A Module For Measuring Expectations in Laboratory Experiments

Nicolás Aragón[*] UC3M

Rasmus Pank Roulund[†] Danmark's NationalBank

August 20, 2019

Abstract

This paper presents a elicitation tool for economic experiments based on Harrison et al, 2017. The tool is user-friendly and enables subjects to forecast the movements of a continuous or discrete variable. The software can address important issues present while eliciting beliefs, such as hedging, risk attitudes and intensity of beliefs. The module is easy to integrate with HTML-based experimental software kits such as oTree (Chen et al., 2016)

1 Introduction

Expectations play a central in many economic phenomena, both in microeconomics, macroeconomics and finance. Though important, rigorous empirical analysis is particularly challenging given that expectations are not directly observable. It is not surprising, therefore, that expectations have come to be analyzed heavily in experimental settings, where elicitation seems possible.

The literature has normally attempted to obtain point estimates of forecasts and, at most, a confidence band. Accuracy is normally incentivized by using monetary rewards. However, even in an experimental setup carefully eliciting expectations represents challenges.

First, by requesting agents to give a point estimate with a confidence band, there is often an implicit assumption that expectations follow a symmetric distribution, e.g. a uniform distribution. This leaves out the possibility that agents may have beliefs over two possible scenarios disjointly (for example, fundamental and bubble values in an asset market). Thus, a point estimate leaves valuable information out of the study and may even bias the elicited belief. Moreover, it

[*]Corresponding author. E-mail: `naragon@econ.uc3m.es`
[†]E-mail. `rasmus@pank.eu`

does not provide information on second moments of expectations, i.e, the role of confidence in a forecast. Second, if subjects need to perform more than one task and are incentivized for both, they may choose to diversify risk between the activities, thus hedging and distorting the elicited beliefs, [Armantier and Treich, 2013]. For instance, if agents need to trade assets and provide forecasts, they may trade at a high value and report expectations at a low value, thus ensuring a less variable payoff. The literature has normally avoided giving large payments for expectations, as to not distort choices via hedging. However, if the objective of the study is to carefully analyze expectations, it can be argued that the reporting of beliefs needs to be incentivized with large expected payoffs. Finally, risk perceptions may change the reported beliefs of subjects. For instance, more risk averse agents tend to report "flatter" distributions than their true beliefs [Harrison et al., 2015].

In this paper we present an open source JavaScript module that enables a clean elicitation of a distribution of beliefs of a discrete or continuous variable, using the method laid out in Harrison et al. [2017]. It allows to control for risk attitudes and hedging possibilities. The module is graphically appealing and user-friendly. It easily integrates with the new wave of HTML-based experimental software, such as oTree [Chen et al., 2016]. It has been successfully used in asset market experiments in Barcelona [Aragón and Pank Roulund, 2019], where it was verified that participants felt comfortable with the software and learned to use it quickly.

The source of this module can be found at https://gitlab.com/pank/forecast.js. Contributions, requests and improvements are welcome and can be submitted via the issue tracker there. The rest of the paper is organized as follows. Section 2 describes the theoretical background that explains the design choices for the module. Section 3 explains how to use the module in an experiment. Section 4 shows examples of the sort of data that has been collected with forecast.js. Section 5 analyzes the learning curve for users and the ease of use. In the appendix we present a standalone HTML demo. An online oTree version can be found on the module's Gitlab page.

2 Background

In this section the theory and background literature that has led to the design choices of the forecast.js module are presented. The principal task of the module is to elicit subjective expectations about a particular event, such as the movement of a continuous or discrete variable. A typical example would be the price of an asset in a future period, but the module can be applied to any other scenario, such as quantities to be produced or voter turnout.

Fundamentally, expectations are subjective probabilities, meaning the ''degree of belief regarding the likelihood of events" [Karni, 2014]. As such, the subjective probabilities in which we are interested follow the usage proposed by Savage [1954] and others before. See Karni [2014] for a full treatment of the historical developments of subjective probabilities.

2.1 Theoretical Considerations

In the forecast.js module, participants report a probability mass function characterized by $\{r_k\}_{k=1}^K$ where K is the number of intervals in a discretized variable of interest. The sum over the grid points equals one, $\Sigma_{k=1}^K r_k = 1$, as it is a probability distribution. Once a participant's forecast distribution is known, a scoring rule needs to be used under the assumption that interval j contains the ex post realized value of the measured variable. A key element to properly elicit expectations is a scoring rule.

A scoring rule is a tool to map the elicited expectations into a score once the accuracy of the expectation is known. Scoring rules may be used to incentivize forecasters to reveal their "true" beliefs in addition to increasing their stakes and effort in the task. A scoring rule must provide incentives such that the participants can maximize their expected outcome by revealing their true beliefs.

In addition to showing the mathematical aspect of eliciting expectations, Savage [1954] also discusses a number of issues with respect to eliciting expectations using scoring rules. A particularly interesting scoring rule for the purpose of experimental economics is the quadratic scoring rule (QSR) first proposed by Matheson and Winkler [1976]. As Harrison et al. [2017] show, the QSR—and any proper scoring rule for that matter—has a number of useful properties as shown below. We denote the reported beliefs as $\{r_k\}_{k=1}^K$and the true beliefs as $\{q_k\}_{k=1}^K$. The QSR has the following properties[1]

1. A participant only reports positive expectations about a specific event, e.g. $r_i > 0$, if his or her true belief of the outcome is positive, i.e. $q_i > 0$.
2. If a risk-averse or risk-neutral participant believes two events are equally likely, say $q_i = q_j$, then the reported expectations are the equal, i.e. $r_i = r_j$.
3. If a risk-averse or a risk-loving participant report the same expectations for two events, e.g. $r_i = r_j$, then the true beliefs are also equal, $q_i = q_j$.
4. If a participant's true beliefs are represented by a symmetric distribution then the means of the reported beliefs and the true beliefs are equal. This is true for unimodal and multimodal distributions alike.
5. If the participant's reported expectations are a symmetric distribution, then the true belief distribution is also symmetric.

