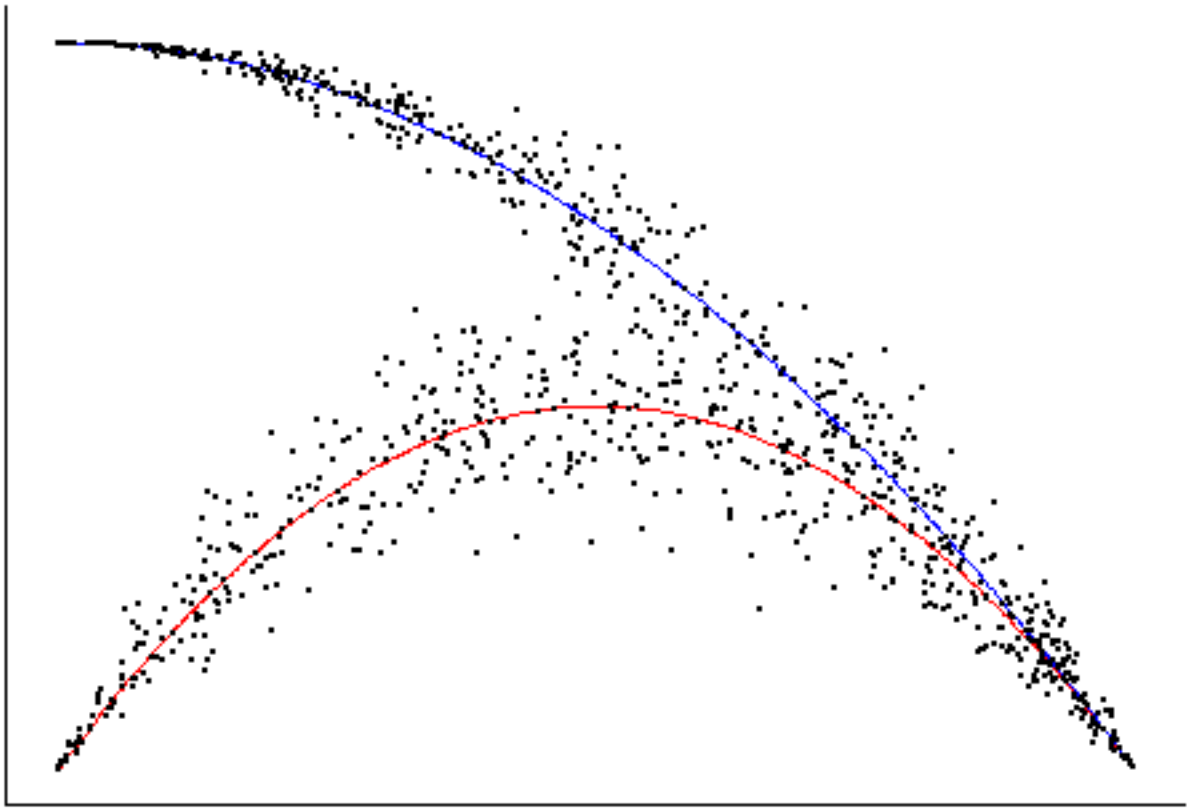
Structural Bayesian Techniques for Experimental and Behavioral Economics

With applications in *R* and *Stan*

James R. Bland

2025-10-16

Contents



Suggested citation:

> Bland, James R. 2025. "Structural Bayesian Techniques for Experimental and Behavioral Economics, with applications in *R* and *Stan*". https://jamesblandecon.github.io/StructuralBayesianTechniques/section.html, version date 2025-10-16

Preface

In 2010 I was invited to attended my first Economics seminar. I was an honors student at the time, and wasn't quite sure what I had got myself into. Glenn Harrison stood up the front of the room and presented some work on estimating models of discounting behavior. I don't know about you, but for me the idea that you can estimate the fundamental parameters of an economic model has always struck me as pretty amazing: we can go all the way from an *economic* model of how people might behave in an environment, to an *econometric* model of the data-generating process of our experiment. This is such a powerful tool, both because it provides us with a lens for evaluating our economic model when we get some data, but it also allows us to comment on and estimate the parameters and quantities that our model says are important.

So that's how I came across *structural*, but why *Bayesian* techniques? As much as I think we should learn from our experiments the way mathematics tells us we should (i.e. use Bayes' rule), I am primarily here because it does the things I want to do better than existing Frequentist techniques. By that I mean in experimental and behavioral economics we have to take heterogeneity seriously. Bayesian hierarchical modeling is a great way to deal with that. Furthermore, in structural applications we often want to comment on *transformations* of our parameters. These are handled really easily within the Bayesian framework. For me, Bayes is not the right tool if you just want to draw a straight line through your data. Then, I'm quite happy to run a linear regression because it's easy and fast. However the models we have in mind when designing and analyzing our experiments are rarely linear or homogeneous.

What I have put together here is a collection of notes on how I think about going from an economic model to an econometric model, how to estimate these econometric models, and how to communicate their results.

Throughout this book, my goal is to equip you with some tools that I have used or developed. This is not a book on econometric theory, although I will certainly mention it when we can take advantage of it. Instead, I want it to be a practical "how to" guide for anyone who wants to use these techniques. As such, there will be at least one example in each chapter, taking you from an economic model to an econometric model, and then showing you how to estimate it. Along the way, I will also show you some computational techniques that might be useful for solving some of the more persnickety economic models.

Setting the stage

1 Introduction

1.1 What does your theory say about your data?

If you are reading this book, I can assume that you are interested in estimating an economic model of how people behave in a particular environment. But what does it mean to "estimate" an *economic* model? When we talk of estimating models, we usually mean *econometric* models. For example we might estimate the coefficients β in a linear regression model:

$$Y_i = \beta_0 + \beta_1 X_i + \epsilon_i$$

From my perspective, this equation is a model for how (say) treatment conditions in an experiment X_i influence choices made by the experiment's participants Y_i. The model assumes a linear relationship between X and Y. The *parameters* of this model $(\beta_0, \beta_1)^\top$ describe the specific linear relationship that exists in the data, and our goal in linear regression is to estimate (and hopefully interpret) these parameters. While there are some special cases in which our theory will tell us that there should be a linear relationship between X and Y,[1] this framework is woefully limiting relative to the rich set of economic models we might want to take to our data. Instead, we need a framework for translating our theoretical ideas about how our participants might behave into *probabilistic predictions* about what the data these participants generate will look like.

[1] For example, for a private-value first-price auction with independent uniform values, the equilibrium bidding function will take the form $\text{bid}_i = \beta_0 + \beta_1 \text{value}_i$.

To fix ideas, suppose that in our experiment, we elicit participants' risk preferences using an investment task:

> You are endowed with 100 tokens, and can invest any amount of these tokens in a lottery. You have a chance of 1/2 (50%) to win $[X]$ times the tokens you bet on the lottery, and a 1/2 (50%) chance that these tokens are lost. (adapted from Bland (2019a))

We have an *economic model*, expected utility maximization, that makes predictions in this environment. Specifically, let y^* be the number of tokens that a participant chooses to invest, then they will choose y^* such that:

$$y^* = \arg \max_{y \in [0,100]} \{0.5u(100 - y) + 0.5u(100 - y + Xy)\}$$

where $u(x)$ is the participant's utility function over certain amounts of money. We further make the simplifying assumption that this utility function takes on the constant relative risk-aversion functional form:

$$u(x) = \frac{x^{1-r}}{1-r}$$

From here, we can solve the maximization problem using the first-order condition (please bear with me, I like to show my working):

$$\begin{aligned}
\max_{y \in [0,100]} & \left\{ 0.5\frac{(100 - y)^{1-r}}{1-r} + 0.5\frac{(100 - y + Xy)^{1-r}}{1-r} \right\} \\
0 &= -0.5(100 - y^*)^{-r} + 0.5(X - 1)(100 - y^* + Xy^*)^{-r} \\
(100 - y^*)^{-r} &= (X - 1)(100 + (X - 1)y^*)^{-r} \\
y^* &= \frac{100(1 - (X - 1)^{-1/r})}{(X - 1)^{1-1/r} + 1}
\end{aligned}$$

Which makes predictions that look like this:

```
library(tidyverse)
dplt1<-(
  expand.grid(X=seq(2.01,6,length=100),r = c(0.2,0.4,0.6,0.8))
  %>% mutate(y=100*(1-(X-1)^(-1/r))/((X-1)^(1-1/r)+1),
             r = paste("r =",r)
             )
)

(
  ggplot(dplt1,aes(x=X,y=y,color=r))
  +geom_path()
  +xlab("X")+ylab("y (optimal choice)")
  +theme_bw()

)
```

How would you go about estimating r? The relationship certainly isn't linear. The solution I will teach you in this book is to modify the model so that it makes *probabilistic* predictions, rather than the *deterministic* predictions made in the figure above. Firstly, this is useful because it means that we can write down a likelihood function. This permits us to estimate our model using the Bayesian techniques described in this book, or maximum likelihood techniques if that is your thing. If done right, this is also a *behaviorally plausible* extension of the model with deterministic choice: like ourselves, our experiments' participants are not infallible, and so will choose something different to our model's point prediction. Furthermore, and

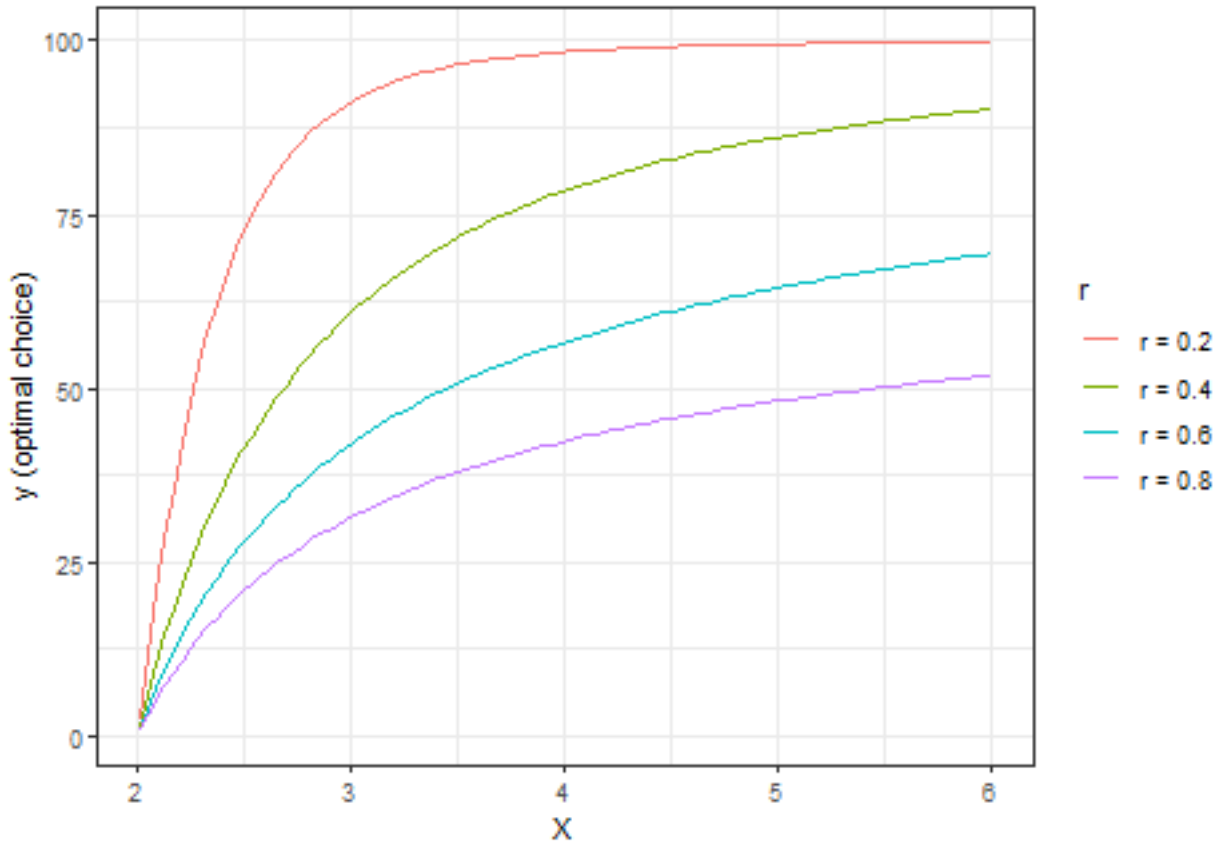


Figure 1: Predictions from a model of expected utility maximization. r is the parameter in the CRRA utility function $u(x) = \frac{x^{1-r}}{1-r}$

perhaps more importantly, it is heroic to assume that the deterministic model of choice we wrote down above is *exactly* what our participants are doing, so having a component of choice that is not explicitly accounted for in the model is probably a good idea.

For this example, a probabilistic extension to the model could involve adding an error term to the end of our prediction:

$$y = \frac{100(1-(X-1)^{-1/r})}{(X-1)^{1-1/r}+1} + \epsilon, \quad \epsilon \sim N(0,\sigma^2)$$

Or perhaps specifying a logistic choice rule, where choices that yield greater utility are more likely to be made than choices that yield less utility:

$$\Pr(y=k) = \frac{\exp\left(\lambda\left(0.5\frac{(100-k)^{1-r}}{1-r} + 0.5\frac{(100+(X-1)k)^{1-r}}{1-r}\right)\right)}{\sum_{j=1}^{100}\exp\left(\lambda\left(0.5\frac{(100-j)^{1-r}}{1-r} + 0.5\frac{(100+(X-1)j)^{1-r}}{1-r}\right)\right)}$$

For example, the predictions using the logistic choice rule with $\lambda = 10$ are now probability distributions:

```
r<-0.4
lambda<-10
dplt<-(
  expand.grid(X = c(2.25,2.75,3.5,5.5),y=seq(0,100,length=101),r=c(0.2,0.4,0.6,0.8))
  %>% mutate(EU = 0.5*(100-y)^(1-r)/(1-r)+0.5*(100+y*(X-1))^(1-r)/(1-r))
  %>% group_by(X,r)
  %>% mutate(pr = exp(lambda*(EU-max(EU)))/sum(exp(lambda*(EU-max(EU)))))
  %>% mutate(X=paste("X =",X),
             r=paste("r =",r))
)

(
  ggplot(dplt,aes(x=y,y=pr))
  +geom_path()
  +facet_grid(r~X,scales="free")
  +theme_bw()
  +xlab("Tokens invested")
  +ylab("probability")
)
```

An added benefit of having a probabilistic model of behavior, even if we do not intend to estimate it, is that we can simulate data for our experiment. This can be useful both for experiment design, and for evaluating the sampling properties of our estimators. For example, suppose we wanted to see what the choices of a participant in this experiment would look like if they had $r = 0.4$ and $\lambda = 10$. The process for simulating data is very similar to the process for constructing the probability distribution:

```
r<-0.4
lambda<-10
set.seed(42)
probDist<-(
  expand.grid(y=seq(0,100,length=101),
              X = seq(2.5,6,length=30)
              )
  %>% mutate(EU = 0.5*(100-y)^(1-r)/(1-r)+0.5*(100+y*(X-1))^(1-r)/(1-r))
  %>% group_by(X)
  %>% mutate(pr = exp(lambda*(EU-max(EU)))/sum(exp(lambda*(EU-max(EU)))))
```

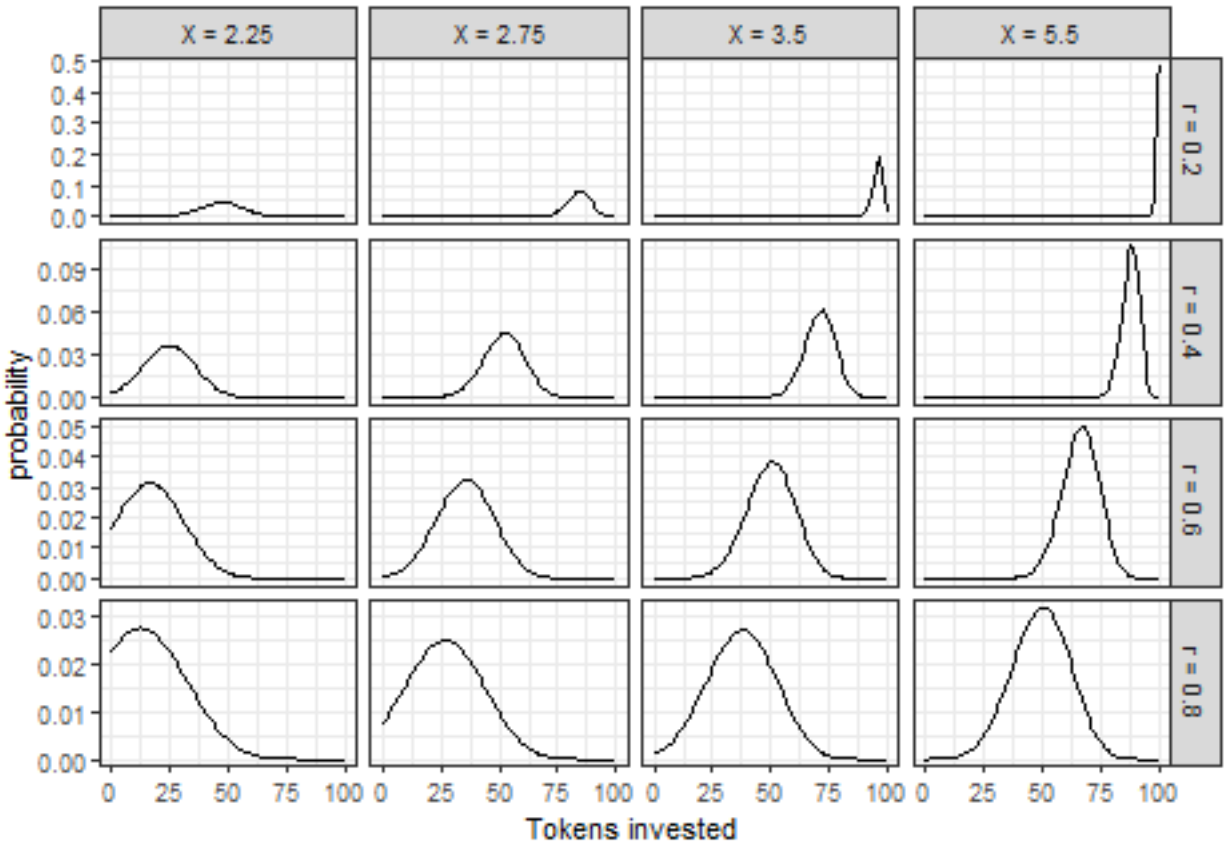


Figure 2: Probabilistic predictions for the investment task using a logistic choice rule

```
)

data<-tibble()

for (xx in (probDist$X %>% unique())) {
  d<-probDist %>% filter(X==xx)
  dataY<-sample(d$y,1,replace=TRUE,prob=d$pr)
  data<-rbind(data,
              tibble(y=dataY,X=xx))
}

(
  ggplot()
  +geom_point(data=data,aes(x=X,y=y))
  +geom_path(data=dplt1,aes(x=X,y=y,color=r))
  +xlab("X")+ylab("y (choice)")
  +theme_bw()
)
```

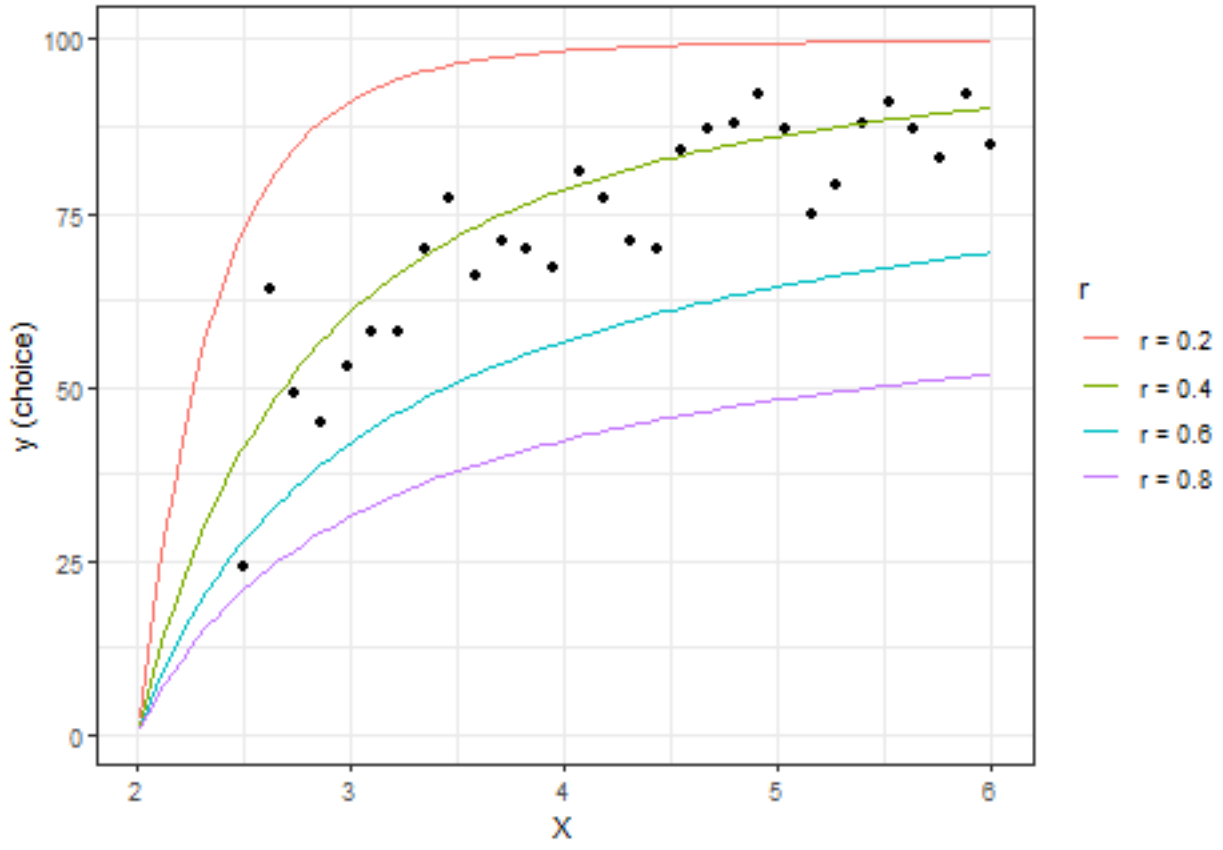


Figure 3: Some simulated data

This kind of simulation exercise is immensely useful. Graphically, we can see that if we want to estimate r using this model and experiment, we will be trying to find the curve in Figure 3 (or maybe one in between) that best matches the data. Our probabilistic model will do all of the work defining what "best" is in this case. From Figure 3, we can see that our model will likely do a good job at distinguishing between $r = 0.2$,

$r = 0.4$, and $r = 0.6$, but might have trouble if we want to resolve the uncertainty about r to a finer resolution. Additionally, now that we have a simulated dataset with known parameters, we can see how well our model estimates these parameters (I do this in the next section). Finally, these predictions help us to understand the implications of the choice precision parameter λ (e.g. how dispersed are data around their point predictions?), and identify treatment conditions (X) that are going to be more (large X) or less (small X) informative about r.

