shows how players see their own and counterpart player's payoffs in the form of vertical heights of colored bubbles.

The code below is an abbreviated excerpt of the payoff function implementing the Hotelling game. The full Java file that defines the Hotelling game discussed here is over 500 lines in length, of which the payoff function is 55 lines and the draw function taking about 270 lines. Its input is the vector of players' current actions (with actions scaled to a value between zero and one). Since flow payoffs of the two players located closest to the right and left edges of the action space are calculated differently than players in the center, actions are first sorted from lowest to highest. Where "`left`" refers to the action choice of the player who choose the nearest lower action (unless there are no players below, in which case `left = 0`). The "`right`" variable refers to the action choice of the player who choose the nearest higher action (unless there are no players above, which case `right = 1`). The variable "`shared`" is a count of the number of players who have chosen the exact same action. Then for each player's action, denoted "`s`" below, the flow payoff, "`u`", is calculated as follows:

```
// Abridged Payoff Function for Hotelling "Linear City"
  if (left == 0) {
     u = s + 0.5f * (right - s);
  } else if (right == 1f) {
     u = 0.5f * (s - left) + (1 - s);
```

```
  } else {
     u = 0.5f * (s - left) + 0.5f * (right - s);
  }
  return config.get("Alpha") * 100 * (u / shared);
```

Current flow playoffs are charted on screen (again, indicated by the vertical height of players' location dots), and added to subjects' point earnings.

As an example of how one might implement a small adjustment to the Hotelling payoff function used here, suppose one desires a game with an "edge player bonus". That is, players located at the far-left or far-right edge of the "linear city" action space earn an extra 10 points. The following slight tweak will implement this (additional code underlined):

```
// Abridged Payoff Function for Hotelling "Linear City"
// with edge player bonus
  if (left == 0) {
     u = s + 0.5f * (right - s) + 0.1;
  } else if (right == 1f) {
     u = 0.5f * (s - left) + (1 - s) + 0.1;
  } else {
     u = 0.5f * (s - left) + 0.5f * (right - s);
  }
  return config.get("Alpha") * 100 * (u / shared);
```

With some Java programming expertise, one can use ConG to implement virtually any payoff function, and use the current game state (i.e. the full account of current and previous action profiles) to draw novel displays. The full .java file discussed here can be found in the Hotelling Extensibility Example section of the ConG documentation website, along with other examples of how a programmer might extend the existing code.

3.5 Limitations

Although the ConG platform is quite flexible, it does have its limits, some of which have already been noted. Here we list the most important limitations as we see them.

- ConG requires client machines have at least a dedicated GPU with 200 MHz to run properly. Most desktop and laptop computers sold since 2009 are adequate. It is recommended that the machine running the server-control program be fairly powerful, having at least a Dual-Core CPU with 2.4 GHz and 4 Gigabytes of RAM.
- Because ConG is written in Java and uses many supporting software libraries, careful bug checking is advised to ensure that it runs consistently under a new operating system or in a new lab. Documentation to assist installation is available on the ConG documentation website.
- Although off-the-shelf versions can run arbitrary 2×2 bimatrix games and a variety of symmetric n-player aggregative games, ConG requires additional Java programming to run more general n-player games, or games involving Brownian state variables.

- ConG comes equipped with three general purpose graphic displays and subject input modes (bimatrix, bubble and heatmap) each of which is easily configurable, but experimenters who require different sorts of displays must develop new Java modules.
- Standard output files logging subject actions and other experiment data are generated automatically in a format suitable for statistical analysis, but there is little scope for changing the way these ticks files are formatted in ConG. For example, generating complete real time event logs will require Java programming.
- ConG is not designed to run two-sided market institutions such as the continuous double auction. We hope later to develop a separate platform for visually-oriented markets that draws on some of the visual language developed for ConG.

4 Conclusion

Laboratory tests of game theory have become increasingly important over the last 60 years. Recently researchers have started to study games that either have continuous action spaces or are played in continuous time, or both. ConG is a flexible software suite intended to facilitate the laboratory study of such games.