[1]See Harrison et al. [2017] for further details.

6. The more risk-averse a participant is, the more the reported distribution will resemble a uniform distribution with the support of the true distribution of beliefs.

In addition, Harrison et al. [2017] report that the reported distributions are "very close" to the true subjective distributions for a wide range of empirically plausible risk attitudes, provided individuals are Expected Utility Maximizers. The discretized version of the QSR can be expressed as follows [Harrison et al., 2017]:

$$s_j(r) = \kappa \left[\alpha + \beta \left(2 \times r_j - \Sigma_{k=1}^{K} r_k^2\right)\right] . \tag{1}$$

Here, κ, α and β are parameters. The scoring rule can be configured directly in the forecast.js module. Parameters α and β penalize spread forecasts, and κ is a scaling parameter.

There are two main issues to address when eliciting beliefs: risk attitudes and the possibility of distorted beliefsdue to hedging.

Hedging arises when the agent needs to perform more than one task. For example, in Aragón and Pank Roulund [2019], participants need to make transactions and forecast prices; and both activities are rewarded. Hanaki et al. [2018] show that subjects indeed show different behavior when they need to both trade and make forecasts. Thus, a payment based on either forecasting or trading performance chosen randomly at the end is better than paying subjects based on both. In this way, subjects need to put maximum effort into both tasks. The software allows to randomize at the end whether the participants are paid by their transaction efforts or by their forecasting efforts. The parameter κ in Equation (1) can be used achieve this. This is further discussed in Section 3.1.

Secondly, risk attitudes may affect the reported beliefs. In particular, risk averse agents tend to report "flatter" distributions than their true beliefs. Even though the levels of risk aversion in the laboratory seem to be low as to significatively distort beliefs, a method of binary lotteries [Harrison et al., 2015] can risk-neutralize subjects enabling the elicitation of beliefs under more general preferences. The software can be easily adapted to perform this task by converting the points into lottery tickets.

3 Using the module

This section shows how the forecast.js module is used and how it may be adapted to the needs of individual experimenters. The main purpose of the module is to enable experimenters to easily obtain measures of a participant's beliefs about the movement of a continuous or discrete variable. By default, this is assumed to be a price.

The forecast module is written in JavaScript and is therefore supported in all major browsers, such as Mozilla Firefox and Google Chrome. The d3.js SVG

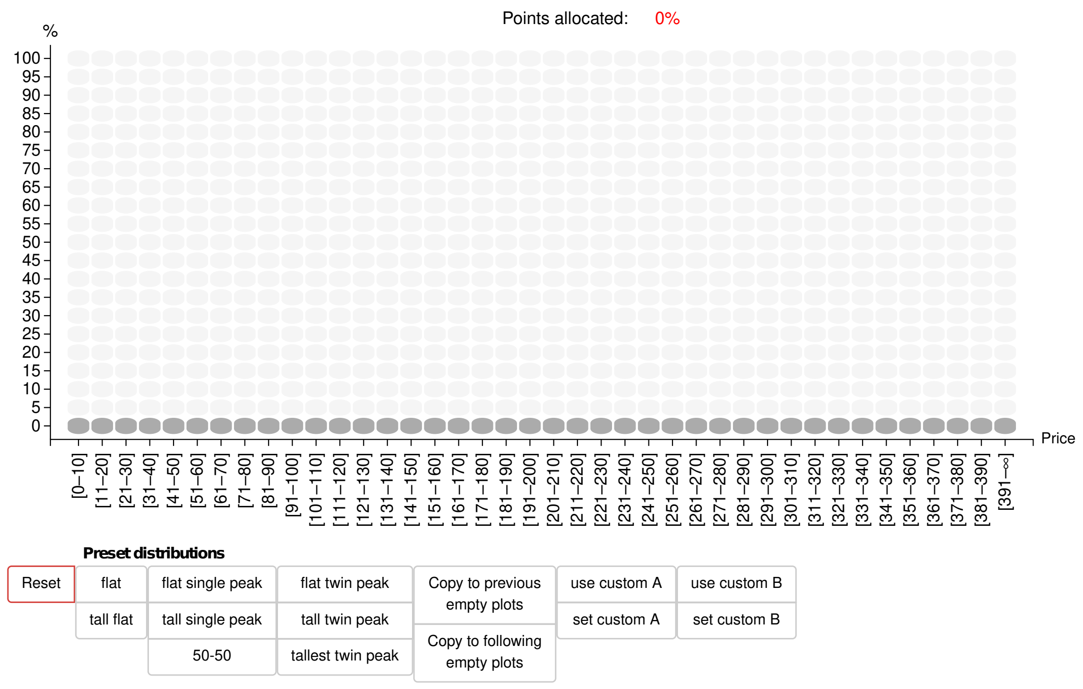

Figure 1: Initial plain canvas presented to subjects.

is the primary library[2] As it is a simple JavaScript module, it can be used with experimental toolkits utilizing web technologies, such as oTree [Chen et al., 2016]. The module makes it possible for participants in economic experiments to easily produce a visual representation of their beliefs, in the form of a probability mass function, by drawing it with their mouse.

When participants are first presented with the tool they observe a ‘’blank canvas” as shown in Figure 1. Along the primary axis, the value of the variable is shown. In the example, the measured variable is the price of asset shares. The domain of the canvas in the picture is $[0, 400]$. The range can be changed by setting the variables `xmin` and `xmax` before loading the module. The picture contains 40 bins, meaning that each bin covers a price range of 10. The number of bins along the primary axis is determined by the variable `xbins`, which can be set before loading the module. The bin width is automatically calculated based on `xmin`, `xmax` and `xbins`. By default, the name of the primary axis is “Price”, but this can be changed by setting the variable `xname`. The number of bins on the secondary axis can be controlled by setting the variable `ybins`. The default value is 20, meaning that participants have 20 tokens to allocate. In other words, each token represents 5% confidence in the bin. To make the

[2]SeeBostock et al. [2011].