1.2 What do your data say about your theory?

From here, it is not too much of a leap to code the model in *Stan*. True, we need to have a good think about what are appropriate priors, and so on, but hopefully you can see that once we have a probabilistic model written down, we can code it up in a similar way to how we derived it. Specifically, note how we specified the expected utility variable `EU`, then calculated the logistic choice probabilities (`softmax`).

```
// Introduction_InvestmentTask.stan

/*
In the data block, we tell Stan how to read in the data we will be passing
it from R.
*/
data {
  int<lower=0> n; // number of observations
  int<lower=0> ny; // size of the choice set
  int y[n];   // choices made by participant
  vector[n] X;   // multiplier for investment
  vector[ny] ygrid; // possible values of y (i.e. the choice set)
  real prior_r[2]; // prior mean and sd for r
  real prior_lambda[2]; // prior mean and sd for log(lambda)
}

/*
In the prarameters block, we tell Stan what the model's parameters are,
and any restrictions we need to place on them. Here we need to restrict
lambda to be positive
*/

parameters {
   real r; // parameter in CRRA utility function x^(1-r)/(1-r)
   real<lower=0> lambda; // logit choice precision
}


/*
In the model block, we tell Stan how the data and priors contribute to the
posterior density
*/
model {

  /*
  In this loop, I go through each observation in the data, calculate the
  expected utility of each possible choice in the choice set, then use this to
  compute the implied probability distribution over actions for that observation.
  I then take the (log) probability of the action that was actually taken and
  add it to the likelihood
```

```
  */
  for (ii in 1:n) {
    vector[ny] EU; // expected utility of choosing each action
    vector[ny] lpr;// (log) probability of choosing each action
    EU = 0.5*pow(100-ygrid,1.0-r)/(1.0-r)+0.5*pow(100+ygrid*(X[ii]-1),1.0-r)/(1.0-r);

    lpr = log_softmax(lambda*EU);

    /*
    increment the log-likelihood
    Stan's "target" is in log probability units.
    */
    target+=lpr[y[ii]];
  }

  // specify the priors for the parameters
  r ~ normal(prior_r[1],prior_r[2]);
  lambda ~ lognormal(prior_lambda[1],prior_lambda[2]);
}
```

Finally, we go back to *R*, or any other programming language that will run *Stan*, and estimate the model. Here I am binning choices into increments of ten to make the computation faster.

```
library(rstan)
rstan_options(auto_write = TRUE)
options(mc.cores = parallel::detectCores()) # Use all the cores

# Get a list of data that Stan can understand.
# Note here that I am rounding the data to the nearest 10 tokens.
# This will speed up the computation a lot.
stanData<-list(y=round(data$y/10),
               X=data$X,
               n=data$y %>% length(),
               ygrid = (1:10)*10,ny=10,
               prior_r = c(0.5,0.25),
               prior_lambda = c(log(10),0.5))

# This if statement isn't needed, but is there so I don't have to run Stan
# every time I compile this book.
if (!file.exists("Outputs/Introduction/Introduction_InvestmentTask.rds")) {
  # Fit the model in Stan
  Fit<-stan("Code/Introduction/Introduction_InvestmentTask.stan",
            data=stanData,
            seed=42)
  # Save the fitted model so I can access it later without having to run
  # Stan again
  saveRDS(Fit,file = "Outputs/Introduction/Introduction_InvestmentTask.rds")
}
# Read in the saved fitted model
Fit<-readRDS("Outputs/Introduction/Introduction_InvestmentTask.rds")

# Display a summary of the estimates
summary(Fit)$summary %>% knitr::kable(caption="Summary of the estimated *Stan* model.")
```

Table 1: Summary of the estimated *Stan* model.

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	
r	0.4150626	0.0001717	0.0075883	0.4001603	0.4100248	0.4151369	0.4201586	0.429389	1
lambda	7.3300355	0.0306710	1.3916342	4.7949716	6.3154328	7.2493079	8.2539359	10.132982	2
lp___	-62.8776918	0.0257126	1.0161700	-65.6160771	-63.2451930	-62.5697220	-62.1778855	-61.900278	1

And that's it! We have taken a model all the way from deriving predictions, expanding it to make probabilistic predictions, and finally to estimating the parameters in the model.

1.3 What do your parameters say about other things?

Let's say that again: **we estimated the parameters in the model**!. We have a model that says that parameter r is important, and here is everything we learned from it in our (simulated) experiment:

```
(
  ggplot(tibble(r=extract(Fit)$r),aes(x=r))
  +geom_histogram()
  +theme_bw()
)
```

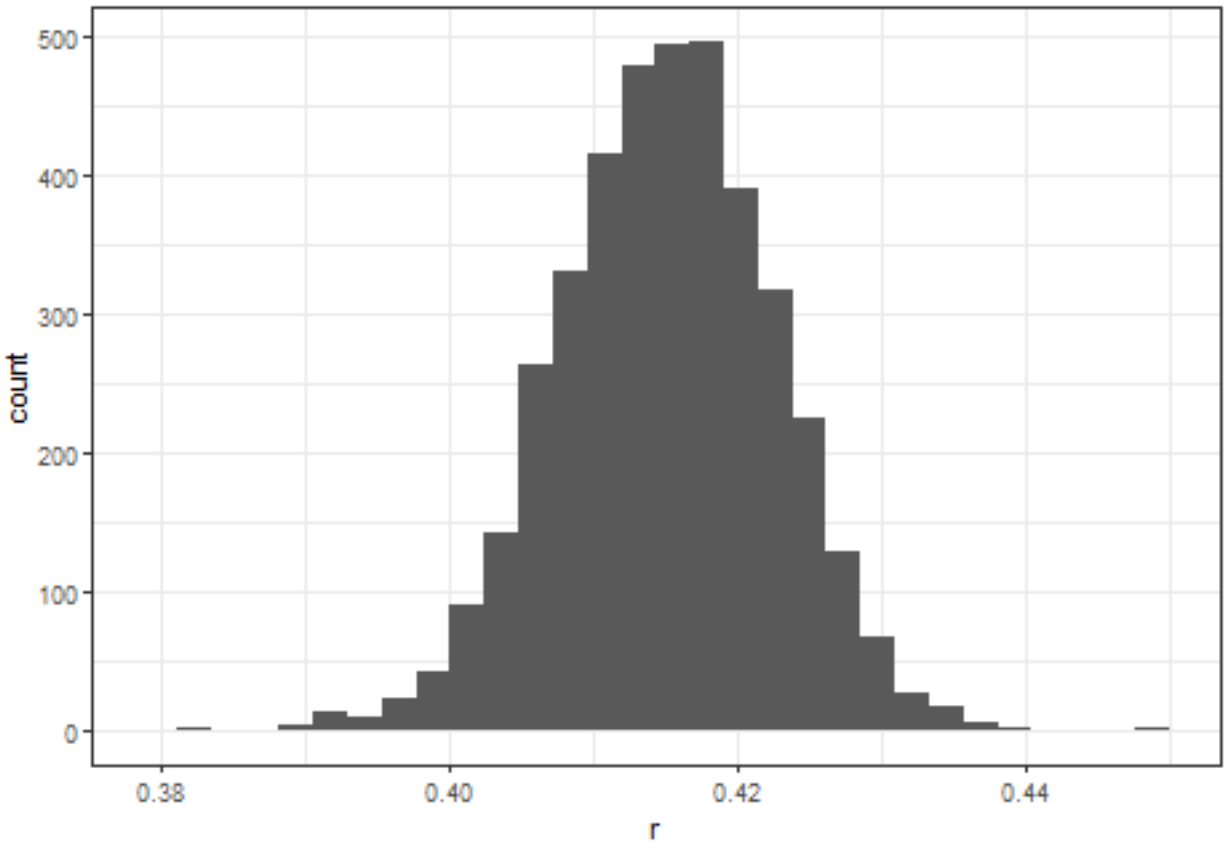


Figure 4: The posterior distribution of CRRA parameter r

But we do not have to stop here. In fact, the benefits of a structural model really start to be recognized here. Not only do we have estimates of our parameter, but because this is an *economic* model as well

as an econometric model, we can also use it to make out-of-sample predictions. That is, we can take the model beyond the confines of the experiment we used to estimate it, and derive its implications in other environments.

For example, suppose that we were designing another experiment where this participant could choose between the following two lotteries:

1. \$2.00 for sure
2. A 70% chance of \$2.50, and a 30% chance of \$1.00.

Our model tells us that they will prefer the risky Lottery 2 if and only if:

$$0.7\frac{2.5^{1-r}}{1-r}+0.3\frac{1^{1-r}}{1-r}\geq\frac{2^{1-r}}{1-r}$$

and since we have a posterior distribution of r, we can compute a posterior probability of this event being true:

```
r<-extract(Fit)$r
mean((0.7*2.5^(1-r)/(1-r)+0.3*1^(1-r)/(1-r)-2^(1-r)/(1-r))>=0)
```

```
## [1] 0
```

In other words, it is unlikely that this participant will prefer the risky lottery over the certain outcome.

Furthermore, one of the benefits of Bayesian estimation is that the posterior distribution of transformations of a model's parameters can be accessed by simply transforming the model's simulated posterior draws. That is, there are no pesky delta-method calculations to do if one wants to compute a nonlinear transformation of parameters. These show up in our models all the time. For example, we can use the model's estimates to calculate the certainty equivalent of this risky lottery (i.e. the sure amount that makes the decision-maker indifferent to the risky lottery). Mathematically, we calculate:

$$\begin{aligned}\frac{C^{1-r}}{1-r}&=0.7\frac{2.5^{1-r}}{1-r}+0.3\frac{1^{1-r}}{1-r}\\C&=\left(0.7\times2.5^{1-r}+0.3\times1^{1-r}\right)^{\frac{1}{1-r}}\end{aligned}$$

So our posterior distribution for C is:

```
d<-(tibble(r=r)
  %>% mutate(C = (0.7*2.5^(1-r)+0.3*1^(1-r))^(1/(1-r)))
)

(
  ggplot(data=d,aes(x=C))
  +geom_histogram()
  +theme_bw()
  +xlab("Certainty equivalent")
)
```

So we can be fairly sure that the certainty equivalent is about \$2.

1.4 What does your expertise say about your parameters?

Of course, before we do any of this, as experts in our field we will actually have a reasonably good idea about what might happen in our experiments. True, we are running an experiment because we don't know *exactly* what will happen. But our understanding of the environment, based on past observations, will surely guide us in designing the experiment, and Bayesian statistics permits this understanding to also guide us in analyzing our data.

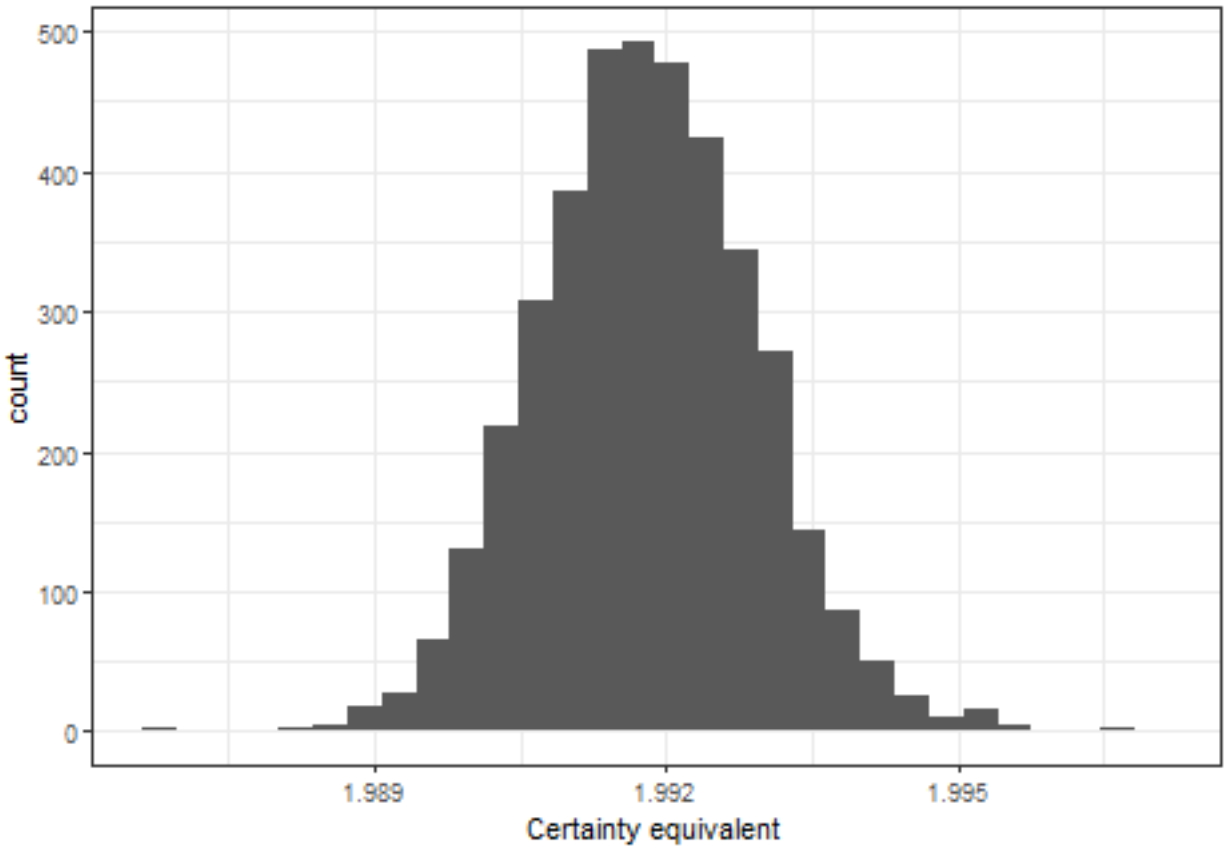


Figure 5: Posterior distribution of the certainty equivalent of the risky lottery

We also use a probabilistic model to express our beliefs about our parameters. These are called *priors*. For example, in the model I estimated above, I stated that my belief for the risk-aversion parameter r was:

$$r \sim N(0.5, 0.25^2)$$

This can be seen in the *R* code where I specify:

```
prior_r = c(0.5,0.25)
```

in the data I pass to *Stan*, and then in the *Stan* program where I specify the prior:

```
r ~ normal(prior_r[1],prior_r[2])
```

Our priors are going to serve (at least) two purposes. Firstly, in the design stage of an experiment, they are going to help us assess which experimental conditions are the most useful to take to our participants. Coupled with a likelihood, they also permit us to simulate data of our experiment that is consistent with our prior beliefs about what our data will look like. This is immensely beneficial even if we do not take a Bayesian model to our data: by having simulated data representative of our prior, we can adjust our experimental conditions so as to best answer our research questions. Furthermore, it provides us with an opportunity to test whether our econometric analysis will be well suited to analyzing the data that comes from our experiment.

Secondly, in the estimation stage of your research workflow, your prior will help your model look for solutions to its data-fitting exercise that are economically plausible. Maximum likelihood estimates of models of choice under risk, for example, are known to be difficult to solve for some participants (see for example Harrison and Ng (2016)) because multiple maxima exist. A prior will re-weight the problem so that more emphasis is placed on parameter values that are ex-ante plausible (see for example Gao, Harrison, and Tchernis (2022), which uses the same dataset as Harrison and Ng (2016)).

However priors can be a double-edged sword: while we want our model to estimate parameters that are economically plausible, we do not want our models to be so overly-informed by a prior that they hardly learn from the data. There is a gentle balancing act in choosing a suitable prior for our models that on the one hand incorporates our understanding of the problem as experts in the field, and on the other hand are not so informative that we might as well not collect data. The solution is *not* to try to be "as uninformative as possible", as this will open up another can of worms (e.g. Bland (2023b)). Instead, we should be methodical about asking not just what our priors say about their parameters, but also what our priors say about *data*, and quantities of interest that we might want to estimate and report. As such, much of this book will follow ideas laid out in Michael Betancourt's *Principled Bayesian Workflow* (Betancourt 2020). I believe that this process strikes a practical balance between priors being (i) something set in stone because they are exactly our beliefs about things, and (ii) something we must worry about at all costs because they will bias our estimates if they are too informative. As such, we will put our priors through the wringer before we collect our data, and this will be to our benefit. This scrutiny will provide us with a deeper understanding of our models and the kind of data we might get from our experiments.

2 Getting started in *Stan*

2.1 Installation in *R*

Here I am going to defer to people who have thought more about the process of installing the *RStan* library in *R*. Go here: https://github.com/stan-dev/rstan/wiki/RStan-Getting-Started.

At the time of writing (2023-0.1-27), you have the choice of installing the latest development version (2.26.x), or the latest released version (2.21.1, July 2020). I use the development version, as it has the capability of within-chain parallelization, which may speed things up if you have enough cores. I will strive to write code that does not rely on the development version, unless I am explicitly showing you how to parallelize something.

2.2 The anatomy of a *Stan* program

In this section, I will walk you through the different components of a *Stan* program. *Stan* takes instructions from you in an organized way, based on which part of the model you are describing:

- In the *data block*, you provide a description of the data
- In the *parameters block*, you list the parameters in the model
- In the *model block*, you describe how the data and parameters are combined to compute the likelihood. You also declare the priors in the model block

There is also the option to have the following blocks of code in a *Stan* program, although it is not necessary:

- In the *functions block*, you declare any user-defined functions
- In the *transformed data* block, you can transform data
- In the *transformed parameters* block, you can generate variables that are transformations of your parameters and data. As these transformed parameters are assumed to be needed in the model block, *Stan* has to compute derivatives with respect to these parameters.
- In the *generated quantities* block, you can ask *Stan* to compute anything using the simulated parameters from the posterior and the data. As these quantities do not show up in the model block, *Stan* does not need to compute derivatives of these.

As an example, I will walk you through writing a *Stan* program for analyzing data from Andreoni and Vesterlund (2001). You can load the data like this:

```
library(tidyverse)
AV2001choices<-read.csv("Data/AV2001choices.csv")
AV2001parameters<-read.csv("Data/AV2001parameters.csv")
```

In this experiment, participants were asked to allocate tokens between themselves and another participant. These tokens were then converted to cash at different rates. You can see how these budget sets were constructed using the `AV2001parameters` dataset:

```
AV2001parameters %>% knitr::kable()
```

X	income	pSelf	pOther
1	40	3	1
2	40	1	3
3	60	2	1
4	60	1	2
5	75	2	1
6	75	1	2
7	60	1	1
8	100	1	1

where `income` is the number of tokens to be allocated, and `pSelf` and `pOther` are the price of converting a token into one experimental currency unit for oneself and the other participant, respectively. The other part of the dataset includes the choices made by participants:

```
AV2001choices %>% head() %>% knitr::kable()
```

X	female	keep1	keep2	keep3	keep4	keep5	keep6	keep7	keep8
1	0	10	30	20	40	25	50	30	50
2	0	40	40	60	60	75	75	60	100
3	1	40	35	60	50	45	75	50	100
4	0	40	30	50	35	45	50	50	50
5	1	20	30	40	45	50	55	40	75
6	0	38	39	58	59	74	73	59	98

where `keep1` is the number of tokens kept by the participant in the first budget set, `keep2` for the second budget set, and so on. Each row corresponds to a participant. For example, the first participant in their first

decision decided to keep 10 tokens. This means that they would have earned 10/3 experimental currency units for themself, and $(40-10)/1$ experimental currency units for the other participant.

For this example, suppose that we would like to estimate how the price of giving, $p^{\text{Other}}/p^{\text{Self}}$,[2] affects whether the participant passes nothing to the other person at all. Specifically, let's estimate the logit model:

$$\text{kept everything} \mid \frac{p^{\text{Other}}}{p^{\text{Self}}} \sim \text{Bernoulli}\left(\Lambda\left(\alpha + \beta \frac{p^{\text{Other}}}{p^{\text{Self}}}\right)\right)$$

where $\Lambda(x) = 1/(1+\exp(-x))$ is the inverse logit transformation. That is, we wish to estimate the parameters α and β. We will use the following priors for the model's parameters, α and β:

$$\alpha \sim N(0, 0.25^2)$$
$$\beta \sim N(0, 0.25^2)$$

Also, since this is a nonlinear model, let's also calculate a prediction of the probability that somebody will keep everything when the price of giving is one:

$$\Pr\left(\text{kept anything} \mid \frac{p^{\text{Other}}}{p^{\text{Self}}} = 1\right) = \Lambda(\alpha + \beta)$$

2.2.1 Data block

First, we will provide a set of instructions for *Stan* to parse the data. In *RStan*, we first put our data into a named list format. This list must include not only the data, but also some information about the size of some variables we are passing *Stan*. We can also pass any other values, like our priors, into the data block. Here is what I came up with:

```
dStan<-list(
  # Information from the parameters data file
  income = AV2001parameters$income,
  pSelf  = AV2001parameters$pSelf,
  pOther  = AV2001parameters$pOther,

  # number of choices made by each participant
  nChoices = AV2001parameters$pOther |> length(),

  # information from the choices data file
  female = AV2001choices$female,
  keep = AV2001choices %>% dplyr::select(keep1:keep8) |> as.matrix(),
  n = AV2001choices$female %>% length(),

  # Specify the priors

  prior_alpha = c(0,0.25),
  prior_beta  = c(0,0.25)
)
```

Here are the instructions for *Stan* to read in these data:

```
data {
  int n; // number of participants
```

[2] That is, if I want the other participant to have one more experimental currency unit, I must pass them p^{Other} tokens, which means I have to give up $p^{\text{Other}}/p^{\text{Self}}$ experimental currency units for myself.

```
  int nChoices; // number of choices per participant

  vector[nChoices] income;
  vector[nChoices] pSelf;
  vector[nChoices] pOther;

  vector[n] female;
  matrix[n,nChoices] keep;

  real prior_alpha[2];
  real prior_beta[2];
}
```

Note that we need to give *Stan a lot* of information about our data. Specifically, we need to declare the type of each variable. When we pass our list of data to *Stan*, it will then check that the variable we have passed to *Stan* matches the type that it is expecting to see. For example, here is an error I generated while writing this program that told me that it expected the variable `female` to be a vector with 8 elements, when in fact it received a vector of 142 elements. This was an error in my *Stan* file: I had mistakenly written `vector[nChoices] female;` instead of `vector[n] female;`.

```
{,eval=F} Error in new_CppObject_xp(fields$.module, fields$.pointer, ...) :    Exception:
mismatch in dimension declared and found in context; processing stage=data initialization;
variable name=female; position=0; dims declared=(8); dims found=(142) (in 'string', line
10, column 2 to column 26)
```

Here is another error it gave me, when I had forgotton to add the prior variables to the list of data:

```
{,eval=F} Error in new_CppObject_xp(fields$.module, fields$.pointer, ...) :    Exception:
variable does not exist; processing stage=data initialization; variable name=prior_alpha;
base type=double (in 'string', line 13, column 2 to column 22)
```

As with other programming languages, it is useful to get to know how to read error messages, and I will attempt to show some of the common ones to you as we work our way through some examples.