ConG comes prepackaged with several standard interface types, including a matrix game interface, a bubble plot interface for games with continuous (or near-continuous) action spaces, and a heat-map interface also for games with continuous action spaces (including bimatrix games with explicit mixed strategies). ConG also comes packaged with several types of pre-programmed payoff functions. These include several common aggregative games (such as Cournot and Public Goods Games), bimatrix games with explicit mixture choices and several varieties of population games. ConG has several additional configurable features such as matching and grouping protocols, a configurable chat interface and a few basic tools for freezing or delaying revisions to subjects' actions. These interfaces, payoff functions and other features require no programming and can be enabled simply by editing and uploading csv configuration files.

Games studied with ConG so far include various two player bimatrix games, population games, public goods games, Cournot games, pricing games, coordination and threshold games, and Hotelling and Cournot oligopolies. For example, Cason et al. (2013) study continuous time Rock-Paper-Scissors games by using triangular heatmaps that change dynamically.

ConG is an open-ended and modular software suite. Experimenters with Java programming skills can use it to create payoff functions, matching procedures, and visual feedback beyond those ever conceived by its creators.

Acknowledgements This project was made possible by National Science Foundation Grant SES-0925039. We are grateful to the undergraduate programmers who contributed to this project: Joseph Allington, Pranava Adduri, Matthew Browne, Ashley Chard, Michael Cusack, Anthony Lim, Alex (Richard) Lou, Vadim Maximov, Jacob Meryett, Laker Sparks, and Sam Wolpert. Graduate researchers, including Jacopo Magnani, Luba Petersen, Jean Paul Rabanal, and undergraduate researchers Tamara Bakarian, Thomas Campbell, Cosmo Coulter, Jenelle Feole, Wade Hastings, Keith Henwood, and Richard Shall also made significant contributions. We would also like to thank Urs Fischbacher for his comments.

References

Cason, T., Friedman, D., & Hopkins, E. (2013, forthcoming). Cycles and instability in a rock-paper-scissors population game: a continuous time experiment. *Review of Economic Studies*.

Cox, J. C., & Swarthout, J. T. (2006). EconPort: creating and maintaining a knowledge commons. In C. Hess & E. Ostrom (Eds.), *Understanding knowledge as a commons: from theory to practice*. Cambridge: MIT Press.

Fischbacher, U. (2007). Z-tree: Zurich toolbox for ready-made economic experiments. *Experimental Economics*, *10*(2), 171–178.

Friedman, D., & Oprea, R. (2012). A continuous dilemma. *The American Economic Review*, *102*(1), 337–363.

Goeree, J. K., & Holt, C. A. (2001). Ten little treasures of game theory and ten intuitive contradictions. *The American Economic Review*, *91*(5), 1402–1422.

Kagel, J. H., & Roth, A. E. (1997). *Handbook of experimental economics*. Princeton: Princeton University Press.

Lipps, D. B., Galecki, A. T., & Ashton-Miller, J. A. (2011). On the implications of a sex difference in the reaction times of sprinters at the Beijing Olympics. *PLoS ONE*, *6*(10), e26141.

Oprea, R., Friedman, D., & Henwood, K. (2012). Separating the Hawks from the Doves: evidence from continuous time laboratory games. *Journal of Economic Theory*, *146*(6), 2206–2225.

Rapoport, A., & Orwant, C. (1962). Experimental games: a review. *Behavioral Science*, *7*, 1–37.

Schwarz, M., & Takhteyev, Y. (2010). *Half a century of public software institutions: open source as a solution to hold-up problem*. NBER Working Paper Series, No. 14946.

Simon, L. K., & Stinchcombe, M. B. (1989). Extensive form games in continuous time: pure strategies. *Econometrica*, *57*(5), 1171–1214.

Thrall, R. M., Coombs, C. H., & Davis, R. L. (1954). *Decision processes*. New York: Wiley.