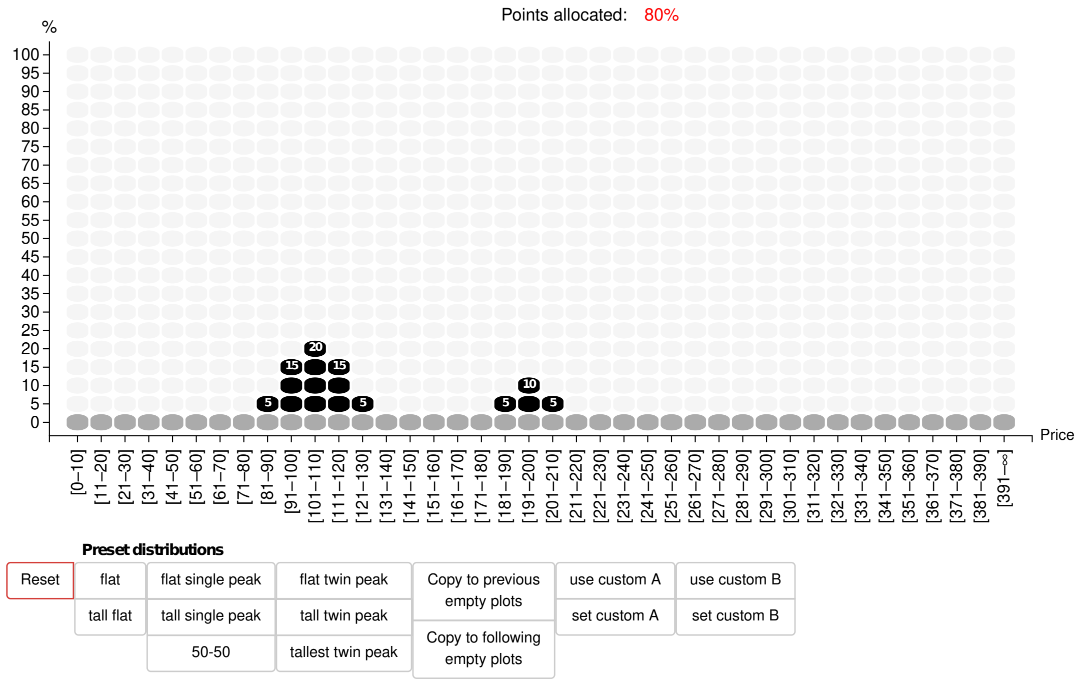

Figure 2: An intermediate forecast. The module warns the user that the forecast is not complete until all "tokens" have been allocated. have been allocated

elicitation intuitive for the participant, it is suggested that this value be kept as a multiple of 5. As the module is meant to produce a probability mass function, the range is fixed at [0%, 100%]. Nonetheless, the name of the secondary axis can be changed by setting the `yname` variable. The size of the canvas, measured in pixel [3] can be set using the variables `fheight` and `fwidth`, representing height and width respectively.

Participants can draw probability mass functions as shown in Figure 2 by using the mouse. The black bars denote the expectations of the forecaster. Above the tokens, the percentage value corresponding to the height of the bar in percentage terms is displayed, representing the confidence. It is possible to drag-and-drop the bars along the primary axis, or to change the height of individual bars by clicking on the tokens. The module is also endowed with customizable pre-configured probability mass functions, making it quicker for participants to forecast. It is also possible to save a specific forecast to use later. The forecast can be reset or finalized using the respective buttons.

[3]Note that this refers to a so-called "CSS pixel", as opposed to the more familiar physical pixels. CSS pixels vary in size depending on the device, as explained by Chien and Nyman [2013]

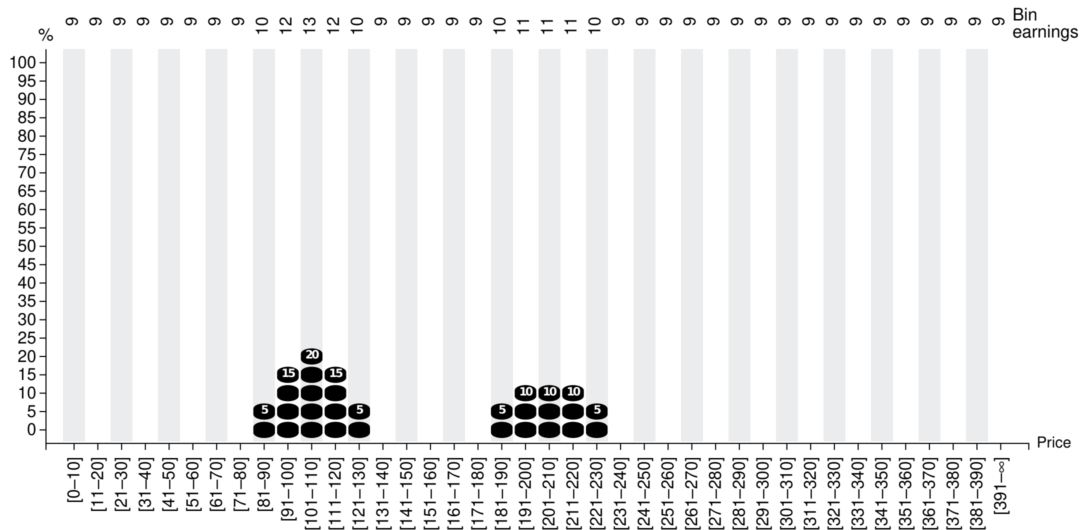

Figure 3: A finalized forecast with the implied scores shown at the top of the screen.

Figure 3 displays a finalized forecast. A forecast can only be finalized once all tokens have been allocated by the forecaster. At the top of the canvas, the score for each bin is displayed, given the scoring rule and the forecaster's distribution. The forecaster is free to change his or her forecast after observing the scores. This can be done by clicking the "edit" button (not shown). Alternatively, the forecaster can accept the forecast and move on to the next stage of the experiment. The score of a specific bin typically depends on the shape of the entire forecast distribution. Thus, it can only be calculated once the distribution has been finalized. By displaying the scores directly above each bin, the experimenter can focus on explaining the broad mechanism of the scoring rule (i.e, penalizing dispersion and increase in accuracy) rather than to focus on the mathematical details.[4] By default, the quadratic scoring rule is used by the forecast.js module, but any scoring rule can be used. To apply a different scoring rule, the experimenter can set the variable `scoring rule` to a function accepting two arguments. The first argument is an index of the distribution to be measured (corresponding to j in Equation 1), while the second argument is the distribution. For instance, to use the logarithmic scoring rule,

$$s_j(r) = log(r_j)$$

it is sufficient to set,

```
scoring rule=function(j,r)return(Math.log(r[j]));
```

[4]This approach was also taken by Harrison et al. [2017].