2.2.2 Transformed data block

In the transformed data block, we can apply any function we like to the data we have loaded in the data block. For this application, note that we will need to transform our choice variable `kept` into our binary dependent variable, which I called `keptEverything`. I also took the vectors `pSelf` and `pOther` and generated the explanatory variable `priceOfGiving`. Here is how I did this in *Stan*:

```
transformed data {

  int keptEverything[n,nChoices];

  vector[nChoices] priceOfGiving;


  // generated a data variable equal to the price of giving
  // note that the . in ./ indicates an element-by-element operation
  priceOfGiving = pOther ./ pSelf;

  // Generate a data variable equal one if the participant kept everything
  // and zero otherwise
  matrix[n,nChoices] fractionKept;
  for (ii in 1:n)  {
```

```
    for (cc in 1:nChoices) {
      if (keep[ii,cc]==income[cc]) {
        keptEverything[ii,cc] = 1;
      } else {
        keptEverything[ii,cc] = 0;
      }

    }
  }

}
```

For those of you familiar with programming languages that permit matrix representations, you may be cringing at this point because of my double `for` loop. Yes, this will slow things down a bit compared to a matrix representation (as I did with calculating the price of giving), but I do this only to make the program easier to read. We will get to vectorizing operations soon enough. Either way, as this is the data block, so it will only execute once. You should worry more about for loops in blocks that include parameters, as these will execute thousands of times to simulate the posterior.

2.2.3 Parameters block

Next, we come to the parameters block. In this block of code, we list our model's parameters. Like the data block, we need to tell *Stan* the type of variable that our parameters are. Unlike the data block, though, these must all be continuous variables. Hence, you can a parameter as a `real`, `vector`, `matrix`, and so on, but you cannot have an `int` parameter.

```
parameters {

  // tell Stan what our parameters are
  real alpha;
  real beta;
}
```

2.2.4 Model block

This is where the guts of your model goes. Here you are going to tell *Stan* how to compute the likelihood, and also what your priors are. Fortunately for us, *Stan* has a whole lot of in-built probability distributions, so we can take advantage of these to make our code more readable. They are also faster to evaluate in some cases.

```
model {

  /*
  Increment the likelihood
  */
  for (ii in 1:n) {
    for (cc in 1:nChoices) {
      keptEverything[ii,cc] ~ bernoulli_logit(alpha+beta*priceOfGiving[cc]);
    }
  }

  // specify the priors
  alpha ~ normal(prior_alpha[1],prior_alpha[2]);
  beta  ~ normal(prior_beta[1],prior_beta[2]);
```

```
}
```

Note here that we are coding our model in almost exactly the same way as we stated it mathematically. In the first part of the block (the nasty double `for` loop that I will get rid of in later examples), we specify the distribution of our data `keptEverything`, and then below this (it doesn't really matter in which order we do this) we state the prior.

2.2.5 Generated quantities block

Finally, we can generate the prediction for the probability of keeping everything. This is done in a similar way to the generated data block, but *Stan* stores these values in the simulation output.

```
generated quantities {

  // prediction when the price of giving is one

  real predictionPrice1;

  predictionPrice1 = 1.0/(1.0+exp(-(alpha+beta)));

}
```

2.2.6 The final product

Putting everything together, this is what we have:

```
// GettingStartedAV2001.stan

data {
  int n; // number of participants
  int nChoices; // number of choices per participant

  vector[nChoices] income;
  vector[nChoices] pSelf;
  vector[nChoices] pOther;

  vector[n] female;
  matrix[n,nChoices] keep;

  real prior_alpha[2];
  real prior_beta[2];
}

transformed data {

  int keptEverything[n,nChoices];

  vector[nChoices] priceOfGiving;


  // generated a data variable equal to the price of giving
  // note that the . in ./ indicates an element-by-element operation
  priceOfGiving = pOther ./ pSelf;

  // Generate a data variable equal one if the participant kept everything
```

```
  // and zero otherwise
  matrix[n,nChoices] fractionKept;
  for (ii in 1:n)  {
    for (cc in 1:nChoices) {
      if (keep[ii,cc]==income[cc]) {
        keptEverything[ii,cc] = 1;
      } else {
        keptEverything[ii,cc] = 0;
      }

    }
  }

}

parameters {

  // tell Stan what our parameters are
  real alpha;
  real beta;
}

model {

  /*
  Increment the likelihood
  */
  for (ii in 1:n) {
    for (cc in 1:nChoices) {
      keptEverything[ii,cc] ~ bernoulli_logit(alpha+beta*priceOfGiving[cc]);
    }
  }

  // specify the priors
  alpha ~ normal(prior_alpha[1],prior_alpha[2]);
  beta  ~ normal(prior_beta[1],prior_beta[2]);


}

generated quantities {

  // prediction when the price of giving is one

  real predictionPrice1;

  predictionPrice1 = 1.0/(1.0+exp(-(alpha+beta)));

}
```

Note that the blocks have to appear in a particular order, otherwise *Stan* will throw an error. The other optional blocks not used in this example are the transformed parameters block, where we can store intermediate values that are used later for calculating the likelihood, and the functions block, where we can write our own user-generated functions. I will introduce these in more detail later.

2.3 Estimating a model

Once we have our *Stan* program, we can go back to *R* and estimate the thing. You do this by first loading the `rstan` library. I then set a couple of options. Here the `mc.cores` option tells *Stan* to use all the cores in my processor. As we are running more than one Monte Carlo chain, this will speed things up by almost a factor of the number of cores. I then select the `auto_write` option, which means that the compiled *Stan* file will be saved. If I use it again without editing it, we will not have to waste time compiling the model again.

```
library(rstan)
rstan_options(auto_write = TRUE)
options(mc.cores = parallel::detectCores())

# This if statement checks to see if I have already estimated this model, and
# only runs the program if the saved results are not there
if (!file.exists("Outputs/GettingStarted/GettingStartedAV2001.rds")) {
  Fit<-stan("Code/GettingStarted/GettingStartedAV2001.stan",
            data=dStan,seed=42)
  # Save the fitted model results
  saveRDS(Fit,file = "Outputs/GettingStarted/GettingStartedAV2001.rds")
}
# load the fitted model results
Fit<-readRDS("Outputs/GettingStarted/GettingStartedAV2001.rds")
```

2.4 Looking at the results

There are a few things we can do now that we have estimated our model. To begin with, we can get a summary table of the parameters and generated quantities like this:

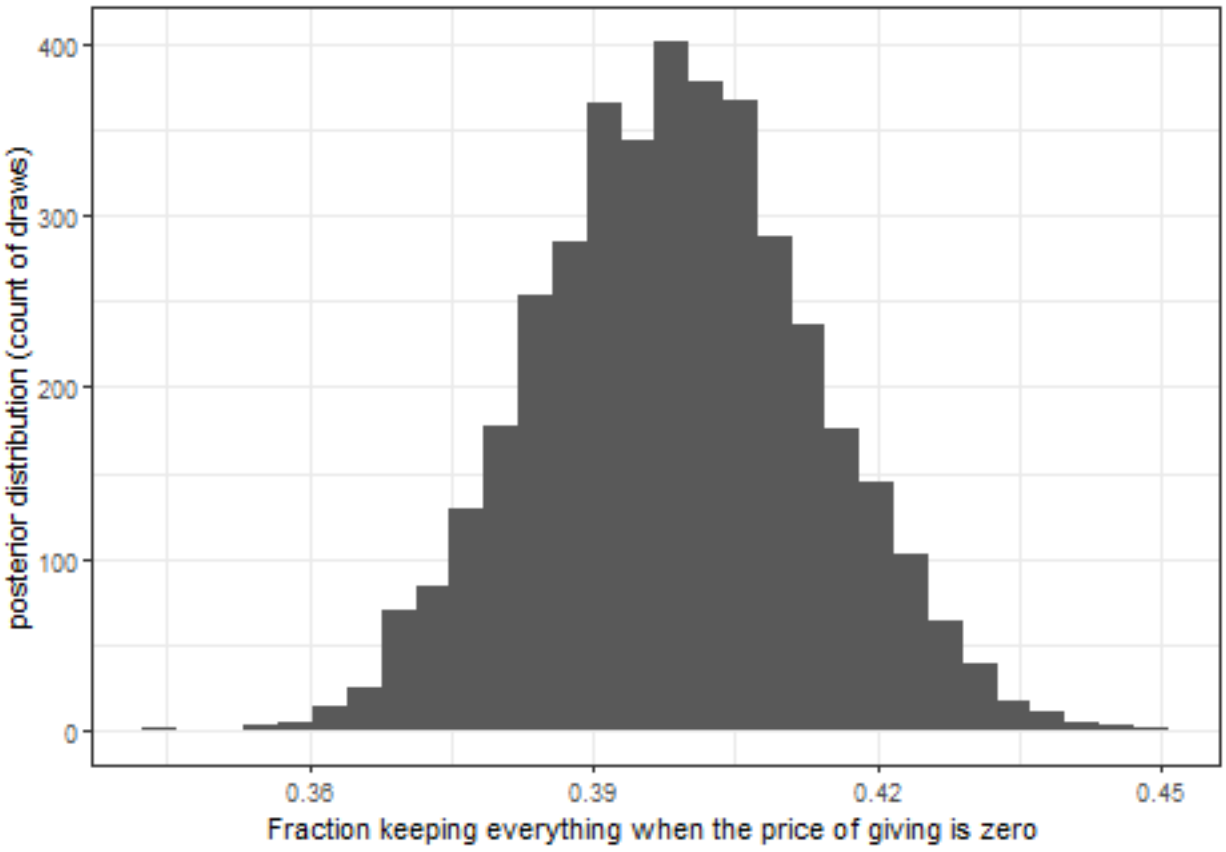
```
summary(Fit)$summary %>% knitr::kable()
```

	mean	se_mean	sd	2.5%	25%	50%	75%
alpha	0.0480923	0.0025922	0.0961676	-0.1442309	-0.0160319	0.0496628	0.1111312
beta	-0.4592476	0.0017745	0.0670033	-0.5930898	-0.5040354	-0.4585798	-0.4149885
predictionPrice1	0.3987266	0.0002827	0.0147165	0.3702469	0.3885788	0.3987644	0.4083643
lp___	-729.5571665	0.0257897	0.9609770	-732.0547891	-729.9745210	-729.2676189	-728.8477195

In particular, note the negative coefficient on β. We can interpret this as participants being less likely to give anything as the cost of giving increases. Good, that checks out.

Furthermore, we can access the draws from the posterior distribution using the `extract` function:

```
draws <-extract(Fit)

(
  ggplot(data=tibble(prediction = draws$predictionPrice1),aes(x=prediction))
  +geom_histogram()
  +theme_bw()
  +xlab("Fraction keeping everything when the price of giving is zero")
  +ylab("posterior distribution (count of draws)")
)
```



3 Probabilistic models of behavior

3.1 The problem with deterministic models

Our economic models typically assume some kind of optimal choice. This is a great place to start, as knowing what someone *should* do in your experiment if they (say) have a particular utility function is going to make some useful predictions and hopefully provide some testable implications. That is, if our model assumes a utility function $U(y, x; \theta)$ over choices y, experimental conditions x, given parameters θ, then our model makes predictions like this:

$$y = \arg\max_{\tilde{y} \in \mathbb{Y}} U(\tilde{y}, x; \theta)$$

The trouble we run into with modeling behavior, though, is (at least) twofold. Firstly, it is not *behaviorally plausible* that our participants will actually do this. Our participants are usually humans, not robots, and so even if we have correctly specified $U(y, x; \theta)$, they will make decisions with some noise. Additionally, it is somewhat heroic to assume that we have correctly specified $U(y, x; \theta)$, and so there is likely a component of utility that is known to the participant, but not by the econometrician. That is, it is likely that participants are actually maximizing some other function $V(y, x; \theta)$, and U is just our approximation of this.

Secondly, as we are using a likelihood-based technique,[3] any departures in the data from what can be rationalized by the deterministic model are going to set our likelihood to zero, meaning that we cannot simulate the posterior: a model of probabilistic choice is a necessity for Bayesian estimation.

[3]Here I say "likelihood-based", because the argument applies to any technique that uses a likelihood. Bayesian estimation and Maximum likelihood estimation both use a likelihood.

3.2 What *is* a probabilistic model?

Unlike a deterministic model, which prescribes an *action* in each choice set, a probabilistic model specifies a *probability distribution* over each choice set. This means that, given an action in the choice set $y \in \mathbb{Y}$ taken by a participant, and parameters θ of the model, we can assign a probability (mass or density) to that action. This means we can compute a likelihood function of our data, and it also means that given parameters θ, we can *simulate* data. Being able to compute a likelihood is a necessary condition for both Bayesian and maxiumum likelihood estimation. Being able to simulate data is also crucial for evaluating our experiment designs and the estimators we apply to our data.

Unlike Equation (3.1), where our model's predictions are nonrandom, instead we will specify the distribution of actions over the choice set. For example if the choice set $\mathbb{Y}$ is the real number line, a probabilistic model could look something like this:

$$y \mid x, \theta \sim N(x - \theta_1, \theta_2^2)$$

In words: "given treatment conditions x and parameters θ, y is normally distributed with mean $x - \theta_1$ and standard deviation θ_2."

An alternative way of writing this down would be to use a probability density function:

$$p(y \mid x, \theta) = \frac{1}{\sqrt{2\pi\theta_2^2}} \exp\left(-\frac{1}{2\theta_2^2}(y - x + \theta_1)^2\right)$$

For another example if the choice set is binary $\mathbb{Y} = \{0, 1\}$, instead of our model specifying either action $y = 0$ or $y = 0$, we specify the probability that each of these actions are taken. A probabilistic model in this case could look something like this:

$$y \mid x, \theta \sim \text{Bernoulli}\left(\Phi(x - \theta)\right)$$

where $\Phi(\cdot)$ is the standard normal cumulative density function. An alternative way of writing the same thing is to use the probability mass function:

$$p(y \mid x, \theta) = \begin{cases} \Phi(x - \theta) & \text{if } y = 1 \\ 1 - \Phi(x - \theta) & \text{if } y = 0 \\ 0 & \text{otherwise} \end{cases}$$

Fortunately for us, *Stan* is geared towards coding up models exactly like this. Once we have our probabilistic model written down, it is fairly straightforward to translate this into *Stan* code. That is, for this binary choice example, we could tell *Stan* that y is distributed according to the Bernoulli distribution as follows:

```
y ~ bernoulli(Phi(x-theta));
```

Alternatively, we can provide the same information to *Stan* by incrementing the likelihood. Here *Stan* actually deals with *log* densities, so we tell it to add the log-probability to its `target` variable:

```
target += y .* log(Phi(x-theta)) + (1.0-y) .* log(1.0-Phi(x-theta));
```

3.3 Example dataset and model

For this chapter, I will use data from Part 1 of Halevy, Persitz, and Zrill (2018). In this task, participants were given a budget of tokens to allocate between two assets. Each asset paid out in mutually exclusive states, which were equally likely to occur. We can see the budget sets presented to each participant in Figure

6. The dots in this Figure show the choices made by the first participant in the dataset, which is coded as `Subject=202`. I will use just the data from this participant for examples in this chapter.[4]

```
library(tidyverse)
library(readxl)
D<-read_excel("Data/Halevy et al (2016) - Data.xlsx",sheet="Budgetline Choices")

d <-D %>% filter(Subject==201)

(
  ggplot(d)
  +geom_abline(aes(slope=-`Y-intercept`/`X-intercept`,intercept = `Y-intercept`))
  +geom_point(aes(x=X,y=Y))
  +theme_bw()+coord_fixed()
  +xlim(c(0,max(d$`X-intercept`)))+ylim(c(0,max(d$`Y-intercept`)))
  +xlab("Prize if state 1 occurs (x)")+ylab("Prize if state 2 occurs (y)")
)
```

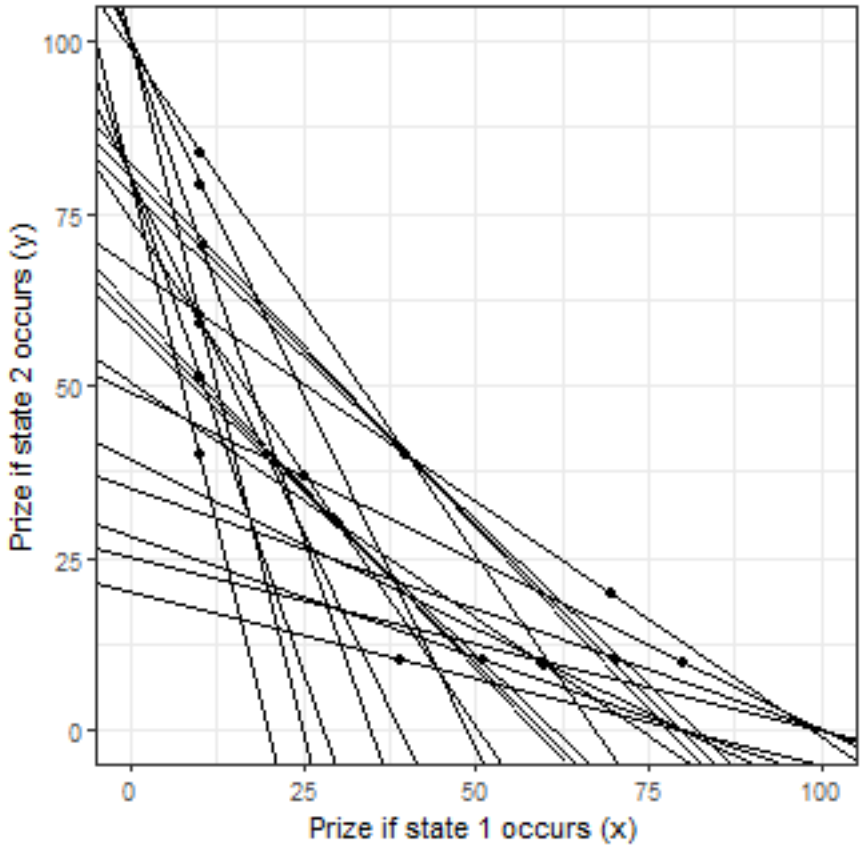


Figure 6: Budget sets and choices of one participant in Halevy, Persitz, and Zrill (2018)

Each budget set is characterized in th data by the y- and x-intercepts of the budget line, so if we denote these as $\bar{y}_t$ and $\bar{x}_t$ respectively for budget set t, then we can write the budget as:

[4]There are a few reasons for this. For one thing, for pedagogical reasons I want to keep the examples in this chapter as simple as possible. Going to more than one participant will complicate matters both computationally and conceptually. We will cover these issues later. Also, *Stan* returns some errors when I estimate these models for many of the other participants, and instead of giving attention to how to understand and address these errors here, I prefer to show you what you can do with a probabilistic model. We will get to understanding and fixing *Stan*'s error warnings later.

$$y = \bar{y}_t - x\frac{\bar{y}_t}{\bar{x}_t}$$

Let's suppose each participant is an expected utility maximizer with a constant relative risk aversion utility function over certain amounts of money:

$$u_i(x) = \frac{x^{1-r}}{1-r}$$

Then the participant's utility maximization problem is:

$$\max_{x,y\geq 0}\left\{0.5\frac{x^{1-r}}{1-r} + 0.5\frac{y^{1-r}}{1-r}\right\} \quad \text{s.t.: } y = \bar{y}_t - x\frac{\bar{y}_t}{\bar{x}_t}$$

We can set up a Lagrangian to solve for the optimal portfolio choice (again, forgive me, I like to show my working):

$$\mathcal{L} = 0.5\frac{x^{1-r}}{1-r} + 0.5\frac{y^{1-r}}{1-r} - \gamma\left(y_t - x\frac{\bar{y}_t}{\bar{x}_t} - y\right)$$

$$\begin{aligned} 0 &= 0.5x^{-r} + \gamma\frac{\bar{y}_t}{\bar{x}_t} \\ 0 &= 0.5y^{-r} + \gamma \end{aligned}$$

$$\begin{aligned} -2\gamma = x^{-r}\frac{\bar{x}_t}{\bar{y}_t} &= y^{-r} \\ \left(\frac{y}{x}\right)^{-r} &= \frac{\bar{x}_t}{\bar{y}_t} \\ \frac{y}{x} &= \left(\frac{\bar{y}_t}{\bar{x}_t}\right)^{\frac{1}{r}} \end{aligned}$$

Substituting this into the budget constraint yields:

$$\begin{aligned} x\left(\frac{\bar{y}_t}{\bar{x}_t}\right)^{\frac{1}{r}} &= \bar{y}_t - x\frac{\bar{y}_t}{\bar{x}_t} \\ x\left(\left(\frac{\bar{y}_t}{\bar{x}_t}\right)^{\frac{1}{r}} + \frac{\bar{y}_t}{\bar{x}_t}\right) &= \bar{y}_t \\ x &= \frac{\bar{y}_t}{\left(\frac{\bar{y}_t}{\bar{x}_t}\right)^{\frac{1}{r}} + \frac{\bar{y}_t}{\bar{x}_t}} \\ y &= \frac{\bar{y}_t\left(\frac{\bar{y}_t}{\bar{x}_t}\right)^{\frac{1}{r}}}{\left(\frac{\bar{y}_t}{\bar{x}_t}\right)^{\frac{1}{r}} + \frac{\bar{y}_t}{\bar{x}_t}} \end{aligned}$$

So our deterministic model of behavior is:

$$x^*(\bar{x}_t, \bar{y}_t) = \frac{\bar{y}_t}{\left(\frac{\bar{y}_t}{\bar{x}_t}\right)^{\frac{1}{r}} + \frac{\bar{y}_t}{\bar{x}_t}}$$

$$y^*(\bar{x}_t, \bar{y}_t) = \frac{\bar{y}_t \left(\frac{\bar{y}_t}{\bar{x}_t}\right)^{\frac{1}{r}}}{\left(\frac{\bar{y}_t}{\bar{x}_t}\right)^{\frac{1}{r}} + \frac{\bar{y}_t}{\bar{x}_t}}$$

3.4 Optimal choice plus an error

One common approach to turning a deterministic model into a probabilistic model, especially in these convex budget set experiments, is to add a mean-zero error term to these deterministic predictions. That is, *on average* the participant chooses (x^*, y^*), but then each decision is implemented with an error. A simple specification for this is to assume that the error term is normally distributed:

$$x \mid x^* \sim N(x^*, \sigma^2)$$

If we were using Frequentist techniques, this is where we might go ahead and estimate the model using nonlinear least squares, as is done in Andreoni and Vesterlund (2001), Andreoni and Miller (2002), and for one of the estimators under consideration in Halevy, Persitz, and Zrill (2018). For the Bayesian application, and for the maximum likelihood application for that matter, we need to be careful about a few things here.[5]

Firstly, the choice of whether to use x or y as our dependent variable is not inconsequential. To see this, note that since most of the lines in Figure 6 do not have a slope of negative one, the probabilistic predictions will be different between these two specifications. That is, if we choose x as our explanatory variable, then it must be that:

$$y \mid y^* \sim N\left(y^*, \left(\frac{\bar{y}_t}{\bar{x}_t}\right)^2 \sigma^2\right)$$

So the variance of y is not the same as the variance of x. Perhaps more alarmingly, the variance of y is a function of the slope of the budget set, but the variance of x is constant, so choices of y will be more or less precise as these budget lines shift. Thus, homoskedasticity in x implies heteroskedastcity in y. This specification therefore suffers from a *labeling problem*, and we will likely end up with different estimates depending on whether we choose x or y as our explanatory variable.

Another modeling issue to consider is whether the distribution $x \mid x^*$ is *truncated* or *censored* at the endpoints of the choice set. For example, what happens to our prediction if $x^* = 3$ but our error term implies that $x = -5$? Such a realization is possible with a normal distribution, but it is a choice that the experiment would have prevented the participant from choosing. For example, let's simulate some draws assuming $x \sim N(1, 2)$, and consider the implications of getting some draws with $x < 0$, which the experiment would not allow a participant to choose:

```
dx <- tibble(x=rnorm(10000,mean=1,sd=sqrt(2)))
dsim<-rbind(
  dx %>% mutate(type = "Normal"),
  dx %>% filter(x>=0) %>% mutate(type="truncated"),
  dx %>% mutate(x=ifelse(x<0,0,x)) %>% mutate(type="censored")
)
(
  ggplot(dsim,aes(x=x))
```

[5] I am *not* saying that nonlinear least squares avoids these problems, just that the problems become more apparent when one is dealing directly with a likelihood.

```
  +geom_histogram()
  +facet_wrap(~type)
  +theme_bw()
)
```

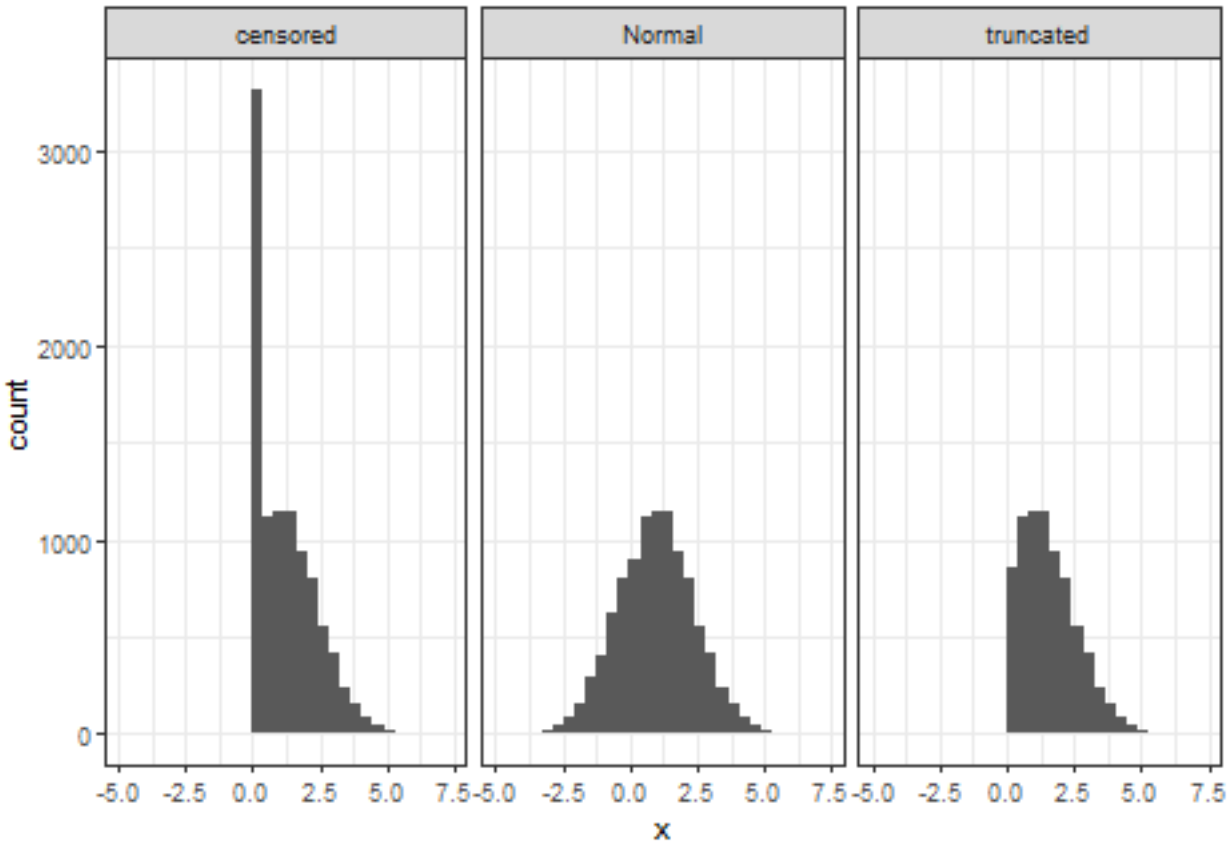


Figure 7: Examples of censored and truncated data.

Figure 7 illustrates the difference between censoring and truncation. Firstly, the un-censored, un-truncated simulated data is shown in the middle panel. Clearly these could not be choices from a convex budget set problem, because the participant would sometimes choose a negative amount of x. The left panel shows what happens when the data are censored at zero. All observations with $x < 0$ are coded as $x = 0$, and we can see that this is happening because there is a spike in the distribution at $x = 0$. Truncation, on the other hand, is shown in the rightmost panel. Here we see no spike because instead of coding all negative numbers as zeros, they are simply thrown out. The choice of which specification to go with here should be specific to the application. For example, in experiments investigating other-regarding preferences such as Andreoni and Vesterlund (2001) and Andreoni and Miller (2002), it is likely that there will be a substantial fraction of selfish participants who pass no money to their partner. In this case a spike of data at zero makes a lot of sense. For the Halevy, Persitz, and Zrill (2018) experiment, on this other hand, choosing $x = 0$ means that the participant wants to earn nothing in one of the two, equally likely, states of the world. This seemed unlikely to me, but we can look at the data to see how uncommon this is:

```
tab<-(D
  |> group_by(`X-intercept`,`Y-intercept`)
  |> summarize(`fraction choosing zero` = mean(X==0 | Y==0))

)
```

Table 2: Fraction of participants choosing either $x = 0$ or $y = 0$ in each of the 22 budget sets in @Halevy2018.

X-intercept	Y-intercept	fraction choosing zero
20.00	80.00	0.2898551
25.00	100.00	0.3285024
27.94	80.00	0.2463768
34.92	100.00	0.3043478
39.03	80.00	0.1739130
48.79	100.00	0.2222222
50.45	74.00	0.0821256
58.50	61.57	0.0144928
60.00	60.00	0.0193237
61.57	58.50	0.0193237
67.26	98.67	0.1159420
74.00	50.45	0.0869565
78.00	82.10	0.0193237
80.00	20.00	0.2995169
80.00	27.94	0.2318841
80.00	39.03	0.1739130
80.00	80.00	0.0144928
82.10	78.00	0.0241546
98.67	67.26	0.1062802
100.00	25.00	0.2946860
100.00	34.92	0.2415459
100.00	48.79	0.1932367

```
tab|> knitr::kable(caption="Fraction of participants choosing either $x=0$ or $y=0$ in each of the 22 bu
```

Table 2 proves me wrong! There are in fact a substantial minority of participants in some budgets sets who choose zero. I will take this onboard with the modeling presented below.[6] In particular, I will implement the censoring specification, rather than the truncation specification.

3.4.1 Estimating a model

For the purposes of this example, I will make some simplifying assumptions. These are mainly to speed up computation. Halevy, Persitz, and Zrill (2018) allow their participants to choose in increments of 0.2 tokens, so if I was being truly faithful to the data-generating process, I would be modeling the choice set as a grid with resolution 0.2. In the interest of making the task less strenuous for my laptop, I normalize the budget sets by dividing by the x intercept, so the choice of x must be on the unit interval. I then round the normalized x choices to the nearest 0.05 units.[7] The information I will pass to *Stan* will be the normalized y-intercept, and the range that the normalized choice fell into:

```
gridsize<-0.05

d<-(
  d
  |> mutate(x= X/`X-intercept`,
            yint = `Y-intercept`/`X-intercept`,
```

[6] I acknowledge that I am being a bad Bayesian in that I have looked at the data before estimating my model here. It probably would have been better to estimate both models (truncation and censoring), and then assess which one is matching the data better.

[7] This is another kind of censoring: I know that the actual choice falls within a range, but I do not know where in the range the actual choice was.

```
            xgrid = floor(x/gridsize),
            # lower and upper bounds for the censored data
            xlb = xgrid*gridsize,
            xub = (xgrid+1)*gridsize,

            # This ensures that the lower and upper bounds at the endpoints of
            # the grid are (-999,0.05) and (0.95,999)
            xlb = ifelse(x==0,-999,xlb),
            xlb = ifelse(x==1,1-gridsize,xlb),
            xub = ifelse(x==1,999,xub)
            )

)
```

That is, if a participant chose, say, a (normalized) $x = 0.23$, I will use the information that $0.20 < x < 0.25$. That is, `xlb=0.20` and `xub=0.25` The probability of a choice x being in this region (x_{lb}, x_{ub}) is then:

$$\Pr\left(x_{lb} < x < x_{ub}\right) = \Phi\left(\frac{x_{ub} - x^*}{\sigma}\right) - \Phi\left(\frac{x_{lb} - x^*}{\sigma}\right)$$

where $\Phi(\cdot)$ is the standard normal cumulative density function. For decisions that were at the endpoints of the choice set, I set $x_{lb} = -999$ if $x = 0$, and $x_{ub} = 999$ if $x = 1$.

Here is the *Stan* file I use:

```
data {
  int<lower=0> n; // number of observations
  real xlb[n]; // lower bound for x choice
  real xub[n]; // upper bound for x choice
  real yint[n]; // y intercept

  real prior_r[2];
  real prior_sigma;
}


parameters {
  real r;
  real<lower=0> sigma;
}

model {

  real xstar[n];
  real px[n];

  for (ii in 1:n) {
    // optimal choice given r
    xstar[ii] = yint[ii]/ (pow(yint[ii],1.0/r)+yint[ii]);
    // probability of the optimal choice + error landing where it did
    px[ii] = normal_cdf(xub[ii],xstar[ii],sigma)-normal_cdf(xlb[ii],xstar[ii],sigma);
  }
```

Table 3: Posterior estimates of the 'optimal choice plus error' model.

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	
r	0.6356843	0.0016724	0.0740045	0.5014449	0.5854661	0.6313009	0.6819358	0.7911945	19
sigma	0.1429142	0.0004520	0.0226142	0.1066118	0.1265328	0.1403869	0.1564103	0.1948155	25
lp__	-62.5379703	0.0278744	1.0820914	-65.2606211	-62.9812424	-62.2161067	-61.7512873	-61.4540023	15

```
  // increment the likelihood
  target +=log(px);

  // priors
  r ~ normal(prior_r[1],prior_r[2]);
  sigma ~ cauchy(0.0,prior_sigma);


}
```

Here's how we estimate it for the one participant:

```
library(rstan)
rstan_options(auto_write = TRUE)
options(mc.cores = parallel::detectCores())

dStan<-list(
  n = dim(d)[1],
  xlb = d$xlb,
  xub = d$xub,
  yint = d$yint,

  prior_r = c(0.5,0.25),
  prior_sigma = 0.008
)

if (!file.exists("Outputs/ProbabilisticModels/ProbMods_HPZ2018_normal.rds")) {
  Fit<-stan("Code/ProbabilisticModels/ProbMods_HPZ2018_normal.stan",
            data=dStan,seed=42)
  # Save the fitted model results
  saveRDS(Fit,file = "Outputs/ProbabilisticModels/ProbMods_HPZ2018_normal.rds")
}
# load the fitted model results
FitNormal<-readRDS("Outputs/ProbabilisticModels/ProbMods_HPZ2018_normal.rds")


summary(FitNormal)$summary |> knitr::kable(caption="Posterior estimates  of the 'optimal choice plus er
```

The estimates in Table 3 seem plausible based on what we know about participants in economic experiments. $r \approx 0.63$ is right in the meaty part of the distribution of typical levels of risk aversion for experiment participants, and $\sigma \approx 0.14$ means that choices are not so noisy that they always get censored at zero or one.

3.5 Utility-based models

Another approach for going from a deterministic to a probabilistic model is to have the utility function determine the entire probability distribution of our data. Note that for the "optimal choice plus error" model,

the only place that the utility function came into the equation was in determining where the distribution of choices was centered. To show you why this might be a problem, let's look a bit deeper into what the participant's utility maximization problem looks like for one decision in Halevy, Persitz, and Zrill (2018). In particular, here is what their utility of choosing an allocation $(x, (1-x)\bar{y}/\bar{x})$ looks like for the normalized budget set when $\bar{y}/\bar{x} = 4$:

```
ybar<-4
r<-0.5
d<-(
  tibble(x=seq(0.001,0.999,length=101))
  |> mutate(utility = 0.5*x^(1-r)/(1-r)+0.5*((1-x)*ybar)^(1-r)/(1-r)
            )
)
optimal<-d$x[which.max(d$utility)]
(
  ggplot(d)
  +geom_path(aes(x=x,y=utility))
  +geom_vline(xintercept=optimal,color="red")
  +theme_bw()
  +ylab("Expected utility")
)
```

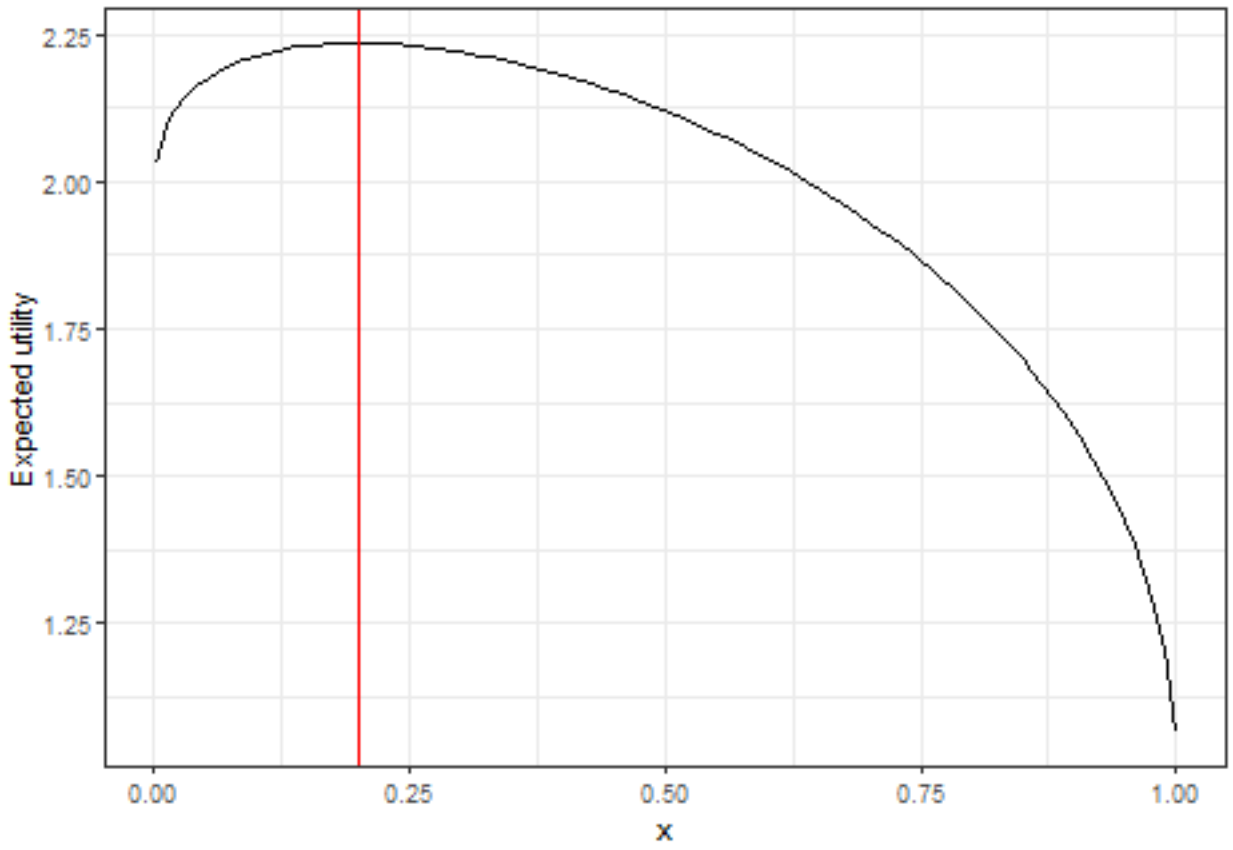


Figure 8: The expected utility function in Halevy, Persitz, and Zrill (2018) for a participant with $r = 0.5$ when $\bar{y}/\bar{x} = 4$. The vertical red line denotes the optimal choice

One important feature of Figure 8 is that the expected utility function is not symmetric: there is a much

sharper drop-off in utility moving to the left of the optimal choice (red line) than to the right. Yet the "optimal choice plus error" model will assume a symmetric distribution centered on the red line (outside of any censoring or truncation). This is because that model does not take into account the *shape* of this plot, only the maximum.

This goes against a lot of our understanding of how actual humans make decisions. In particular, they are more likely to take actions that yield higher utility than those that yield lower utility. In order for our model to reflect this, then loosely speaking, what we want for our probabilistic model to look something like this:

$$p\left(y \mid x, \theta\right) = g\left(U(y, x; \theta)\right)$$

with $g'(u) > 0$. That is, the probability of choosing an action is increasing in the utility associated with taking that action.

By far the most common implementation of this is the logistic choice rule, which mathematically looks like this:

$$p(y \mid x, \theta) = \frac{\exp\left(\lambda U(y, x; \theta)\right)}{\sum_{z \in \mathbb{Y}} \exp\left(\lambda U(z, x; \theta)\right)}$$

where $\lambda > 0$ measures a participant's *choice precision*: as λ increases, the participant is more sensitive to payoffs. For this decision in Halevy, Persitz, and Zrill (2018), the probabilistic predictions from logit choice look like this:[8]

```
LAMBDA<-c(1,10,30,40)
dlogit<-tibble()

for (ll in LAMBDA) {
  tmp<-(d
    |> mutate(
               lu = ll*utility,
               px = exp(lu-max(lu))/sum(exp(lu-max(lu))),
               lambda = paste("\u03bb =",ll)
               )
  )
  dlogit<-rbind(dlogit,tmp)
}

(
  ggplot(dlogit,aes(x=x,y=px,linetype=lambda))
  +geom_path()
  +geom_vline(xintercept=optimal,color="red")
  +ylab("probability")
  +theme_bw()
)
```

In particular, the distributions in Figure 9 are *not* symmetric: they have a longer right tail than left tail. This is because making "mistakes" by choosing slightly to the right of the optimal choice (red line) are less costly than choosing slightly to the left.

[8]Note that when I computed the logit choice probabilities in Figure 9, I first subtracted the maximum utility. This does not mathematically change the answer, but computationally is much more stable because we avoid exponentiating large positive numbers.

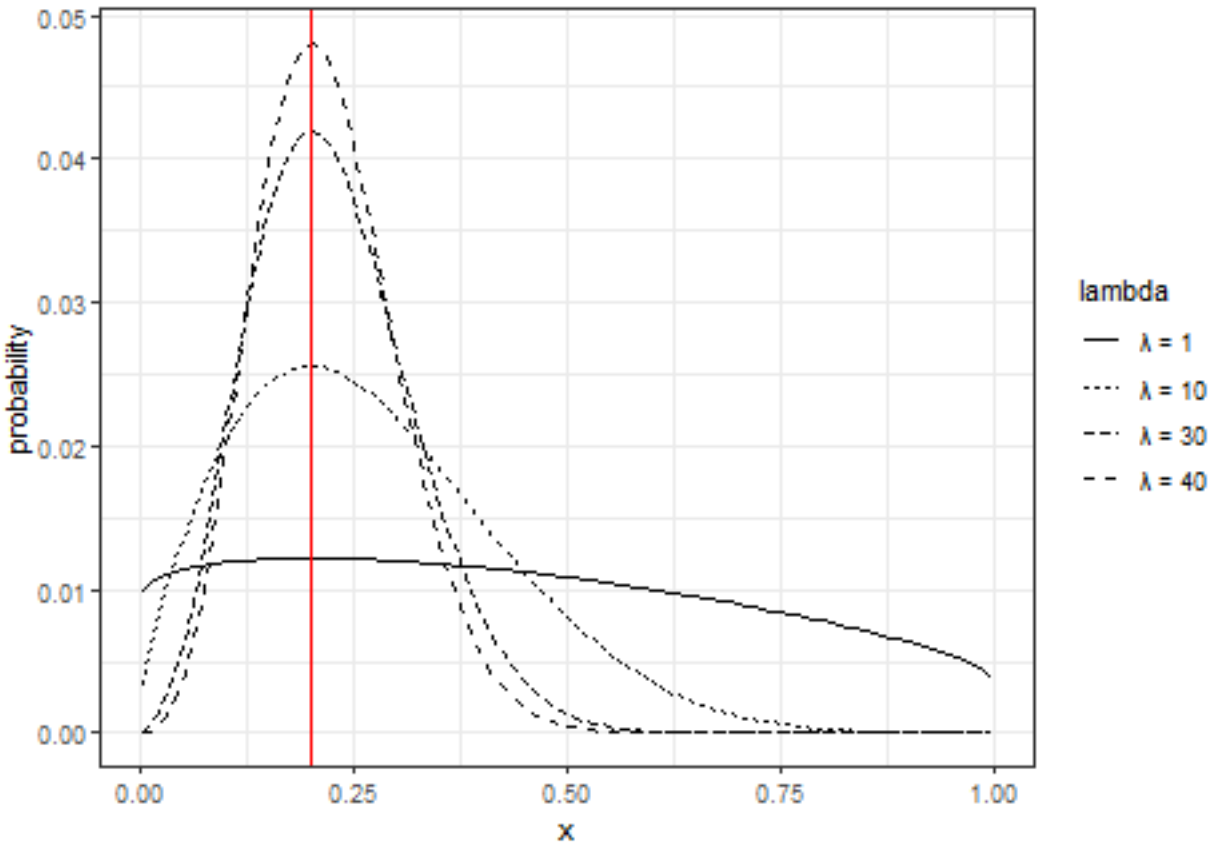


Figure 9: Logistic choice in Halevy, Persitz, and Zrill (2018) for a participant with $r = 0.5$ when $\bar{y}/\bar{x} = 4$. The vertical red line denotes the optimal choice

3.5.1 Estimating a model

Here is the *Stan* file for estimating this logistic choice model. As with the previous model, I need to discretize the choice set. For this implementation, instead of just computing a point prediction and adding an error, I need to compute utility for every possible choice that could be made. This is done over the grid veriable `xgrid`, and the element of this grid that most closely matches the participant's choice is `xi`.

```
data {
  int<lower=0> n; // number of observations
  int gridsize;
  vector[gridsize] xgrid; // grid to discretize choice of x over
  int xi[n]; // index for the grid for choice of x

  real yint[n]; // y intercept

  real prior_r[2];
  real prior_lambda;
}


parameters {
  real r;
  real<lower=0> lambda;
}

model {



  for (ii in 1:n) {
    vector[gridsize] EU;
    vector[gridsize] probX;

    // expected utility of choosing each x in the grid
    EU = 0.5*pow(xgrid,1.0-r)/(1.0-r)+0.5*pow((1.0-xgrid)*yint[ii],1.0-r)/(1.0-r);

    // logit choice probabilities
    probX = log_softmax(lambda*EU);

    // increment the likelihood by the log probability chosen
    target += probX[xi[ii]];

  }

  // priors
  r ~ normal(prior_r[1],prior_r[2]);
  lambda ~ cauchy(0.0,prior_lambda);


}
```

And here is how I pass the data to *Stan* to estimate the model in *R*:

```
gridsize<-20
Xgrid <- seq(1/(2*gridsize),1-1/(2*gridsize),by=1/gridsize)
```

Table 4: Posterior estimates of the logit choice model.