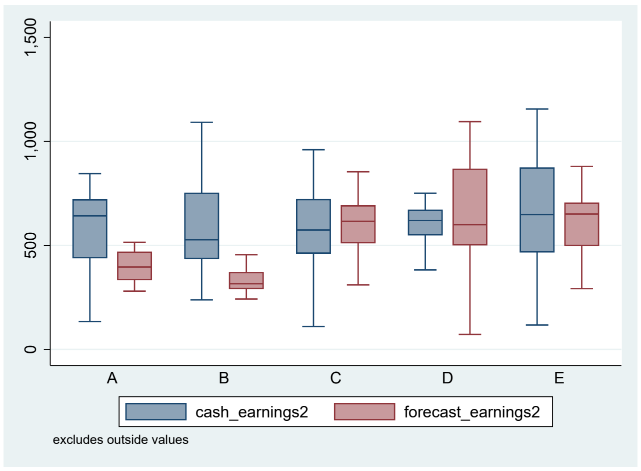

Figure 4: Payment distributions across sessions. Parameters are $\alpha = 6$, $\beta = 14$ in all sessions. $\kappa = 4$ in sessions C, D and E, and $\kappa = 2$ in sessions A and B.

3.1 Calibration

There are three main parameters in the QSR to be calibrated in the module. First, α and β must provide incentives for accurate forecasts. In that way, they need to penalize spread forecasts and reward accuracy. In Aragón and Pank Roulund [2019], α and β are chosen such that payments are either zero or slightly negative in very spread out forecasts. Regardless, in an experiment when forecasting task is repeated, final negative payments are very unlikely.

Secondly, the parameter κ can be used to avoid hedging when participants must do more than one task. This is achieved by rewarding participants randomly for either one of the activities. To do so, κ must be calibrated so that the expected payment under both activities is the same. In Figure 4, payments are presented transactions and forecasting. The amount of cash in the economy is fixed in expected terms, as there are the initial endowments of each participant and the expected dividends times the number of periods. Aragón and Pank Roulund [2019] use pilot sessions to calibrate average payments. Figure 4 shows average payments under both tasks for different parameters levels. As we can see, a value of κ of 4 under these conditions generates equal expected payments for participants under both tasks.

3.2 Accessing the Data

As the expectations about the future movement of the variable in question are elicited, the values must be collected and stored. Typically, the experimental software will take care of storing the data once it knows how to collect it. The forecast.js module has two main outputs. The first one is an array representing the distribution of expectations that has been elicited. This is a standard probability distribution summing to one. It has the same number of elements as there are bins along the primary axis. The second output is a vector of scores, containing an individual score for each bin, assuming that the bin contains the correct value ex post. To illustrate, consider an experiment where a participant forecasts some value over six bins. Consider the following example of a forecast,

$$\hat{r} = (0, 0.15, 0.35, 0.35, 0.15, 0)\,. \tag{2}$$

In this example, the participant states that the probability of the true value materializing in the interval represented by the second bin is 15%, and so forth. This value is stored as the string `=[0, 0.15, 0.35, 0.35, 0.15, 0]` by forecast.js. By default, the distribution is stored in the `dist` variable. The name of the storage field can be changed by setting the variable `dist var prefix`. If the participant is asked to do more than one forecast, they will be stored in the variables `dist1, ..., distn` (see Appendix for details). Scores are also stored in a variable named, by default,`score`. The prefix can be changed by setting the variable `score var prefix` before loading the forecast.js module. Using the quadratic scoring rule with $\alpha = \beta = 10$, and using the above example distribution, the scores of $\hat{r}$ would be $(7, 10, 14, 14, 10, 7)$. These would be stored in the score variable as `="[7, 10, 14, 14, 10, 7]"`.

4 The generated data

This section illustrates the type of data that is generated with the forecast.js module. We first show an example of a forecast series generated with the module and then turn to the time consumption of the usage of the module. As the timing data shows, participants are generally quick at learning how to use the module and at making forecasts. Finally, we turn to the precision of module over time, to see to what extent learning takes place.

We use data from Aragón and Pank Roulund [2019]. In that experiment, traders are asked to predict the movement of the asset price at the beginning of each round, i.e. a one-period-ahead forecast. The experiment follows the structure of a standard experimental market [Smith et al., 1988]. The price is determined via a call market for a single asset. The asset pays a random dividend each round, with an expected value of 12 each period. The fundamental value of the asset is determined as $f(t;T) = 12(T + 1 - t)$ where T is the total number of rounds in the market and t is the current round. In these markets, prices typically tend to start below the fundamental value, increase above it, and finally burst to a level below it [Palan, 2013].

Figure 5 illustrates the predictions of a participant throughout an entire experiment. The vertical line displays the realized equilibrium price and the histogram

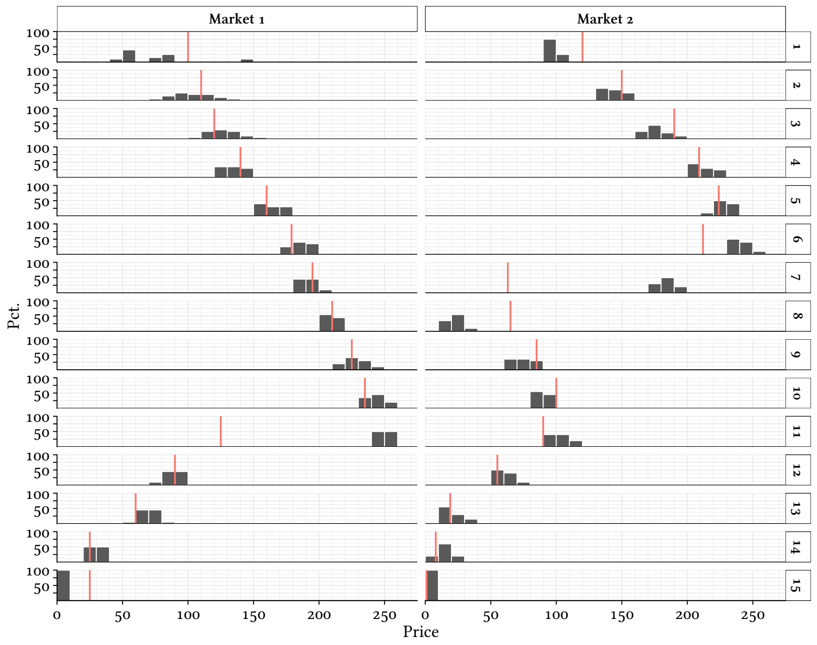

Figure 5: Example of individual expectations. The black histogram corresponds to the price expectations of the participant, and the thin, red vertical line corresponds to the ex post realized price.

illustrates the participant's forecast. The participant took part in an experiment with two consecutive markets, each consisting of 15 periods. Market 1 is shown in the left column and Market 2 is shown in the right column. Each line corresponds to a round in the market. In the example, the bin width is 10 points, corresponding to a confidence of 5%. This means there were 20 tokens to be allocated by the participant, who, in turn, was scored using a quadratic scoring rule. The graph illustrates the type of data that can be elicited with the forecast.js module. In this particular example, the participant is not certain about the exact range of the price in the first period of market 1 (the left column), and thus chooses a disjoint probability mass function. However, after the second period, the participant ceases to use disjoint distributions. The participant has positive beliefs in the right price brackets for all remaining periods. In general, the participant also becomes more confident about the price development from period 1 to period 11, in the sense that the range of the distribution is overall decreasing. After market 1 crashes in period 11, the participant manages to correctly predict the price again. In market 2, the participant generally manages to correctly predict the price between period 2 and 6, and again from period 11 until the end of the market.

While this particular participant tends to spread his or her forecasts over two to three bins, at times (s)he does use wider expectation distributions, such as in period 3 in both markets. Often, one bin, not necessarily the median, receives more weight than other bins, as shown in period 13 in market 2. As such, the example illustrates to what extent it is possible to gather more detailed information about the participant's expectations, compared with when imposing e.g. a fixed deviation of the distribution.

4.1 Timing Considerations

An important aspect of the forecast.js module is that it is quick to use for participants, so as not to delay the experiment. This means that experimenters can elicit full expectation distributions from participants without a big toll on the duration of the experiment. Moreover, the results show that the elicitation time is decreasing as the participants become more accustomed to it, lowering the total time costs if an experimenter plans to elicit expectations several times.

4.2 Understanding Questions

We propose two sets of understanding questions to be used with this module as to make sure participants understand the tool. The HTML version of these questions can be found in the Gitlab page of the module, and can be easily implemented in oTree-based experiments.

The first set of questions requires the participants to get familiar with the tool. It requires them to put a certain amount of tokens in certain prices and explains

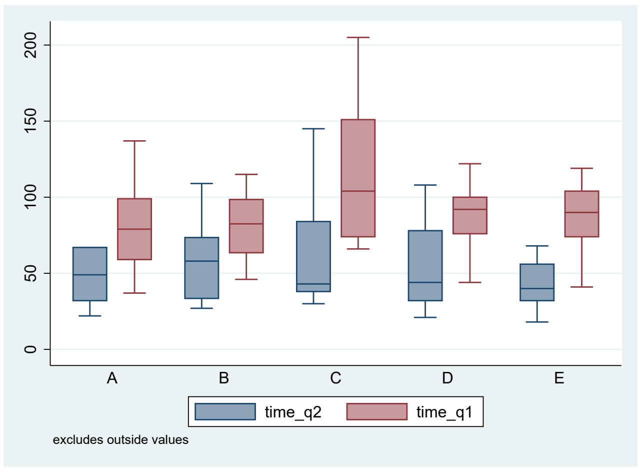

Figure 6: A box plot of time spent (in seconds) on the understanding questions.

the meaning. For example, they are asked to put half of the tokens in the bin corresponding to a price of 100, and it is explained that this means they think that half of the times the price will be that one. Subsequently, participants are asked to drag the created distributions, and are required to modify a preset distribution. The second set of questions requires students to understand the earnings they would get once a forecast is finalized. The time taken to answer these questions in the Aragón and Pank Roulund [2019] experiment is shown in Figure 6. Overall, it takes few minutes for the participants to get acquainted with the tool.

4.3 Timing over time

Figure 7 displays a box and whisker diagram for the time spent on eliciting expectations in each round in the Aragón and Pank Roulund [2019] experiment. The lower hinge of the box corresponds to the lower quartile of the forecasting duration, the middle line is the median and the upper hinge is the upper quartile. The whiskers correspond to the largest (smallest) observation at most 1.5 times the interquartile range (i.e. the height of the box) away from the top (bottom) hinge. The line is a simple linear trend showing that, on average, participants become faster at using the module over time. As the graph shows, most participants use between 72 and 110 seconds in the first round where they use the tool. The mean and the median are 92 and 88 seconds respectively. After the first period, the median time generally decreases. One important caveat is that we cannot disentangle to what extent the time is decreasing due to increased familiarity with the module and to what extent it is decreasing because participants understand the game better. In a regression setup we can partially control for characteristics of the underlying experimental data.