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	
r	0.6180463	0.0022409	0.0823109	0.479191	0.5672429	0.6121625	0.6612087	0.7932096	1
lambda	19.4787417	0.2092722	8.1674884	5.569901	13.7510580	18.7127683	24.4728880	37.7761375	1
lp__	-61.6231103	0.0584521	1.3736571	-65.357691	-62.0096403	-61.1958480	-60.7450303	-60.4548983	

```
d<-(
  d
  %>% rowwise()
  %>% mutate(xi = which.min(abs(x-Xgrid)))
)

dStan<-list(
  n = dim(d)[1],
  xgrid = Xgrid,
  gridsize=gridsize,
  xi=d$xi,
  yint = d$yint,

  prior_r = c(0.5,0.25),
  prior_lambda = 0.08
)
```

```
## Warning: Unknown or uninitialised column: `yint`.
```

```
if (!file.exists("Outputs/ProbabilisticModels/ProbMods_HPZ2018_logit.rds")) {
  Fit<-stan("Code/ProbabilisticModels/ProbMods_HPZ2018_logit.stan",
            data=dStan,seed=42)
  # Save the fitted model results
  saveRDS(Fit,file = "Outputs/ProbabilisticModels/ProbMods_HPZ2018_logit.rds")
}
# load the fitted model results
FitLogit<-readRDS("Outputs/ProbabilisticModels/ProbMods_HPZ2018_logit.rds")


summary(FitLogit)$summary |> knitr::kable(caption="Posterior estimates of the logit choice model.")
```

The posterior estimates are summarized in Table 4. While we cannot directly compare the parameters λ and σ, we can compare the estimates of the risk-aversion parameter r. These are both similar in terms of the posterior mean and the posterior standard deviation.

3.5.2 Doing something with the estimates

While having and estimate of r and λ might be useful to your research question on their own, note that because they are parameters of a probabilistic choice model, they can be used to make predictions and derive implications. To fix ideas, let's just focus on the first decision that this participant made. For this decision, the normalized y intercept of the budget line was `yint=0.3492`, and they chose to invest a normalized amount of `x=0.702` in the asset that pays out in State 1. Let's begin by extracting our estimated model's posterior draws, and then we will use them to derive some predictions.

```
postdraws<-tibble(r=extract(FitLogit)$r,lambda=extract(FitLogit)$lambda)

(
```

```
  ggplot(postdraws,aes(x=r,y=lambda))
  +geom_point(size=0.3)
  +theme_bw()
  +ylab("\u03bb")
)
```

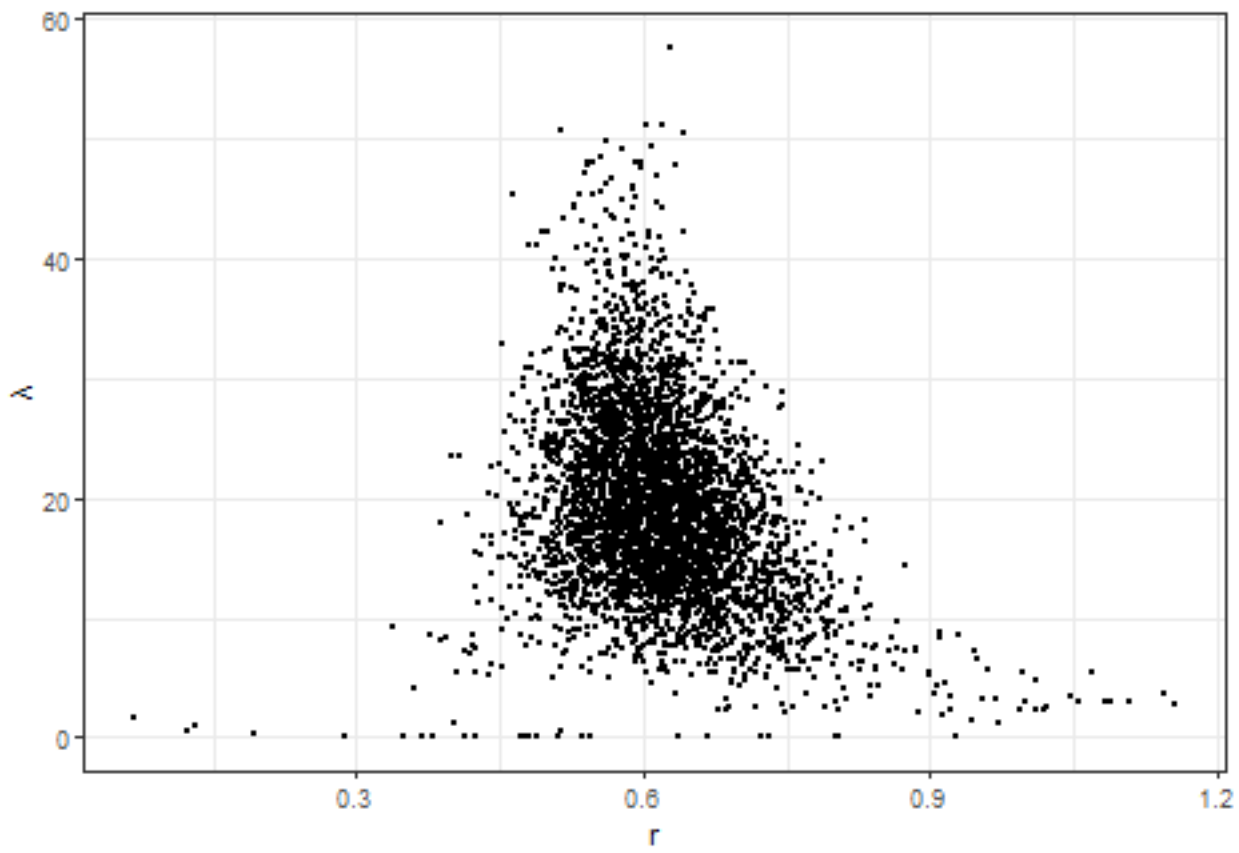


Figure 10: Posterior draws from the estimated logit choice model

Firstly, let's look at our model's prediction for this decision. Recall that since we estimated a probabilistic model, our models prediction will be a probability distribution. Furthermore, since we have 4,000 draws from the posterior distribution, we will have 4,000 posterior draws of this probability distribution. Here is what I came up with the visualize the model's prediction for this choice

```
yint<-0.3492

ProbPrediction<-tibble()

for (ss in 1:dim(postdraws)[1]) {
  r<-postdraws[ss,]$r
  lambda<-postdraws[ss,]$lambda

  EU<-0.5*Xgrid^(1-r)/(1-r)+0.5*((1-Xgrid)*yint)^(1-r)/(1-r)
  Pr<-exp(lambda*EU-max(lambda*EU))/sum(exp(lambda*EU-max(lambda*EU)))
  ProbPrediction<-rbind(ProbPrediction,tibble(x=Xgrid,Pr=Pr))
}
```

```
means<-ProbPrediction |> group_by(x) |> summarize(meanPr = mean(Pr))

(
  ggplot()
  +geom_jitter(data=ProbPrediction,aes(x=x,y=Pr),size=0.2,alpha=0.1)
  +geom_line(data=means,aes(x=x,y=meanPr),color="red",linewidth=2)
  +theme_bw()
  +geom_vline(xintercept=0.702,color="blue",linewidth=2,linetype="dashed")
  +xlab("Investment in the state 1 asset")
  +ylab("Probability")
)
```

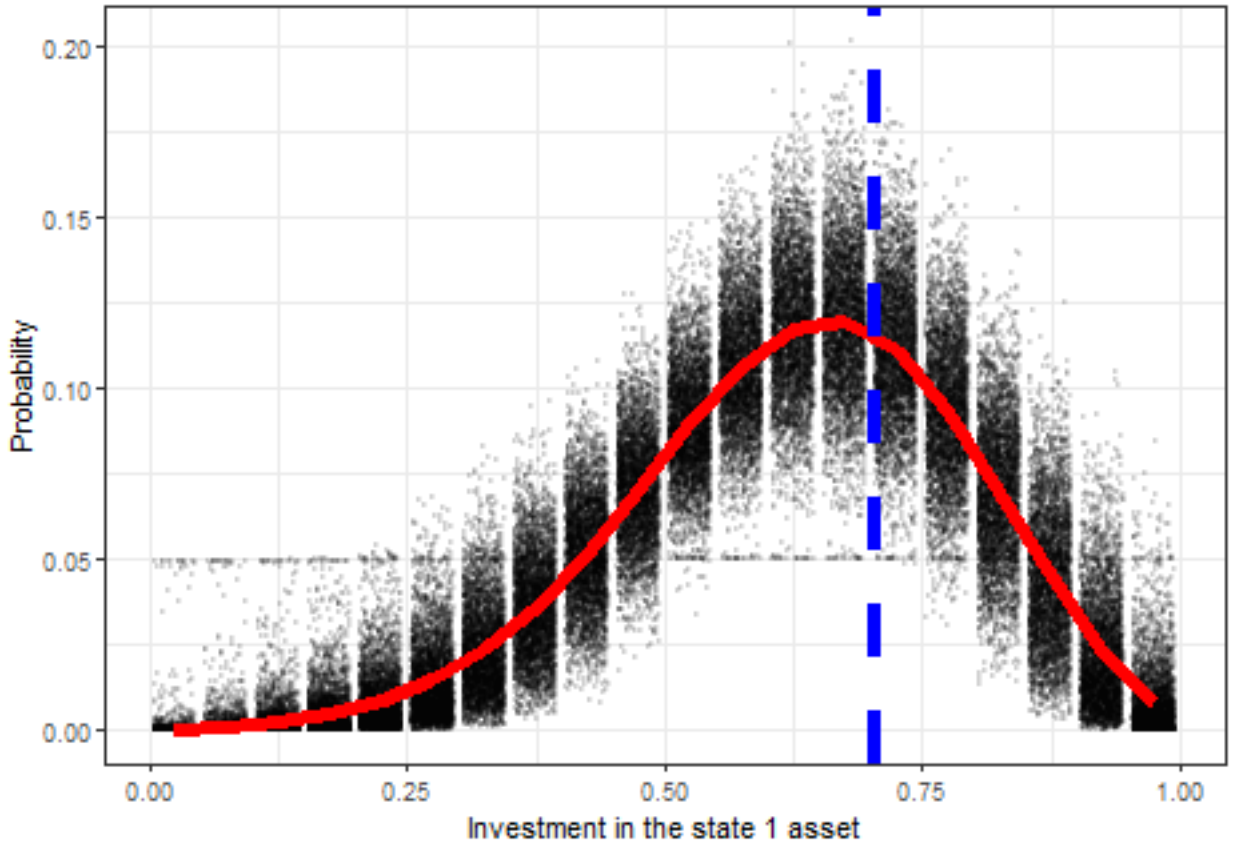


Figure 11: Posterior predictive distribution for the participant. Dots show draws from the individual posterior distribution. The red curve shows the mean of this predictive distribution. The dashed blue line shows the choice that was made by the participant.

So it looks like the participant chose fairly closely (blue line) to what our model estimated was their optimal choice.[9]

Finally, whenever I see a plot like Figure 6, I always wonder what the indifference curves look like. So in the interest of satisfying my curiosity, let's work out what their indifference curve looks like for the choice that they made. To begin with, the equation of the indifference curve is:

[9]We should be wary of using this as evidence that we have a good fit for our model, because this participant's choice was used in estimating the model. In a later chapter I will discuss some cross-validation techniques that can be used to assess goodness of fit.

$$\bar{u} = 0.5\frac{x^{1-r}}{1-r} + 0.5\frac{y^{1-r}}{1-r}$$
$$y^{1-r} = 2(1-r)\bar{u} - x^{1-r}$$
$$y = \left(2(1-r)\bar{u} - x^{1-r}\right)^{\frac{1}{1-r}}$$

where $\bar{u}$ is the utility of the indifference curve. Let's work out what the indifference curves look like using our 4,000 posterior draws:

```
indiff<-(
  expand.grid(r = postdraws$r,x=seq(0.1,0.9,length=20))
    |> mutate(
      ubar = 0.5*0.702^(1-r)/(1-r)+0.5*((1-0.702)*0.3492)^(1-r)/(1-r),
      y = (2*(1-r)*ubar-x^(1-r))^(1/(1-r))
    )
  |> group_by(x)
  |> summarize(mean=mean(y),
               p05=quantile(y,probs=0.05,na.rm=T),
               p95=quantile(y,probs=0.95,na.rm=T),
               p25=quantile(y,probs=0.25,na.rm=T),
               p75=quantile(y,probs=0.75,na.rm=T)
               )
)

(
  ggplot(indiff,aes(x=x))
  +geom_abline(slope = -0.3492,intercept = 0.3492,linewidth=1)
  +geom_ribbon(aes(ymax=p95,ymin=p05),alpha=0.2)
  +geom_ribbon(aes(ymax=p75,ymin=p25),alpha=0.4)
  +geom_line(aes(y=mean))
  +theme_bw()
  +xlab("Prize if state 1 occurs (x)")+ylab("Prize if state 2 occurs (y)")
  +geom_text(x=0.702,y=0.349*(1-0.702),label="X")
)
```

Figure 12 shows the estimated indifference curve for this first decision, Including 95% and 50% Bayesian credible regions around its point prediction. Note that in this process, once we have draws from the posterior, it is very easy not just to get point predictions that are transforms of our parameters (like this indifference curve), but also the *entire posterior distribution* of these transforms. This will be extremely useful to us, as our structural models' predictions are often nonlinear transformations of the parameters. Other quantities as well, such as utility, indifference curves, reservation values, and so on, are just as easily computed.

4 Considerations for choosing a prior

Conceptually, the Bayesian techniques I am teaching you in the book are very similar to how we would approach the same inferential goals using maximum likelihood estimation.[10] Both work with a likelihood function that specifies the distribution of data given the model's parameters. The big difference between the two is that when using Bayesian techniques, we also need to specify a prior. That is, the likelihood is our formal statement about how our model's parameters and the experiment generate data, and the prior is our formal statement about how much uncertainty we have about our model's parameters before running an experiment. Whether we have had a good hard think about the prior we want to use, whether we have just

[10]Probably the biggest difference other than the prior is the differing approach to dealing with unobservable heterogeneity, which we will get to in the hierarchical models chapter.

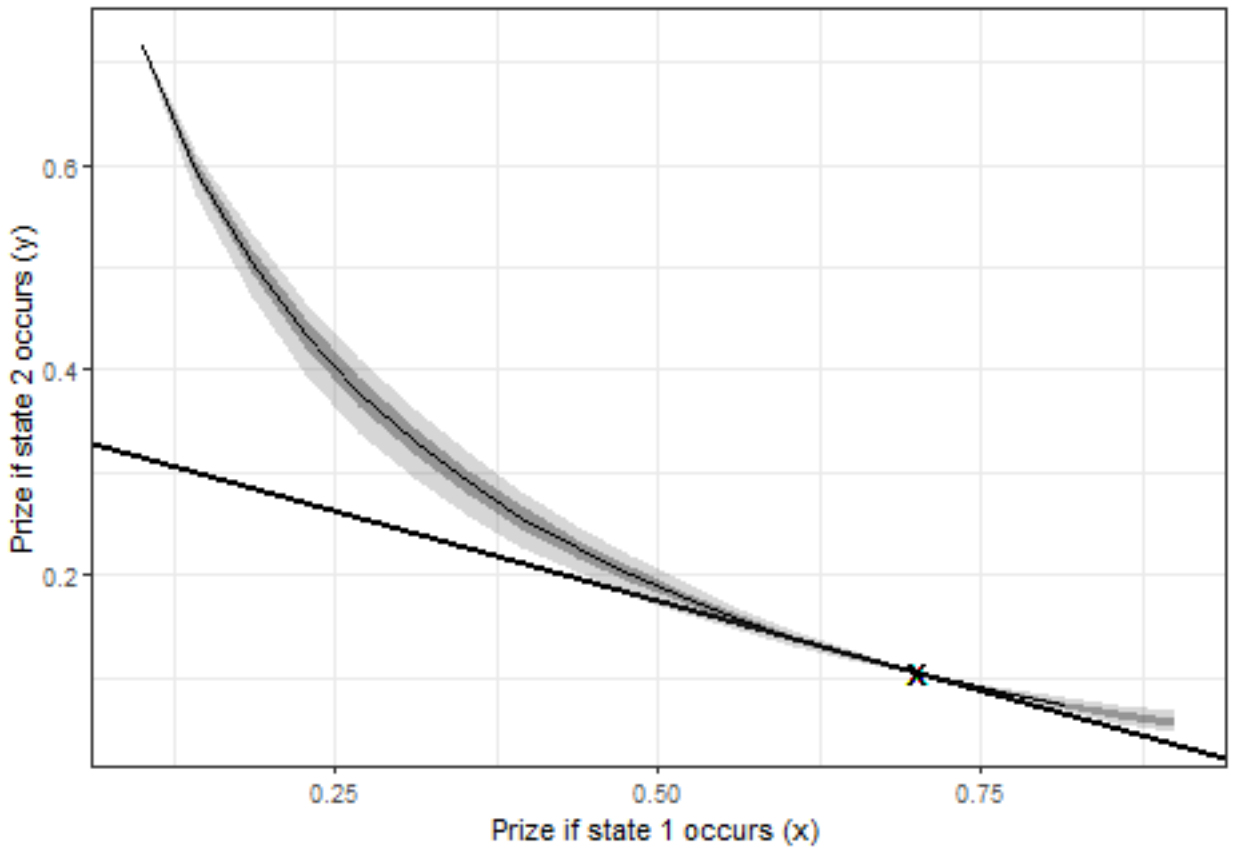


Figure 12: Estimated indifference curve of the first choice made by the participant. Shaded areas show 50% and 95% Bayesian credible regions. X denotes the choice made by the participant.

thrown up our hands and gone for an improper "uninformative" flat prior, or anything in between, these are modeling choices. We should therefore pay at least some attention to our priors, and how they affect our estimates. This is especially the case for economic experiments, where sample sizes can be relatively small, and so the prior may still have substantial influence over the posterior.

We can assess our priors in several ways. If we want to take a stand on priors representing our *actual prior beliefs* about the uncertainty in our parameters, we can calibrate our priors using our expert knowledge of what reasonable and unreasonable parameter values are. For example, this could include choosing distributions that pin down what we think are reasonable 5% and 95% lower and upper bounds on the parameters.[11] We can also scrutinize our priors on *parameters* to check that they say about behavior. For example, we may check to see whether our calibrated prior makes reasonable predictions for behavior in the experiment. This might be useful if we are very uncertain about what reasonable values of our parameters are, but we have a better idea of what behavior might look like. Additionally, we can assess our prior's performance in recovering the parameters of the model. That is, is there a prior that results in an estimator with better sampling properties than other choices?

A prior is useful not just in the *estimation* stage of our research, but also in the *design* stage. Priors can be immensely useful for tuning an experiment to best estimate our parameters of interest. A prior that actually reflects what we believe are plausible parameter values can help us choose experimental conditions that are the most informative of these parameters. This chapter is focused on ways for us to *select* a prior, not how to use it to design an experiment. I will devote a chapter to this at some point.

4.1 Example model and experiment

For this chapter I will use as an example the modified dictator game part of Bruhin, Fehr, and Schunk (2019), where participants made 78 pairwise choices of allocations between themselves and another participant. For this chapter, we will focus on estimating a simple one-parameter[12] other-regarding utility function:

$$U(\pi_1, \pi_2, \alpha) = \alpha\pi_1 + (1-\alpha)\pi_2, \quad 0 \leq \alpha \leq 1$$

where π_1 and π_2 are the payoffs of the self and the other, respectively, and $\alpha \in [0,1]$ is the weight placed on the self's payoff. Hence, $\alpha = 1$ corresponds to self-regarding preferences. We will also assume a logit probabilistic choice rule with choice precision $\lambda > 0$. Given the two allocations (π_1^X, π_2^X) and (π_1^Y, π_2^Y), we can write the likelihood as:

$$\begin{aligned} p(y_t = 1 \mid \alpha, \lambda, \pi_t) &= \Lambda\left(\lambda\left(\alpha\pi_{t,1}^X + (1-\alpha)\pi_{t,2}^X - \alpha\pi_{t,1}^Y - (1-\alpha)\pi_{t,2}^Y\right)\right) \\ &= \Lambda\left(\lambda\left(\alpha(\pi_{t,1}^X - \pi_{t,1}^Y) + (1-\alpha)(\pi_{t,2}^X - \pi_{t,2}^Y)\right)\right) \end{aligned}$$

where $y_t = 1$ iff allocation X is chosen in decision t, zero otherwise. $\Lambda(x) = (1+\exp(-x))^{-1}$ is the inverse logit function.

The 78 pairwise choices in Bruhin, Fehr, and Schunk (2019) are shown in Figure 13. Each line segment represents a pairwise choice, with the coordinates of the endpoints showing the two allocations.

```
d1<-(read.csv("Data/BFS2019_choices_exp1.csv")
     |> mutate(experiment=1)
)
d2<-(read.csv("Data/BFS2019_choices_exp2.csv")
     |> mutate(experiment=2)
)
D<-(rbind(d1,d2)
```

[11] If you have read other chapters of this book, you can see that I am a big fan of using this technique, at least as a starting point.