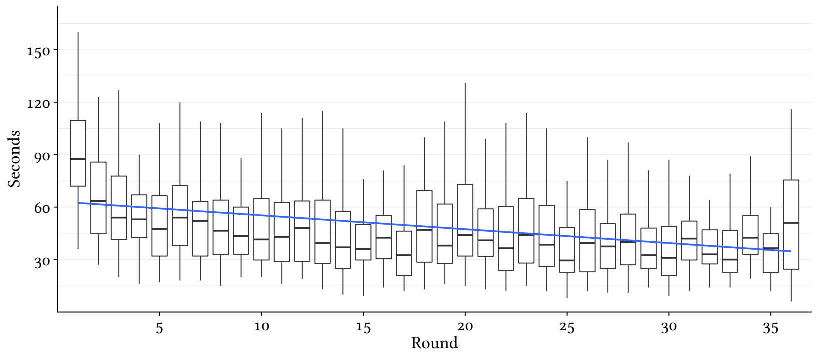

Figure 7: Time spent on eliciting expectations. $N = 60$ observations per round from rounds 1–30 and $N = 24$ observations in periods 31–36.

	Elicitation time			
	(I)	(II)	(III)	(IV)
Intercept	71.76***			
	(3.05)			
t	-2.23***	-2.34***	-3.31***	-2.72***
	(0.30)	(0.29)	(0.37)	(0.38)
t^2	0.04***	0.05***	0.006***	0.006***
	(0.001)	(0.001)	(0.001)	(0.001)
Participant FE		Yes	Yes	Yes
Session FE			Yes	Yes
Market FE			Yes	Yes
Round FE				Yes
Obs	1944	1944	1944	1944
R^2	0.08	0.32	0.33	0.34

Table 1: Regression showing the impact of repeated use of forecast.js.

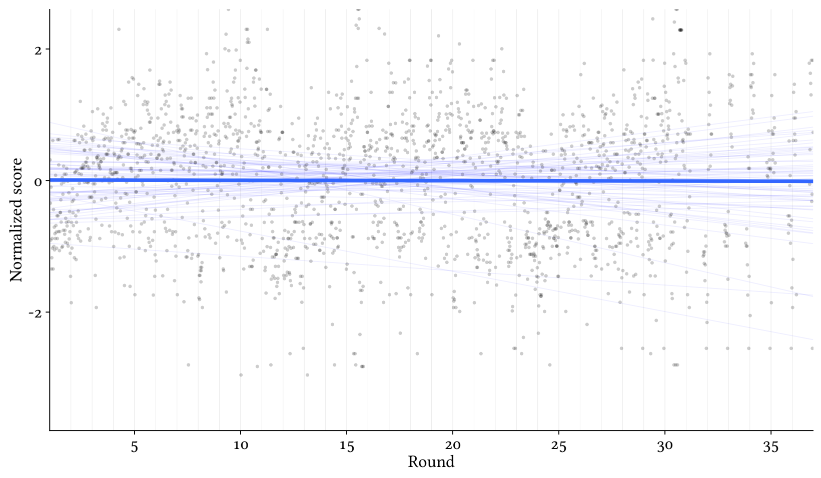

Figure 8: Normalized scores over time. Each of the dots corresponds to a normalized score. Scores are normalized within sessions. Each of the thin lines correspond to the score development of a single participant. The thick line corresponds to the average development of scores. Scores are calculated using the quadratic scoring rule (1).

Table 1 shows some specifications in which the time (in seconds) spent using the tool is regressed on the round, t, and a quadratic component, t^2. A set of fixed effects is included, to account for participants' characteristics and the experiment. The data contains a total of 1944 observations, split across 60 participants. The round number corresponds to the number of times the participants have forecast the experiment. The first column (I) shows that the elicitation time decreases by approximately 2 seconds per round for the first 3 rounds, after which it drops by 1.5–1 seconds for rounds 4–15. This indicates that participants spent some time getting acquainted with the elicitation tool, but they were relatively quick to learn how to use it. Column (II) includes individual fixed effects for each participant. As can be seen, the main conclusion holds: time spent is decreasing with repeated usage of the module, but is more pronounced in the first rounds. Column (III) and Column (IV) include additional fixed effects. In summary, Figure 7 and Table 1 show that the elicitation of participants' expectations is fast with the forecast.js module, even in a relatively complex situation, and the time spent per elicitation decreases with repetition.

4.4 Prediction precision over time

It is important that the elicited expectations are consistent across the experiment and that the quality of the elicitation is not hindered by the complexity of the elicitation tool. In this section it is examined whether people tend to perform better as they gain experience with the elicitation tool. Figure 8 displays normalized scores across sessions for each period. The thick line displays the overall trend, which gives equal weight to each observation. The thin, gray lines are the trend lines for each of the 60 participants. The main message is that, on average, participants do not improve at forecasting. One interpretation of this result is that the learning curve for the module is not steep. This is a desirable quality, as it implies that one can interpret the participants' expectations equally, irrespective of where in the time spectrum they were elicited. We also explored learning effects via a regression of individual scores (as computed by the quadratic scoring rule) with round number, individual fixed effects, and several measures of forecast biases. We find no direct timing effects.

5 Conclusion

This paper introduces a new module for eliciting expectations from participants in economic experiments. It can be applied to a wide array of economic problems in both microeconomics (voter turnout, fairness, collusion) and macroeconomics and finance (inflation, credibility, asset markets).

The module enables researchers to elicit full subjective probability mass functions of individual expectations and allows to take into eliminate the problems of risk attitudes and hedging. The module can be easily incorporated in HTML-based frameworks, such as oTree and can also be adapted to individual experimenters' needs.

Using real data, we show that participants tend to get faster at using the module with repeated usage. This suggests that experimenters may use it to elicit expectations several times. We also show that participants' scores do not tend to change significantly over time, suggesting that they are not improving at making predictions.

References

Nicolás Aragón and Rasmus Pank Roulund. Certainty and decision-making in experimental asset markets. Thesis chapter, European University Institute, 2019. URL `http://pank.eu/experiment`.

Olivier Armantier and Nicolas Treich. Eliciting beliefs: Proper scoring rules, incentives, stakes and hedging. *European Economic Review*, 62:17–40, 2013.

Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. $\mathbb{D}^3$: Data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12): 2301–2309, 2011. URL `http://vis.stanford.edu/papers/d3`.