[12] Coupled with the logit probabilistic choice rule, this means we will have two parameters total. This makes plotting parameter values much easier compared to a 3+ parameter model.

```
    # Just use the dictator game data
    |> filter(dg==1)
    |> mutate(
      self_alloc = ifelse(choice_x==1,self_x,self_y),
      other_alloc = ifelse(choice_x==1,other_x,other_y)
    )
)

# We just need one participant's data because everything in this chapter will be simulation
BFS2019<-D |> filter(sid==102010050706)

(
  ggplot(
    BFS2019,

  )
  +geom_segment(aes(x=self_x,y=other_x,xend=self_y,yend=other_y))
  +theme_bw()
  +xlab("Own payoff")+ylab("Other's payoff")
  +geom_abline(slope=1,intercept=0,linetype="dashed")
  +coord_fixed()
)
```

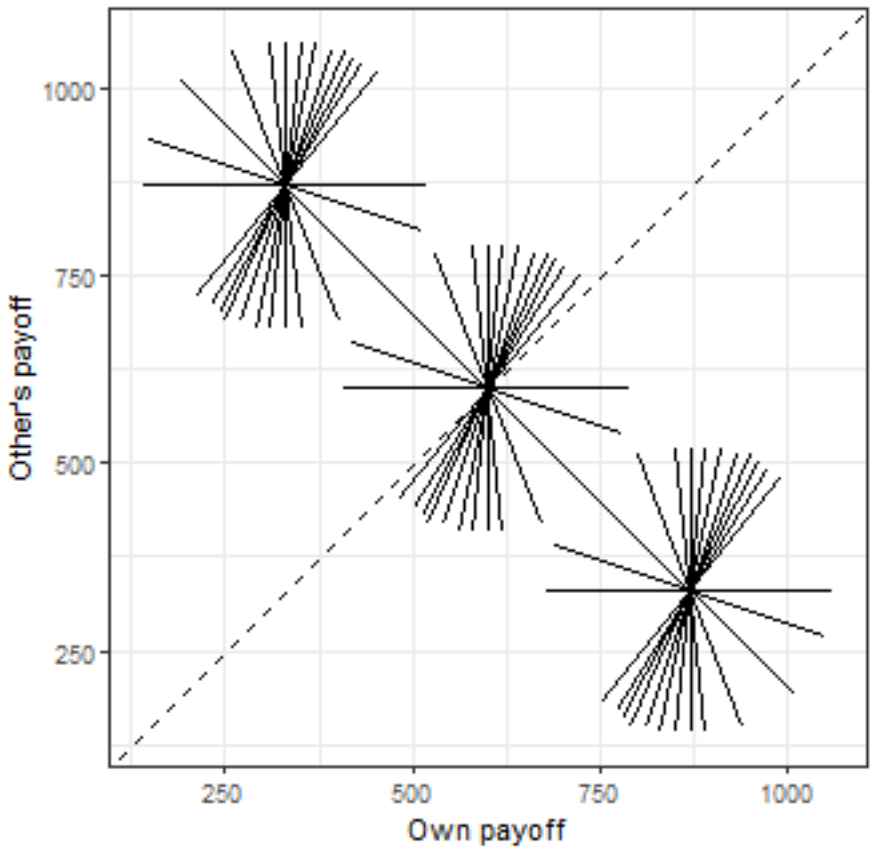


Figure 13: The 78 pairwise choices over allocations of money in the modified dictator game. Each pariwise choice is denoted by the endpoints of a line.

4.2 Getting the support right

To begin with, we need to choose which *family* of distributions we will use for our prior. This is where we rule out any nonsensical values, and make sure we do *not* rule out anything that could be reasonable. For example, if a parameter could in principle take on any real number, then a normal distribution makes sense. This allows us the freedom of choosing two properties of the prior distribution (the mean and standard deviation) to appropriately express our prior uncertainty in a parameter.

For the preference parameter α, we have been given the restriction $\alpha \in [0, 1]$, so here we need to choose a distribution that has a support of the unit interval. In this case, choosing something like the Beta distribution:

$$\alpha \sim B(a_\alpha, b_\alpha)$$

the probit-normal distribution:

$$\Phi^{-1}(\alpha) \sim N(m_\alpha, s_\alpha^2)$$

or the logit-normal distribution:

$$\log(\alpha) - \log(1 - \alpha) \sim N(m_\alpha, s_\alpha^2)$$

could be reasonable choices, as they all permit us to independently choose two properties of the distribution. Furthermore, if the *uniform* distribution is something we would want to consider using, the Beta distribution (setting $a_\alpha = b_\alpha = 1$) and the probit-normal distribution (setting $m_\alpha = 0$, $s_\alpha = 1$) nest this distribution, so these may be useful choices. For this example, I will choose the probit-normal distribution, as I use it quite commonly in my hierarchical models when I need to make this parameter restriction on an individual-level parameter.

For the choice precision parameter λ, we need to ensure that it is positive: negative values would imply that participants are *less* likely to choose the action with the greater utility, which is nonsensical. A good choice in this instance is the log-normal distribution. Again, this permits us to independently pin down two properties of the distribution.

$$\log \lambda \sim N(m_\lambda, s_\lambda^2)$$

With the likelihood and the family of prior specified, we can go ahead and code this model up in *Stan*. This will be useful even before we have data, because we can use the *Stan* model to simulate data and explore the sampling properties of our estimator. Here you will see a few extras added to the model so we can do some more with it. In particular the `UseData` variable lets us turn the likelihood contribution on or off. That way we can get draws from both the prior and the posterior with the same model. Also, the `UsePrior` variable turns the prior on and off. This will enable us to see what happens when we use improper, flat, "uninformative" priors. It will also permit us to do maximum likelihood estimation.

```
// BFS2019priors.stan

data {
  int<lower=0> n; // number of decisions
  int y[n]; // choices. 1=X, 0=Y

  vector[n] self_x; // payoff to self with allocation x
  vector[n] other_x; // payoff to other with allocation x
  vector[n] self_y; // payoff to self with allocation y
  vector[n] other_y; // payoff to other with allocation y
```

```
  // priors
  real prior_lambda[2]; // log-normal
  real prior_alpha[2]; // probit-normal

  // Turns on and off whether we are using the data
  int<lower=0,upper=1> UseData;

  // Turns on or off whether we are using the prior, or
  // "uninformative" priors
  int<lower=0,upper=1> UsePrior;
}


parameters {
  real probitAlpha;
  real<lower=0> lambda;
}

transformed parameters {
  real<lower=0,upper=1> alpha = Phi(probitAlpha);
}


model {

  // priors
  if (UsePrior==1) {
    probitAlpha ~ normal(prior_alpha[1],prior_alpha[2]);
    lambda ~ lognormal(prior_lambda[1],prior_lambda[2]);
  }

  if (UseData==1) {
    y ~ bernoulli_logit(lambda*(
      alpha*(self_x-self_y)+(1-alpha)*(other_x-other_y)
       ));
  }

}

generated quantities {

  // predicted probability of choosing allocation X
  vector[n] probX = inv_logit(lambda*(
      alpha*(self_x-self_y)+(1-alpha)*(other_x-other_y)
       ));

  // Simulated data, 1 = chose allocation X
  int simX[n] = bernoulli_logit_rng(lambda*(
      alpha*(self_x-self_y)+(1-alpha)*(other_x-other_y)
       ));

}
```

4.3 Eliciting reasonable priors

As discussed in detail in Betancourt (2020) (Section 1.1), it is important for our models not to be built on assumptions that are in conflict with our "domain expertise". That is as experts in the field, we may have a reasonable understanding of what are likely and unlikely parameter values before running an experiment, and we should incorporate this information into our prior. However translating a likely qualitative, general understanding of how participants might behave into a mathematical statement about the distribution of some parameters may not be straightforward. Two techniques Betancourt (2020) advocates for are the *prior pushforward* check, which checks whether our prior about a *parameter* is consistent with our understanding of our participants, and the *prior predictive* check, which checks whether our prior about a parameter is consistent with our understanding of our prior belief about *summaries of the data* that we might get. That is, a prior pushforward check is to check whether our chosen prior places too much, or not enough, mass in the wrong places of the parameter's support, whereas the prior predictive check is to check whether our prior implies something implausible about summaries of our data (such as participants' choices).

4.3.1 Parameter values and the prior pushforward check

Once we have settled on a family of distributions, perhaps the most direct way to hone in on the distribution within that family that we will actually use is by exploring the properties of these distributions. If we are just choosing a prior for a single (i.e. scalar) parameter, then we will most often have only a handful of degrees of freedom in our prior choice once we have selected a family of distributions. For example, the normal and beta families each allow us to independently pin down two properties of the distribution, while the exponential, Cauchy, and LKJ priors only allow for one. It is therefore very feasible to explore the properties of these distributions graphically.

```
d<-expand_grid(
  m = c(-0.4,0,0.4,0.8,1.2),
  s = c(0.25,0.5,1,2),
  sim = 1:10000
) |>
  rowwise() |>
  mutate(alpha = rnorm(1,mean=m,sd=s) |> pnorm()) |>
  ungroup() |>
  mutate(mtag = paste("m =",m),
         stag = paste("s =",s))

(
  ggplot(data=d,aes(x=alpha))
  +geom_histogram(bins=40)
  +facet_grid(stag~mtag)
  +theme_bw()
  +xlab(expression(alpha))
  +scale_x_continuous(n.breaks=4)
)
```

For example Figure 14 shows a range of priors from the probit-normal family of distributions that we could use for our prior for the other-regarding preference parameter α. In this figure, I especially want to draw your attention to the bottom row, which sets the prior scale parameter s_α to 2. If limited to these sixteen options, but without seeing Figure 14, one may be tempted to choose one of the priors with $s_\alpha = 2$. This is because these priors have the largest variance. However inspection of the figure reveals that choosing one of these priors implies a starkly bi-model distribution for α. To put this into context, these "uninformative" priors imply that it is very likely that α is close to one or zero, but very *unlikely* that it is anywhere in between. While $\alpha = 1$ corresponds to self-regarding preferences, and so we may in fact like to have a fair chunk of prior probability mass near this value, the bunching at $\alpha = 0$ should be particularly worrisome: this participant is completely selfless! In fact, we may even want to be worried about there being any appreciable mass in the $\alpha < 0.5$ region, as this participant would place more weight on the other's payoff than they do their own. My

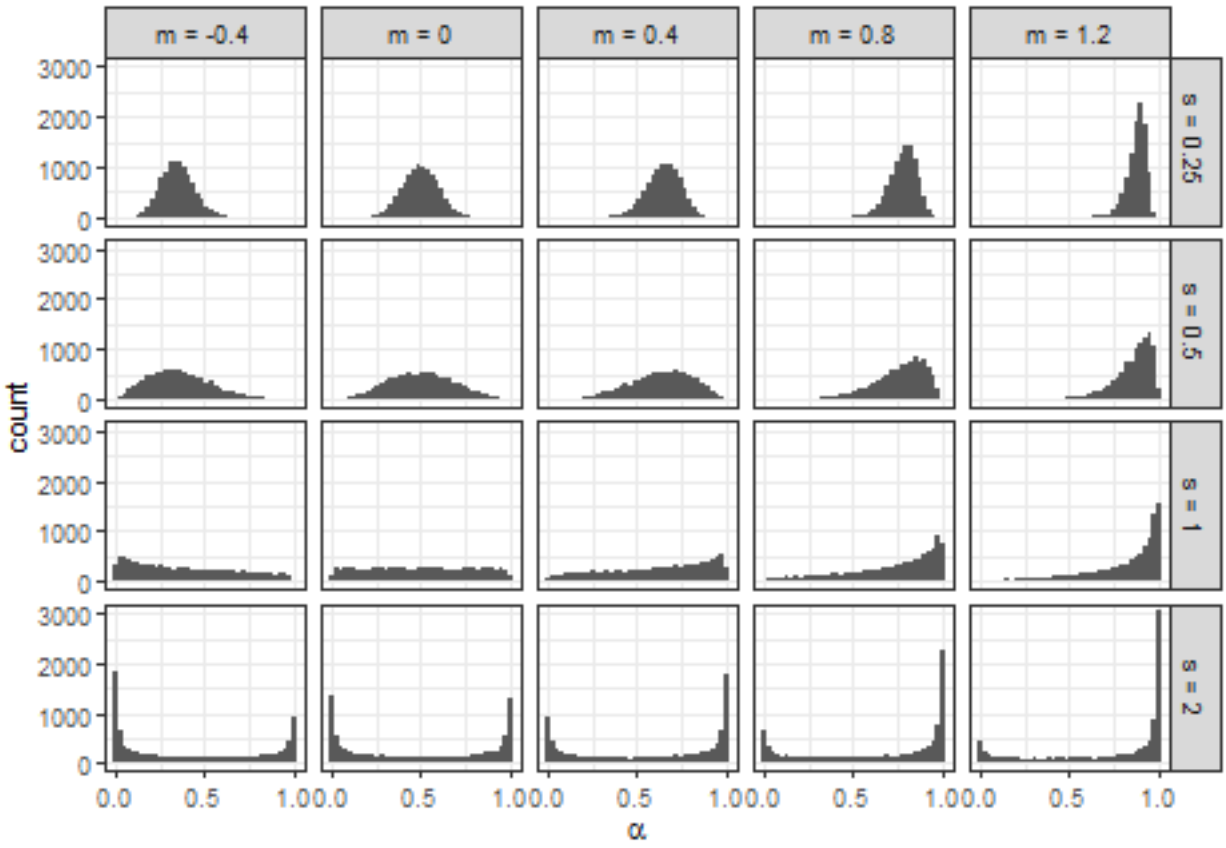


Figure 14: Histograms of 10,000 draws from the probit-normal distribution for various location (m) and scale (s) parameters.

own assessment of this figure is that the $m = 0.8$, $s = 0.5$ panel covers the $0.5 < \alpha < 1$ region fairly well, and while not ruling it out, does not place too much mass in the $\alpha < 0.5$ region.

While plotting alternative priors in something like Figure 14 can be helpful, another useful tool can be to write down a system of equations that pins down some properties that we would like the prior to have. Suppose for example that we think our prior should only place 1% probability mass on the $\alpha < 0.5$ region. This could be because we believe that it is highly unlikey that people will care more about the other person's payoff than their own. This condition tells us that:

$$\begin{aligned} p(\alpha < 0.5) &= 0.01 \\ p(\Phi^{-1}(\alpha) < \Phi^{-1}(0.5)) &= 0.01 \\ p(X < 0) &= 0.01, \quad \text{where: } X = \Phi^{-1}(\alpha) \sim N(m_\alpha, s_\alpha^2) \\ \Phi\left(\frac{-m_\alpha}{s_\alpha}\right) &= 0.01 \\ -m_\alpha &= -2.33 s_\alpha \end{aligned}$$

where $\Phi(x)$ is the standard normal cumulative density function, and $\Phi^{-1}(x)$ is its inverse.

As we have two parameters in the prior distribution to play with, we can choose one more property of the distribution. Therefore suppose that we want 20% of the probability mass to be in the $0.95 < \alpha$ range, meaning that we expect there to be a large fraction of very self-regarding participants. This sets up the following equation:

$$\begin{aligned} p(\alpha < 0.95) &= 0.8 \\ p(X < 1.64) &= 0.8 \\ \Phi\left(\frac{1.64 - m_\alpha}{s_\alpha}\right) &= 0.8 \\ \frac{1.64 - m_\alpha}{s_\alpha} &= 0.842 \\ m_\alpha &= 1.64 - 0.842 s_\alpha \end{aligned}$$

Adding the two conditions together gets us:

$$\begin{aligned} 0 &= 1.64 - 3.17 s_\alpha \\ s_\alpha &\approx 0.52 \end{aligned}$$

And then subtracting the first from the second:

$$\begin{aligned} 2m_\alpha &= 1.64 - 3.17 \times 0.52 + 2.33 \times 0.52 \\ m_\alpha &\approx 1.2 \end{aligned}$$

which is not too far off the prior I selected by eyeballing Figure 14. Note that the parameter m_α pins down the prior *median* for α, which is $\Phi(1.2) \approx 0.88$. While an explicit expression for the mean of the probit-normal distribution does not exist, we can approximate it with Monte Carlo integration:

```
rnorm(1000,mean = 1.2,sd=0.52) |> pnorm() |> mean()
```

```
## [1] 0.85508
```

Of course, if we had some estimates of our parameters from previous studies, then these could also aid us in choosing a prior. For example in Bland (2023b), I calibrate a prior for risk-aversion using a distribution of estimates for a treatment reported in Holt and Laury (2002). Even better than this, the population-level

estimates we get from hierarchical models have exactly the right interpretation we need to be priors for individual-level parameters. Therefore, if we have existing data that could speak to the parameters of a model we want to estimate, using these data to appropriately calibrate our priors could be very helpful.

4.3.2 Predictions and other derived quantities: The prior predictive check

A parameter like α can be relatively easy to interpret on its own: it represents the weight a participant places on their own payoff. However this is not always the case. For example, I find the logit choice precision parameter λ difficult to interpret. For one thing, its units are inverse utility units, whatever they are. Furthermore, without a sense of the *scale* of payoffs in an experiment, it is difficult to know what a particular value of λ implies for actual behavior. Fortunately, we can always transform a parameter, or group of parameters, into probabilistic predictions of behavior in our experiment (otherwise we would not be able to write down a likelihood), and so we can explore the implications of the more difficult-to-interpret parameters' priors using these predictions. Additionally, we can also transform our parameters into other, economically meaningful quantities, that may be easier to interpret. For example in an experiment on decision-making under risk, the parameters in our structural model might permit us to compute a certainty equivalent, which is a translation of von-Neumann-Morgenstern utility units into currency. The latter I find much easier to interpret.

For the logit choice precision parameter λ, I like to think about its implications for choice *probabilities*. To see how to do this, note that for a given utility difference $\Delta > 0$, the probability of choosing the action that yields the greatest utility is $(1 + \exp(-\lambda\Delta))^{-1}$. Hence, if we have a sense of the scale of utility differences in our experiment, we can get a sense of the influence λ has on choice probabilities. For Bruhin, Fehr, and Schunk (2019) for example, we can look at the unique values of the utility difference, evaluated at the prior median α of 0.88:

```
alpha<-0.88
Delta<-BFS2019 |>
  mutate(DU = (alpha*self_x+(1-alpha)*other_x-alpha*self_y-(1-alpha)*other_y) |> abs()) |>
  dplyr::select(DU) |>  unique() |> unlist() |> as.vector()

Delta
```

```
##  [1]  80.0 214.4 334.4 247.2 148.8 214.4 116.0  80.8 212.8 199.2 247.2 212.8
## [13]  45.6  80.0  10.4 184.0 116.0 148.8  45.6  10.4 302.4 334.4  80.8 302.4
## [25] 199.2  80.8  80.0 247.2 184.0  10.4 148.8 199.2  45.6 116.0
```

```
d<-expand.grid(
  Delta = Delta,
  lambda = 10^(seq(-5,1,length=1000))
  ) |>
  mutate(
    Pr = 1/(1+exp(-lambda*Delta))
  )

(
  ggplot(d,aes(x=lambda,y=Pr,group=Delta))
  +geom_path()
  +theme_bw()
  +scale_x_continuous(trans="log10")
  +xlab(expression(lambda))+ylab("Probability of choosing action with higher utility")
)
```

We can then plot the logit choice rule as a function of λ to see how choice probabilities could change with λ. This is done in Figure 15. Here we can see that for $\lambda < 10^{-4}$, all decisions are effectively coin flips: we might not want to assign too much prior probability in this region. Likewise for $\lambda > 1$, decisions are effectively deterministic. Suppose then, that we decide to assign 99% of our prior probability to the region $10^{-4} < \lambda < 1$,

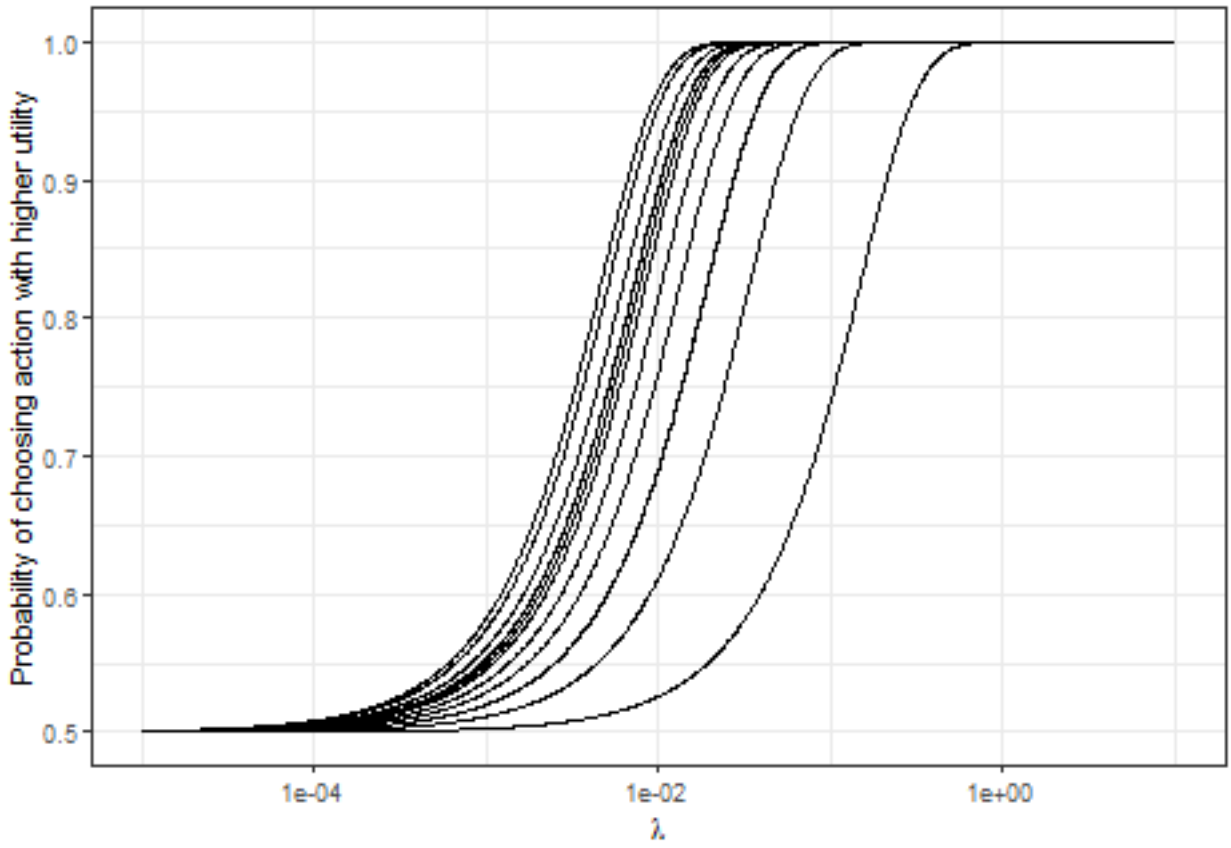


Figure 15: The implications of choice precision λ on choice probabilities.

and use these endpoints as the 0.5th and 99.5th percentiles of the prior. This gives us a system of equations to solve for m_λ and s_λ:

$$\begin{aligned} -2.57 s_\lambda &= \log(10^{-4}) - m_\lambda \\ 2.57 s_\lambda &= \log(1) - m_\lambda \end{aligned}$$

Adding them together yields:

$$\begin{aligned} 2 m_\lambda &= \log(10^{-4}) + \log(1) \\ m_\lambda &\approx -4.61 \end{aligned}$$

Subtracting the first from the second yields:

$$\begin{aligned} 2 \times 2.57 s_\lambda &= \log(1) - \log(10^{-4}) \\ s_\lambda &\approx 1.79 \end{aligned}$$

If we code it up the right way, we can also have our *Stan* model help us in choosing a prior. This is because we can always turn off the likelihood contribution with an appropriately-placed `if` statement. This is done using the `UseData` variable in the *Stan* model above. For the calibrated priors for α and λ, I simulate the *prior predictive* distribution of choices. I then summarize these distributions in Figure 16. This provides a useful "sanity check" to test whether or not our priors are implying anything weird about choices in the experiment.

```
PriorCheck<-"Code/Priors/priorCheck.rds" |>
  readRDS() |>
  data.frame() |>
  arrange(mean)

PriorCheck<-PriorCheck[rownames(PriorCheck)!="lp__",]

PriorCheck<-PriorCheck |>
  mutate(cumul = (1:n())/n())

(
  ggplot(PriorCheck,aes(x=mean,y=cumul))
  +geom_point()
  +geom_errorbar(aes(xmin = X25.,xmax = X75.))
  +geom_errorbar(aes(xmin = X2.5.,xmax = X97.5.),alpha=0.2)
  +theme_bw()
  +ylab("Cumulative density of prior mean")
  +xlab("Choice probability")
)
```

4.4 Assessing the sampling performance of a prior

Another way to investigate the implications of a prior is to explore the sampling properties of our estimator. As we have a probabilistic model, this can be done through simulation. That is, since we have a likelihood, we can draw simulated data from $y \mid \theta$, the distribution of y given a specific realization of parameters θ. Furthermore, since we also have a prior, we can simulate data that are consistent with our prior belief about these θs. In particular, the following simulation process:

1. Draw parameters $\tilde{\theta}$ from the prior
2. Draw data $y \mid \tilde{\theta}$ using the likelihood
3. Simulate the posterior $\theta \mid y$ using the simulated data.