Daniel L. Chen, Martin Schonger, and Chris Wickens. oTree—an open-source platform for laboratory, online, and field experiments. *Journal of Behavioral and Experimental Finance*, 9:88–97, 3 2016.

Tim Chien and Robert Nyman. Css length explained. Mozilla hacks, Mozilla, 2013. URL `https://hacks.mozilla.org/2013/09/css-length-explained/`.

Nobuyuki Hanaki, Eizo Akiyama, and Ryuichiro Ishikawa. Effects of different ways of incentivizing price forecasts on market dynamics and individual decisions in asset market experiments. *Journal of Economic Dynamics and Control*, 88:51–69, 2018.

Glenn W. Harrison, Jimmy Martínez-Correa, J. Todd Swarthout, and Eric R. Ulm. Eliciting subjective probability distributions with binary lotteries. *Economics Letters*, 127:68–71, 2015.

Glenn W. Harrison, Jimmy Martínez-Correa, J. Todd Swarthout, and Eric Ulm. Scoring rules for subjective probability distributions. *Journal of Economic Behavior & Organization*, 134:430–448, 2 2017.

Edi Karni. Axiomatic foundations of expected utility and subjective probability. In Mark J. Machina and W. Kip Viscusi, editors, *Handbook of the Economics of Risk and Uncertainty*, volume 1, chapter 1. North Holland, 2014.

James E. Matheson and Robert L. Winkler. Scoring rules for continuous probability distributions. *Management Science*, 22(10):1087–1096, 1976.

Stefan Palan. A review of bubbles and crashes in experimental asset markets. *Journal of Economic Surveys*, 27(3):570–588, 2013.

Leonard J. Savage. *The Foundations of Statistics.* John Wiley, New York, 1954.

Vernon L. Smith, Gerry L. Suchanek, and Arlington W. Williams. Bubbles, crashes, and endogenous expectations in experimental spot asset markets. *Econometrica*, 56(5):1119–1151, 1988.

6 Appendix

6.1 Using forecast.js in a standalone HTML page

In this section we show a minimal example of how to include the forecast module on a plain HTML page.

```
    <!DOCTYPE html>
<html lang="en">
  <head>

    <meta charset="utf-8">
    <title>Forecast.js elictitation module</title>

    <!-- Load the CSS styles needed for forecast.js -->
    <link rel="stylesheet" href="forecast-js.css"/>

    <!-- forecast.js depends on d3.js.   -->
    <!--I assume both are available in the "js" folder -->
    <script type="text/javascript" src="/js/d3.v3.min.js"></script>
    <script type="text/javascript" src="/js/forecast.js"></script>

  </head>

  <body>

    <!-- First, a block with usage instructions -->

    <div id="instructions">
      <h2>Price prediction instructions</h2>
      <p>
        Use the forecast tool to predict future prices.
      </p>
      <p>
        Make distrbutions by clicking on the gray boxes or by using
        one of the predefined distribution available via the buttons
        under the tool.
      </p>
      <p>
        After you have made a distribution you can
        change the location by dragging it with the mouse.
      </p>
      <p>
        When you are satisfied with your choice click <em>Finalize</em>.
      </p>
      <p>
        Note that once you have clicked finalized, the score is
        available as a JSON string under the element with the
        IDs <code>dist</code> and <code>value</code>.
      </p>
    </div>

    <!-- The following will initialize the an instance of forecast.js -->
    <script>

      <!-- We can change the settings using pre-defined variables -->
      var xname = "Price";
      var xmax = 100, xbins = 10;
      var alpha = 1, beta = 1;

      <!--   Give the instance the id "forecast";  -->
      <!--   Saving the instance as element "forecast_instance"  -->
      forecast_instance = add_forecast_widget("forecast");
    </script>
  </body>
</html>
```

6.2 Using forecast.js with oTree

In this section we show a minimal example of how to include the forecast module in the experimental software suit oTree [Chen et al., 2016]. To initialize oTree, set up a new virtual oTree environment, make a new oTree folder, and install oTree. While the oTree manual explains this in detail, found at `https://otree.readthedocs.io/en/latest/install.html`. The following should suffice on UNIX-like systems:

```
# First, create a new oTree environment and activate it
mkdir oTree; cd oTree
virtualenv3 env
source env/bin/active

# Install the latest version of oTree
pip3 install -U otree-core

# Create a new oTree instance
otree startproject elicitation-project
cd elicitation-project

# Create a new "app".
```

```
otree startapp simple_elicitation
```

This initializes a new oTree installation in its own virtual environment and creates a new oTree "app" called `simple elicitation`. This app contains three important elements: the files `models.py`, `pages.py`, and the `templates` folder.

First, create a new folder called static and copy `d3.v3.min.js`, `forecast.js` and `forecast.css` into this folder.

The static folder holds images, style files and JavaScript programs. The Django documentation of the static folder can be found here

`https://docs.djangoproject.com/en/1.11/howto/static-files`.

6.2.1 Setting up `models.py`

Next, we look at the `models.py` file. The `models.py` file defines and structures the space in which the data is stored. For instance, it defines variables/columns that are to be created in the database that stores the data. To store the elicited forecasts from participants, we must thus create a field to store the elicitation in. The `forecast.js` module automatically stores the elicitation in a JSON array, stored as a string,as mentioned above. In the example below, the fields `pred` and `score` in the `player` class are examples of how the elicitation results can be stored.

The rest of the file is standard. The first part is imports and are mostly setup automatically by oTree itself. In addition, some useful libraries and functions are imported, starting with `random`.

In the Constants class of the file the variables for the `forecast.js` module can be set up. The Player class sets up the fields in which participants' data are stored. In particular, in this example, the fields `forecast.js pred` and `score` are stored per participant in the player class.