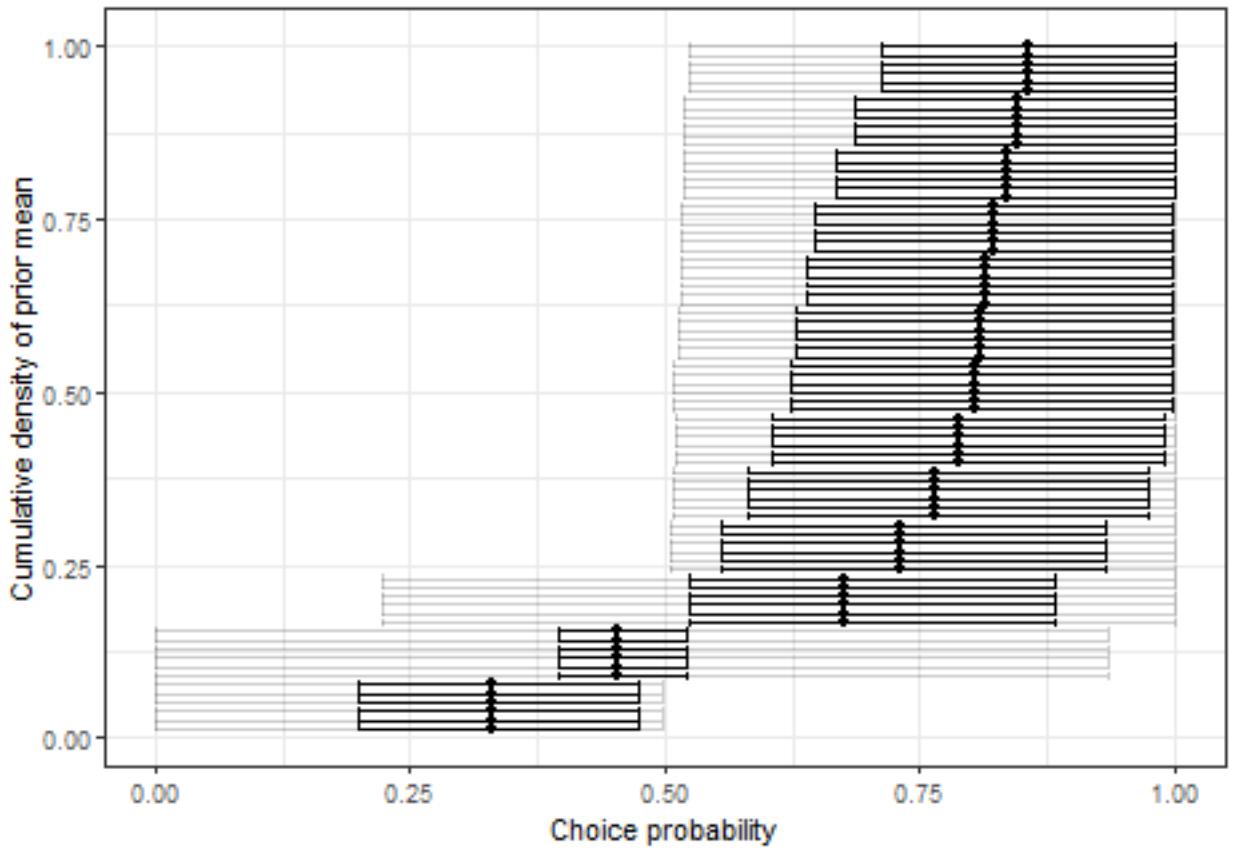


Figure 16: Prior distribution of choice probabilities. Dots show prior means. Heavy error bars show a 50% Bayesian credible region, and light error bars show a 95% Bayesian credible region.

produces data (in step 2) and estimates (in step 3) that are consistent with our prior. This information will let us do two important diagnostics of our estimation process. Firstly, we can ask if the model *recovers its parameters well*. That is, are our estimates $\hat{\theta}$ in some sense "close" to their true value θ? Secondly, we can check whether our model is showing any pathologies by comparing the posterior distributions to the prior distribution. That is if all is working correctly, then draws from the prior in step 1 should have the same distribution as the posterior draws from step 3.

I demonstrate how this simulation is done in the *R* code at the end of this chapter. Basically, we first use *Stan* to get us some parameter draws from the prior distribution, and one simulated dataset consistent with each of these prior draws. I do this by running my *Stan* program with the `UseData` variable set to zero. Then for each of these simulated datasets, we fit the model and store the posterior draws.[13]

4.4.1 Does our model recover its parameters well?

A useful diagnostic for our model is to ask whether it estimates our parameters or quantities of interest well. Using the simulation described above, we can compare our posterior means to the prior draws that generated them. I show an example of this in Figure 17. Each dot represents a simulated dataset, with the horizontal coordinate equal to the (true) prior draw of α that generated the dataset, and the vertical coordinate equal to the posterior mean of α. Ideally, all of these points would fall on the 45° line (red dashed line), however we can expect the posterior mean to be pulled at least a little bit toward the prior mean, hence the smoothed fit (blue curve) crossing the 45° line. Of course, this plot is just as much a function of your chosen prior as it is of your likelihood and experiment design. If this plot is not to your liking, then re-designing your experiment at this stage could go a long way.[14]

```
ParamRecovery<-readRDS("Code/Priors/SimSummary.rds") |>
  group_by(SimStep,alpha_true,lambda_true,alpha_MLE,lambda_MLE) |>
  summarize(alpha = mean(alpha),lambda = mean(lambda))

(
  ggplot(ParamRecovery,aes(x=alpha_true,y=alpha))
  +geom_point(size=1,alpha=0.5)
  +theme_bw()
  +geom_abline(intercept=0,slope=1,linetype="dashed",color="red")
  +geom_smooth(formula="y~x",method="loess")
  +geom_vline(xintercept=pnorm(1.2),color="red")
  +xlab("True value")
  +ylab("Posterior mean")
  +ylim(c(0,1))
)
```

In fairness to the Bayesian model, I also did the same exercise using the maximum likelihood estimator. The calibration plot can be seen below in Figure 18. Here there is clearly much more variance in the estimates, with a fair number of instances where the estimate falls at one of the endpoints of the parameter space.

```
(
  ggplot(ParamRecovery,aes(x=alpha_true,y=alpha_MLE))
  +geom_point(size=1,alpha=0.5)
  +theme_bw()
  +geom_abline(intercept=0,slope=1,linetype="dashed",color="red")
  +geom_smooth(formula="y~x",method="loess")
```

[13] The process is simple, but took a while (a couple of hours on my laptop). If you're playing at home, now would be a good time to slay a few lynels.

[14] For this experiment in particular, which was not designed to estimate this utility function, I suspect that including more decisions where the allocations are connected by a line segment with a negative slope would improve this plot dramatically. This is because the indifference curves of the utility function that we are using will be downward-sloping, parallel, straight lines. Bruhin, Fehr, and Schunk (2019) designed this experiment with inequality-aversion in mind, rather than the efficiency-loving model that I use here.

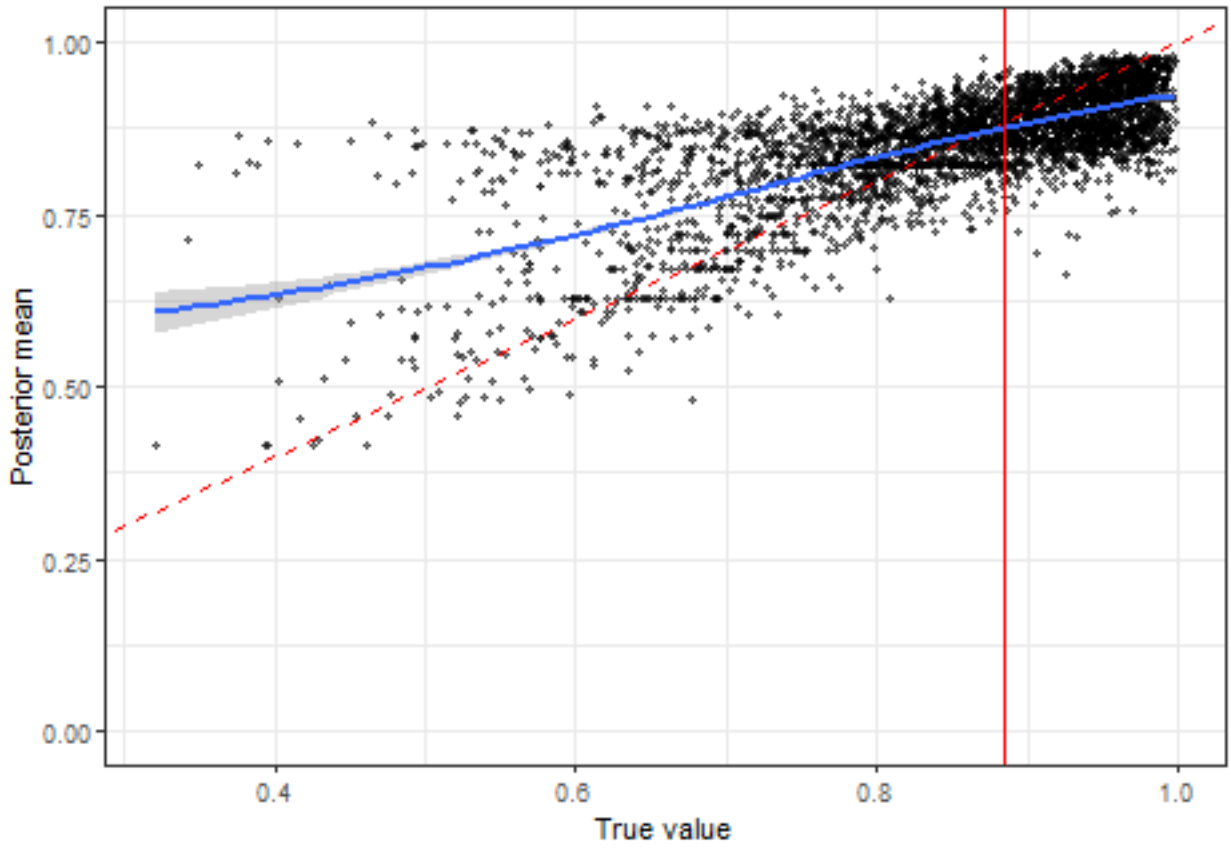


Figure 17: Parameter recovery plot for α

```
  +geom_vline(xintercept=pnorm(1.2),color="red")
  +xlab("True value")
  +ylab("Posterior mean")
  +ylim(c(0,1))
)
```

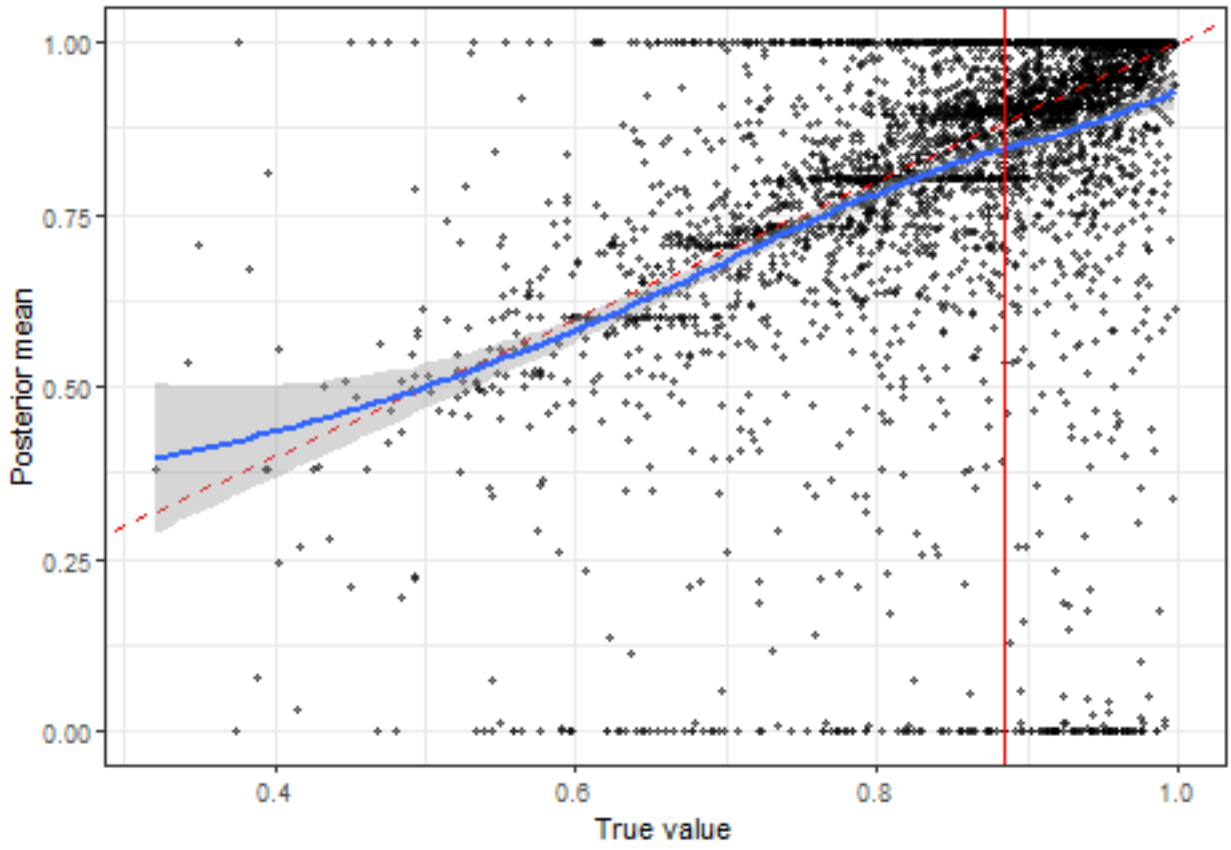


Figure 18: Parameter recovery plot for α using maximum likelihood estimation

4.4.2 Do we see any pathologies in the estimation process?

Another sanity check for our prior and model that we can do with the above simulation process is to compare the draws from the prior to the draws from the posteriors. This is because the *average* posterior distribution should be the prior distribution (see Betancourt (2020) Section 1.2). We can do this by computing the rank of each prior draw within its corresponding posterior distribution (that we estimated using the dataset simulated from that prior draw). Then, if we plot the histogram of these ranks, we should get something close to a uniform distribution, which is what we get in Figure 19. I show these histograms alongside a 95% range for the binomial distribution (using the normal approximation) in order to give you a perspective for what normal statistical variation should look like.[15] It is common for computational problems to manifest themselves in easy-to recognize ways in these plots (Betancourt 2020).

```
Rank<-readRDS("Code/Priors/SimSummary.rds") |>
  group_by(SimStep) |>
  summarize(
    alpha = sum(mean(alpha_true)<alpha),
```

[15]Betancourt (2020) also does this, it is not my original idea.

```
    lambda = sum(mean(lambda_true)<lambda),
  ) |>
  pivot_longer(cols = c(alpha,lambda),names_to="parameter",values_to = "rank")


SimSize<-Rank$SimStep |> max()
nbins<-20

(
  ggplot(Rank,aes(x=rank))
  +geom_histogram(bins=nbins)
  +facet_wrap(~parameter)
  +theme_bw()
  +geom_hline(yintercept = SimSize/nbins,color="red")
  +geom_hline(yintercept = SimSize/nbins+1.96*sqrt(SimSize*(1/nbins)*(1-1/nbins)),color="red",linetype=
  +geom_hline(yintercept = SimSize/nbins-1.96*sqrt(SimSize*(1/nbins)*(1-1/nbins)),color="red",linetype=
)
```

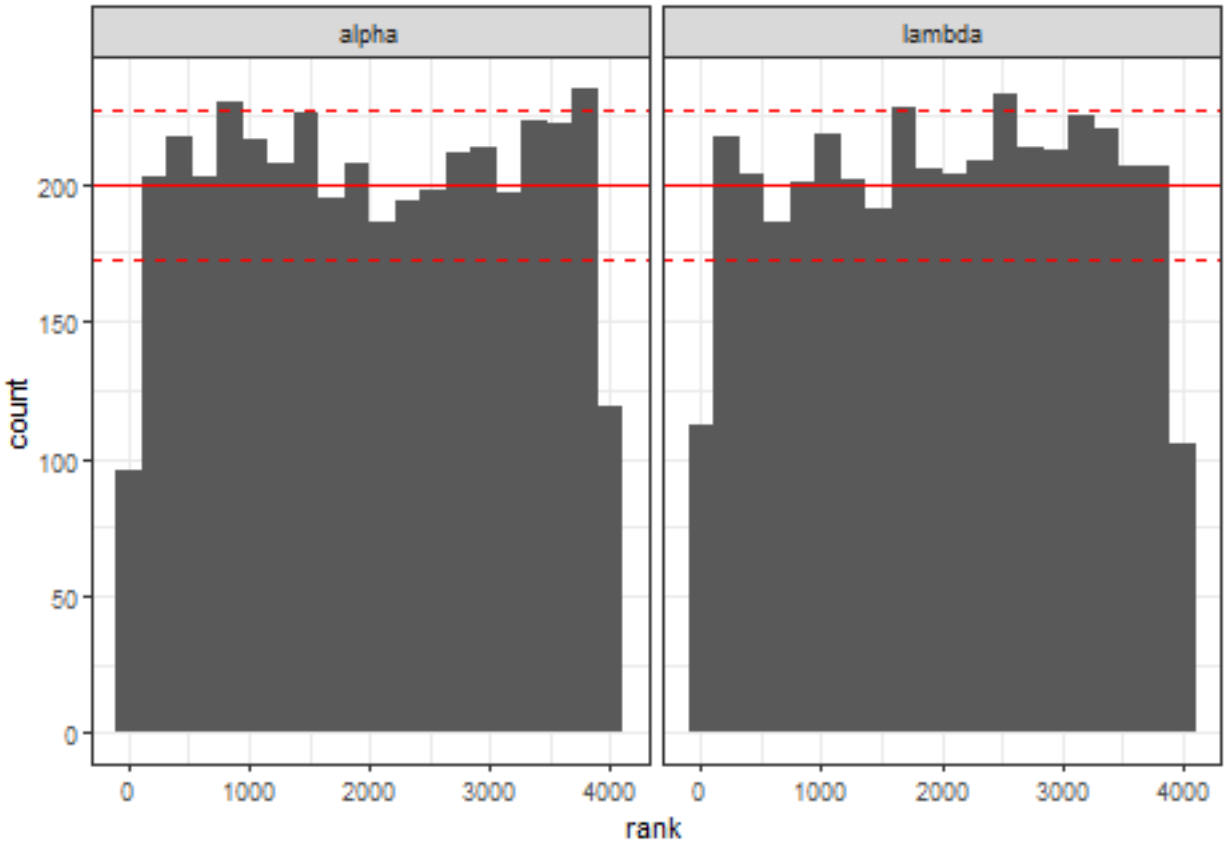


Figure 19: Histograms of ranks of prior draws within their corresponding posterior simulations.

Alternatively, we could visualize these ranks as an empirical cumulative density function, as I have done in Figure 20. Here, we are hoping that the plot will be of a straight line.

```
(
  ggplot(Rank,aes(x=rank))
  +stat_ecdf()
```

```
  +facet_wrap(~parameter)
  +theme_bw()
  +geom_abline(intercept=0,slope = 1/SimSize,color="red",linetype="dashed")
)
```

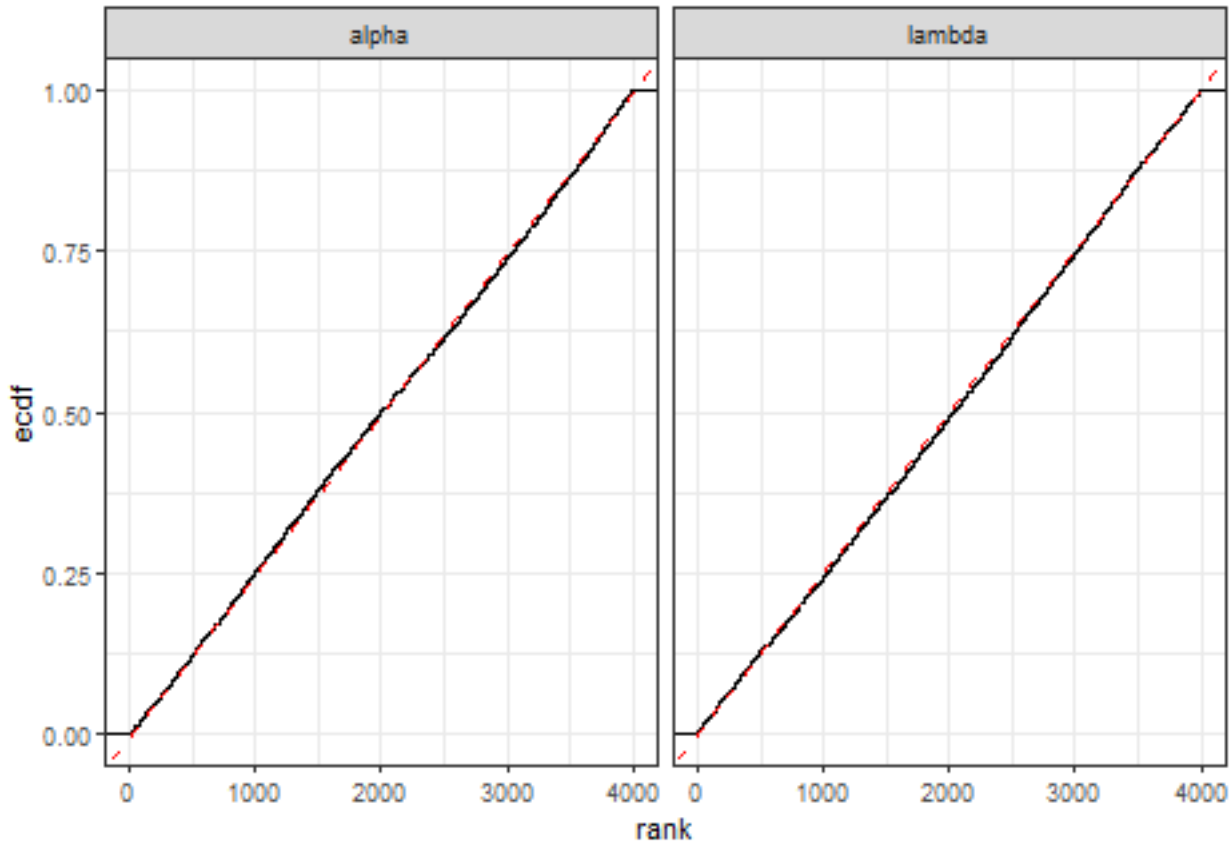


Figure 20: Empirical cumulative density function of ranks of prior draws within their corresponding posterior simulations. Dashed red line shows the ideal case of a uniform distribution.