In the group class, we find the score and the correct number of points in the bin in which the realized price fell.

```
      # -*- coding: utf-8 -*-
# <standard imports>
from __future__ import division
from otree.db import models
from otree.constants import BaseConstants
from otree.models import BaseSubsession, BaseGroup, BasePlayer
from otree import widgets
from otree.common import Currency as c, currency_range
from django.conf import settings
# </standard imports>
```

```
import random
import json
from bisect import bisect_left
from quantile import quantile
from django.core.validators import RegexValidator

author = 'Rasmus Pank Roulund and Nicolas Aragon'

doc = """
Showcase of forecast.js
"""
keywords = ("Forecasting", "Finance", "Elicitation", "Trade")


class Constants(BaseConstants):
    players_per_group = None

    ## Prediction parameters
    prediction_kappa = 5
    prediction_alpha = 6
    prediction_beta = 20

    xbins_n = 40
    xmax = 400
    xbins = range(0, xmax, xmax//xbins_n)
    ybins_n = 100//5

class Player(BasePlayer):
    # <built-in>
    subsession = models.ForeignKey(Subsession)
    group = models.ForeignKey(Group, null = True)
    # </built-in>

    ## Elicitation score and number of points in right bin
    elicitaion_earning = models.CurrencyField(initial=c(0))
    points_in_realized_price = models.PositiveIntegerField(initial=0)

    pred = models.TextField(doc = "Allocation of prediction points",
                            validators=[RegexValidator(
                                regex='\[(([0-9]+ ?,? ?)+\]')])
    score = models.TextField(doc = "Scores assuming price in bin",
                             validators=[RegexValidator(
                                 regex='\[(([0-9]+ ?,? ?)+\]')])
```

```
class Group(BaseGroup):
    # <built-in>
    subsession = models.ForeignKey(Subsession)
    # </built-in>

    realized_price = max([random.normalvariate(150,50), 0])

    def update_price(self):
        """Update the price, allowing for structural breaks"""
        raise NotImplementedError

    def calculate_elicitation_earnings(self):
        """Update the earnings from predictions in previous periods.

        The predicted earnings are calculated for each round within
        the current market and is calculated based on the player's
        predictions.

        """

        pred_field = "pred"
        score_field = "score"

        # Find the index of the bin with the realized price:
        price = self.realized_price
        pbins = Constants.pbins
        N = Constants.ybins_n
        realized_price_index = max(0, bisect_left(pbins, price)-1)

        ## Now iterate over each player in the group and update the
        ## variables for
        ## - How many points they allocated correctly
        ## -
        for p in self.get_players():
            preds = json.loads(getattr(p, pred_field))
            scores = json.loads(getattr(p, score_field))

            p.point_in_realized_price = preds[realized_price_index]
            p.elicitaion_earning = scores[realized_price_index]
```

6.2.2 The pages.py file

To insert the module into the actual experiment, the module must be added to the `pages.py` file. An example is included below.

```
    # -*- coding: utf-8 -*-
from __future__ import division
from . import models
from ._builtin import Page, WaitPage
from .models import Constants
from django.utils.translation import ugettext as _

import json

##* PREDICTION PAGES

class Prediction(Page):
    """Pages for making predictions.

    Participants will make predictions for all future periods.

    """
    form_model = models.Player

    def get_form_fields(self):
        """Return a list of prediction fields to modify

        Actually, it is not necessary to dynamically make this list...
        """
        fields = ["pred"
                  # ,"custom_A", "custom_B"
                 ]
        return(fields)

    def vars_for_template(self):
        variables = {
            "alpha"   : Constants.prediction_alpha
            "beta"    : Constants.prediction_beta
            "kappa"   : Constants.prediction_kappa
            "xmax"    : Constants.xmax
            "xbins_n" : Constants.xbins_n
            "ybins_n" : Constants.xbins_n}
        return(variables)

    def error_message(self, vals):
        """Make sure that predictions do not exceed 100%.

        Add check in case somebody is altering variables with the
        JavaScript console.

        """
        preds = {key: vals[key] for key in vals.keys()
                 if key.startswith("pred")}
        for pred in preds.values():
            p = sum(json.loads(pred))
            if p > Constants.ybins_n:
                return (_("You have assigned more than 100%!"))
            elif (p < Constants.ybins_n):
                return (_("Not all points have been assigned."))



class Results(Page):
    """This page shows the earnings in a single round of the market."""

    # timeout_seconds = 45

    def vars_for_template(self):
        m, r = Constants.interpret_round[self.subsession.round_number]
        change =  self.group.equilibrium_price * self.player.shares_change
        temp_vars = {
            "total_cash_earning": (self.player.dividend_earnings + change),
            "asset_trading": change}
        if r >= 3:
            temp_vars["timelimit"] = 30
        return (temp_vars)

class OneMoreRoundP(Page):
    """Ask if the player wants one more round"""

    def is_displayed(self):
        return(False)

class FinalResult(Page):
    """Show the results of all """

    def is_displayed(self):
        return(False)

page_sequence = [Prediction,
                 Results,
                 OneMoreRoundP,
                 FinalResult]
```

6.2.3 Display forecast modules on the pages

The HTML template may be added using a code similar to the following example. This adds the forecast element to a page while using the experiment's favorite parameters. If the page is saved as, for instance, `InsertForecast.html`, it can be "called" from other pages by issuing the following Django template command:

```
    {% include 'asset_simple/InsertForecast.html' %}~
```

```
{% load staticfiles otree_tags %}
{% load staticfiles %}
{% load i18n %}

<!-- Hidden fields to store predictions -->
{% for field in form %}
  {{ field.as_hidden }}
{% endfor %}

<script type="text/javascript" src="{% static 'asset_simple/d3.v3.min.js' %}"></script>
<link rel="stylesheet" href="{% static 'asset_simple/forecast_style.css' %}">

<script>
  var alpha = {{ alpha }};
  var beta = {{ beta }};
  var kappa = {{ kappa }};
  var xmax = {{ pmax }};
  var xbins_n = {{ xbins_n }};
  var ybins_n = {{ ybins_n }};
  var forecast_div_elm = "forecast_widgets_container";
</script>

<script type="text/javascript" src="{% static 'asset_simple/forecast.js' %}"></script>

<div id = "forecast_widgets_container">
</div>
<script>
  forecast_instance = add_forecast_widget("forecast");
</script>
```