4.5 *R* code used for this chapter

```
library(tidyverse)
library(rstan)
  # These models estimate a lot faster without parallelization
  # This is because the initialization of the cores takes much
  # longer than the estimation itself
  options(mc.cores = 1)
  #options(mc.cores = parallel::detectCores())
  rstan_options(auto_write = TRUE)
  rstan_options(threads_per_chain = 1)

model<-stan_model("Code/Priors/BFS2019priors.stan")

# Load the data

d1<-(read.csv("Data/BFS2019_choices_exp1.csv")
```

```
     |> mutate(experiment=1)
)
d2<-(read.csv("Data/BFS2019_choices_exp2.csv")
     |> mutate(experiment=2)
)
D<-(rbind(d1,d2)
    # Just use the dictator game data
    |> filter(dg==1)
    |> mutate(
      self_alloc = ifelse(choice_x==1,self_x,self_y),
      other_alloc = ifelse(choice_x==1,other_x,other_y)
    )
)

# Data from just one participant
BFS2019<-D |> filter(sid==102010050706)

# Checking that it works
dStan<-list(
  n = dim(BFS2019)[1],
  y = BFS2019$choice_x,
  self_x = BFS2019$self_x,
  self_y = BFS2019$self_y,
  other_x = BFS2019$other_x,
  other_y = BFS2019$other_y,

  prior_lambda = c(-4.61,1.79),
  prior_alpha = c(1.2,0.52),
  UseData = 1,
  UsePrior = 1
)

Fit<-sampling(model,data=dStan,seed=42)

## Simulating the prior distribution of choices

dStan$UseData<- 0
Fit<-sampling(model,data=dStan,seed=42,
              pars = "probX",include=TRUE)
summary(Fit)$summary |> saveRDS("Code/Priors/priorCheck.rds")

## Simulating some data from the prior, then estimating the models
# using those data

SimData<-sampling(model,data=dStan,seed=42,
              pars = "probX",include=FALSE)

SIMDATA<-extract(SimData)$simX
  SIMDATA |> saveRDS("Code/Priors/SimData.rds")
alpha<-extract(SimData)$alpha
lambda<-extract(SimData)$lambda

dStan$UseData<-1
```

```
SimSummary<-tibble()

for (ss in 1:dim(SIMDATA)[1]) {
  print(paste("Sim step",ss,"of",dim(SIMDATA)[1]))

  # Estimate the model
  dStan$y<-SIMDATA[ss,] |> as.vector()
  Fit<-sampling(model,data=dStan,seed=42,
                pars = c("alpha","lambda"),include=TRUE)

  # Also check to see what the MLE looks like
  dStanMLE<-dStan
  dStanMLE$UsePrior<-0

  MLE<-optimizing(model,data=dStanMLE,seed=42)

  SimSummary<-rbind(
    SimSummary,
    tibble(
    SimStep = ss,
    alpha_true = alpha[ss],
    lambda_true = lambda[ss],
    alpha = extract(Fit)$alpha,
    lambda = extract(Fit)$lambda,
    alpha_MLE = MLE$par["alpha"],
    lambda_MLE = MLE$par["lambda"]
    )
  )
}

SimSummary |> saveRDS("Code/Priors/SimSummary.rds")
```

Building blocks

5 Representative agent and participant-specific models

Our participants are not homogeneous. Therefore representative agent models do not get us very far, both as economic models, and as econometric models. For economic models, this is because choices do not aggregate unless preferences meet some very strict criteria. Econometric models motivated from these economic models will inherit these problems, and create their own. In particular, we should be worried about *heterogeneity bias*, which occurs when we assume that a parameter is homogeneous across participants, when it is in fact heterogeneous. For example Wilcox (2006) shows that conclusions about the type of learning that participants are doing in a game will be biased in favor of one model (reinforcement learning) if heterogeneity is not addressed in the econometric model.

While I therefore avoid pooled representative agent models for most of my work, they do provide us with a useful stepping stone in taking an economic model through the process of becoming a probabilistic econometric model. Estimating a representative agent model, even if we do not take it too seriously, is a useful proof-of-concept before advancing further into the depths of the hierarchical structure that our data probably deserves. This is because the representative agent model will likely identify certain coding or computational issues that are specific to your problem. For example in Bland (2022) I spent a lot of time honing my pooled Quantal Response Equilibrium model before moving on to the hierarchical and off-equilibrium specifications. This

first established for me that it was in fact feasible to estimate a Quantal Response Equilibrium model at all in *Stan*. If it hadn't been, I would have discovered this much earlier in the research process than had I tried to do the hierarchical, off-equilibrium with risk-aversion model first. In the process I was able to identify the bits of the code that were slowing things down, and work out more speedy solutions to computing Quantal Response Equilibrium in a much simpler environment. Furthermore, as you will see in some later chapters of this book, almost all of the work that follows once you have a representative agent model to extend it to a hierarchical or mixture model is quite mechanical: it will look the same irrespective of the specific model you are taking to the data. As such, once you get the hang of these more complicated specifications, I suspect that you will actually spend more time thinking about the parts of your code that would also be in the representative agent model.

Representative agent models are also useful if all we want are participant-specific estimates. When all we are using are data from one participant, the representative agent and the participant are the same person. Instead, the "representative agent" assumption is probably better called a "stable preferences" assumption, where we are assuming that the participant's utility function remains the same for the entire experiment. In this chapter, I will therefore present you with a guide to estimating participant-specific models.

5.1 Participant-specific models

5.1.1 Example data and economic model

In this section, I will use data from the modified dictator games in Bruhin, Fehr, and Schunk (2019). In this part of the experiment, 174 participants made 78 pairwise choices over allocations of money between themself and another participant. The 78 pairwise allocations, and the choices of the first participant to show up in the data, can be seen in Figure 21. Here you can see these choices described by a line connecting the two allocations. There is a dot at the allocation chosen by the first participant. In this Figure you can see that each participant is presented with choices where they always earn less than the other person (top-left corner), when they always earn more than the other person (bottom-right corner), and when the two allocations span over both of these regions (in the middle of the plot).

```
library(tidyverse)

d1<-(read.csv("Data/BFS2019_choices_exp1.csv")
     |> mutate(experiment=1)
)
d2<-(read.csv("Data/BFS2019_choices_exp2.csv")
     |> mutate(experiment=2)
)
D<-(rbind(d1,d2)
     # Just use the dictator game data
     |> filter(dg==1)
     |> mutate(
       self_alloc = ifelse(choice_x==1,self_x,self_y),
       other_alloc = ifelse(choice_x==1,other_x,other_y)
     )
)

(
  ggplot(
    D |> filter(sid==102010050706),

  )
  +geom_segment(aes(x=self_x,y=other_x,xend=self_y,yend=other_y))
  +geom_point(aes(x=self_alloc,y=other_alloc))
  +theme_bw()
```

```
  +xlab("Own payoff")+ylab("Other's payoff")
  +geom_abline(slope=1,intercept=0,linetype="dashed")
  +coord_fixed()
)
```

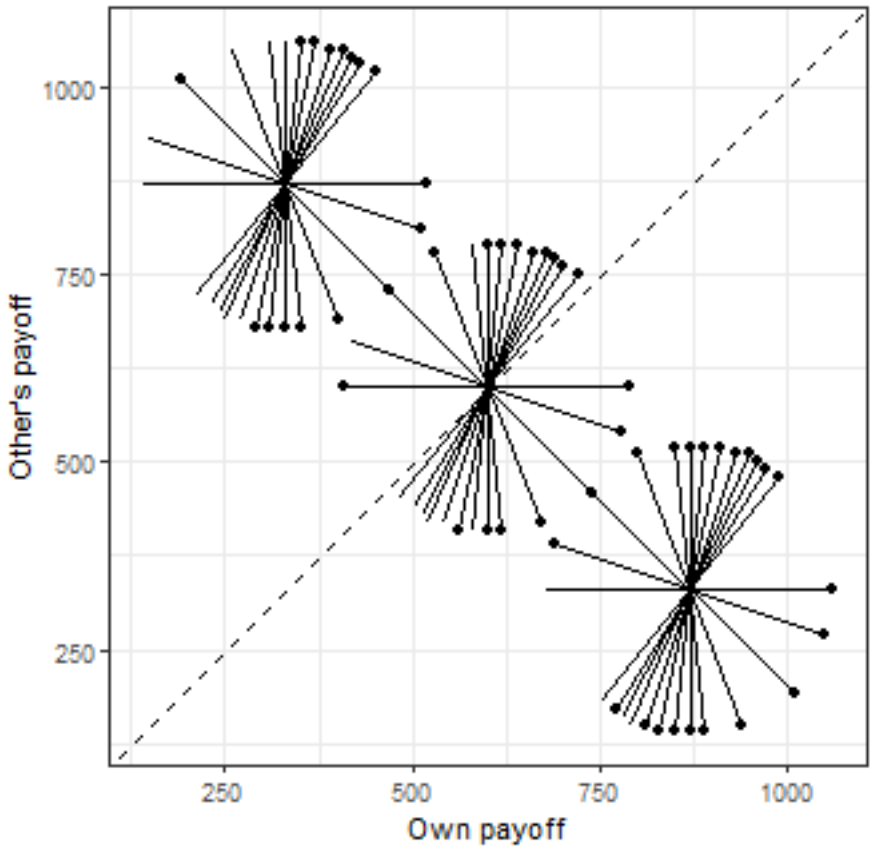


Figure 21: The 78 pairwise choices over allocations of money. Each pariwise choice is denoted by the endpoints of a line. The decisions of the first participant are shown as dots at the chosen endpoint.

Bruhin, Fehr, and Schunk (2019) estimate the parameters in the Fehr and Schmidt (1999) model, which assumes that the decision-maker's utility over allocations of money is linear in their own earnings, and the earnings of the other person, but there is a change in the marginal utility of money depending on whether the other person is earning less or more than the decision-maker. Here is the Fehr and Schmidt (1999) model (written just for two people, which is all we have here):

$$U(\pi_1, \pi_2; \alpha, \beta) = \pi_1 - \alpha \max\{\pi_2 - \pi_1, 0\} - \beta \max\{\pi_1 - \pi_2, 0\}$$

Where π_1 is the decision-maker's earnings, π_2 is the other person's earnings, and α and β are the parameters of the model. Fehr and Schmidt (1999) further impose the parameter restrictions of $\beta \in [0, 1]$ and $\alpha > \beta$, which make it a model of *inequality aversion*: all else held equal, the decision-maker prefers equal allocations of money. Restricting $\beta < 1$ means that the decision-maker would not be willing to burn money to equalize earnings. As these are parameter restrictions that are testable, I will instead estimate the models without these restrictions, and then comment on how empirically relevant these parameter restrictions seem to be. This is also how Bruhin, Fehr, and Schunk (2019) approaches the problem.

Other than the restrictions initially imposed by Fehr and Schmidt (1999), it is also meaningful to divide the

α-β space into four regions based on the signs of these parameters. These regions tell us qualitatively the kind of preferences that the parameters imply:

- $\alpha > 0$ and $\beta > 0$: the participant is *inequality-averse*. Holding their own income equal, they prefer outcomes where the other person earns the same as them.
- $\alpha < 0$ and $\beta < 0$: the participant is *inequality-loving*. Holding their own income equal, they prefer to have the other person earning a lot more or a lot less than them.
- $\alpha < 0$ and $\beta > 0$: the participant is *efficiency-loving*. Holding their own income equal, they prefer the other person to earn more.
- $\alpha > 0$ and $\beta < 0$: The participant is *efficiency-averse* or *competitive*. Holding their own income equal, the prefer the other person to earn less.

5.1.2 Going to the probabilistic model

Since we have a deterministic model specified above, we need to modify it so that it makes probabilistic predictions. As we have binary choice, it does not make sense to estimate a "optimal choice plus error" model. Therefore I will use the utility-based model of probabilistic choice discussed in the previous chapter: logit choice. This is also the model used in the maximum likelihood specification of Bruhin, Fehr, and Schunk (2019). In the data, each allocation is coded as "Choice X" and "Choice Y", and the binary variable identifying which allocation is chosen is equal to one if and only if X is chosen. Therefore, we can parameterize the logit choice model as follows:

$$\Pr\left(X_{i,t} = 1 \mid \alpha_i, \beta_i, \lambda_i\right) = \Lambda\left(\lambda\left(U(\pi^X_{1,t}, \pi^X_{2,t}; \alpha, \beta) - U(\pi^Y_{1,t}, \pi^Y_{2,t}; \alpha, \beta)\right)\right)$$

where $\Lambda(x) = 1/(1 + \exp(-x))$ is the inverse logit function.

We could also do this by specifying a distribution:

$$X_{i,t} \sim \text{Bernoulli}\left(\Lambda\left(\lambda\left(U(\pi^X_{1,t}, \pi^X_{2,t}; \alpha, \beta) - U(\pi^Y_{1,t}, \pi^Y_{2,t}; \alpha, \beta)\right)\right)\right)$$

Therefore we have three parameters to estimate: α and β, the parameters in the original economic model, and logit choice precision λ, which we need to make the model probabilistic.

5.1.3 A short side quest into canned estimation techniques

Before moving on, it is very important to note a practical implication of the Fehr and Schmidt (1999) model with logit choice: **it is a logit model**. As in, **it is a logit model like you might estimate in *Stata* or *R*!**. This is an immensely useful realization that could speed up your analysis if you were OK with estimating this model with pre-packaged estimation routines. Specifically, with a few transforms of the data, you can estimate λ, and (linear transforms of) α, and β using a standard logit. To see this, note that we can transform the data as follows:

$$d^X = \max\{\pi_2 - \pi_1, 0\}, \quad a^X = \max\{\pi_1 - \pi_2\}$$

and similarly for allocation Y, we can then write the model as:

$$\begin{aligned}\Pr(X_{i,t} \mid \alpha_i, \beta_i, \lambda_i) &= \Lambda\left(\lambda\left(\pi^X_{1,t} - \alpha_i d^X_t - \beta_i d^X_t - \pi^Y_{1,t} + \alpha_i d^Y_t + \beta_i d^Y_t\right)\right) \\ &= \Lambda\left(\lambda(\pi^X_{i,t} - \pi^Y_{i,t}) + \lambda\alpha_i(d^Y_t - d^X_t) + \lambda\beta_i(a^Y_t - a^X_t)\right)\end{aligned}$$

which is a logit model without an intercept, and so we can estimate the model as follows (here I am just doing it for the first participant):

```r
library(stargazer)
D<-(
  D
  |> rowwise()
  |> mutate(dX = max(c(0,other_x-self_x)),
            dY = max(c(0,other_y-self_y)),
            aX = max(c(0,self_x-other_x)),
            aY = max(c(0,self_y-other_y))
            )
  |> ungroup()
  |> mutate(x1 = self_x-self_y,
            x2 = dY-dX,
            x3 = aY-aX)
)
model<-glm(data=D|> filter(sid==102010050706),
           formula=choice_x~x1+x2+x3-1,
           family=binomial(link="logit"))
stargazer(model,type="html",digits=6)
```

Dependent variable:

choice_x

x1

0.008173***

(0.001845)

x2

0.000443

(0.001339)

x3

0.001269

(0.001277)

Observations

78

Log Likelihood

-38.115470

Akaike Inf. Crit.

82.230940

Note:

p<0.1; ***p<0.05;*** p<0.01

So here we have $\hat{\lambda} = 0.0081731$. We haven't directly estimated α and β. Instead, we have estimated $\lambda\alpha$ and $\lambda\beta$, so we need to divide by $\hat{\lambda}$ to get $\hat{\alpha} = 0.0541735$, and $\hat{\beta} = 0.1552275$.

5.1.4 Assigning priors

Up to this point, I have not talked much about priors. This stops here! Our priors are the information we give our econometric model as experts (yes, you're one of them) about the economic model.

Let's start with the choice precision parameter λ. For me, this is always the most difficult to assign, because it does not have a great economic interpretation. Sure, larger values mean more precise choices, but you really don't have any idea about scale until you look at some predictions implied by it. Let's calibrate this prior to a reasonable distribution based on what it implies about the choices of a self-regarding participant. That is, when we set the other paranmeters $\alpha = \beta = 0$, what do the probabilistic predictions of our model look like as we change λ. To do this, I will first get a smaller dataset together of all the unique payoff differences faced by a selfish decision-maker:

```
d<-(D
    |> mutate(absDiff = abs(self_x-self_y))
    |> dplyr::select(absDiff)
    |> unique()
    )

d |> as.vector() |> print()
```

```
## $absDiff
##  [1] 140 200 380 240 120  80  40 280 180   0 160 360
```

It looks like we have twelve of them. Let's plot the probability of choosing the action with the greatest payoff against λ. Since λ (kind-of) happens on a log scale, it is helpful to have this in a log scale:

```
lambdaCheck<-(
  expand.grid(lambda = 1/(1-seq(0.0001,0.6,length=1001))-1,
              absDiff = d$absDiff)
  |> mutate(
    PrOptimal = 1/(1+exp(-lambda*absDiff))
  )
)
(
  ggplot(data=lambdaCheck,aes(x=lambda,y=PrOptimal,group=absDiff))
  +geom_path()
  +theme_bw()
  +scale_x_continuous(trans="log10")
  +xlab("\u03bb")+ylab("Probability of choosing the action with the higher payoff")
)
```

It looks to me like most of the interesting stuff happens at $\lambda < 0.1$. Since λ must be positive, I will use a log-normal prior, which gives me two parameters to play with, the location and scale:[16]

$$\log \lambda \sim N(m_\lambda, s_\lambda^2)$$

I will calibrate this prior so that the 95th percentile of λ is 0.1, and the 5th percentile of λ is 0.0001. The system of equations to solve this is:

[16] This also helps with my observation that choice precision happens roughly on a log scale. We can express the distribution as a mean of the logged variable, and a relative scale of the distribution.

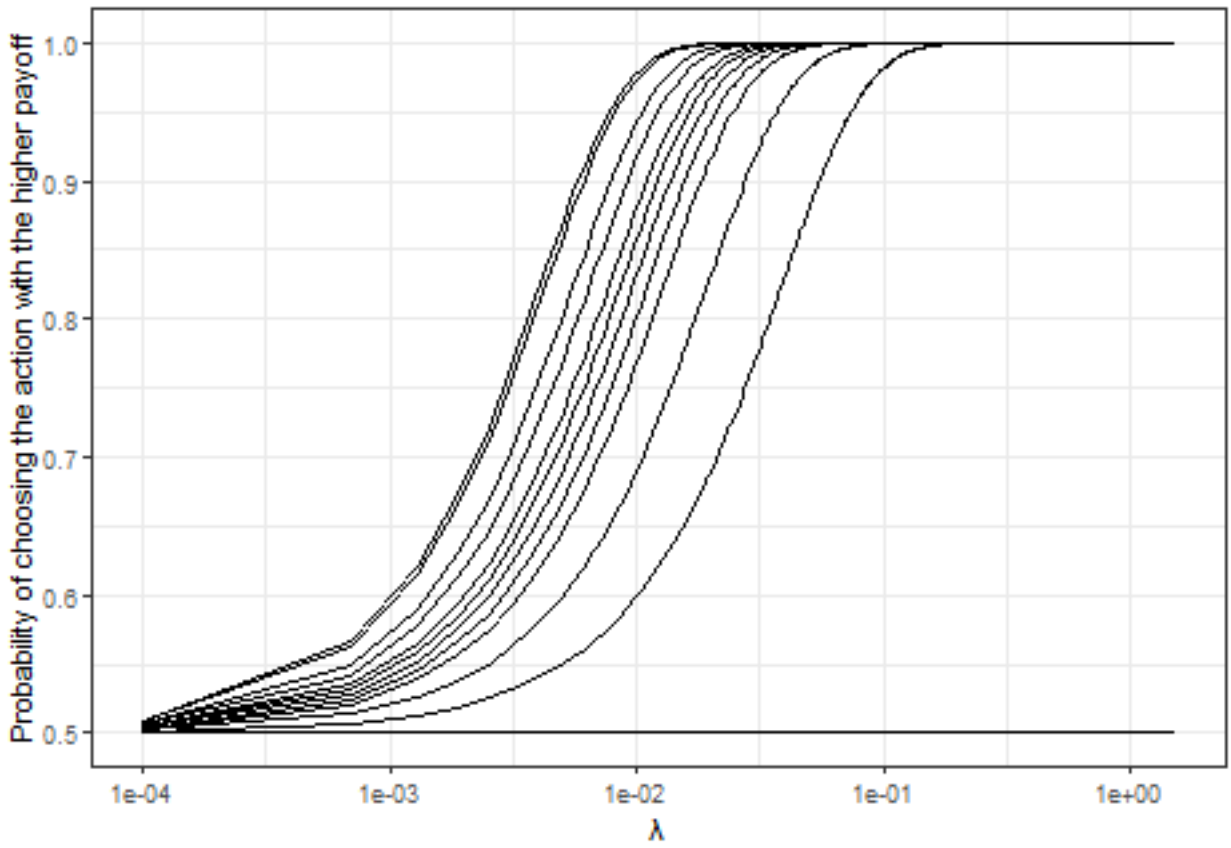


Figure 22: Probability of choosing the optimal action for selfish decison-makers as a function of choice precision λ. Each curve shows the model's prediction for a different payoff difference from the experiment.

$$
\begin{aligned}
\log 0.1 &= m_\lambda + 1.64 s_\lambda \\
\log 0.0001 &= m_\lambda - 1.64 s_\lambda \\
\log(0.1) - \log(0.0001) &= 2 \times 1.64 s_\lambda \\
s_\lambda &= \frac{\log(0.1) - \log(0.0001)}{2 \times 1.64} \approx 2.11 \\
m_\lambda &= \log 0.1 - 1.64 \times 2.11 \approx -5.76
\end{aligned}
$$

That leaves us with assigning priors to α and β. I will lump these together and use the same prior for both parameters. I think this simplification of the problem is OK because we can interpret both of these as marginal utilities. Here I note that if either parameter is greater than one in absolute value, then the participant will respond very strongly to inequality (or efficiency, or something else, depending on their signs). Let's place 75% of the prior probability mass of both parameters in the range $(-1, 1)$, which means that our full prior distribution is:

$$
\begin{aligned}
\alpha &\sim N(0, 0.67^2) \\
\beta &\sim N(0, 0.67^2) \\
\log \lambda &\sim N(-5.76, 2.11^2)
\end{aligned}
$$

5.1.5 Estimating the model for one participant

Here is the *Stan* program I wrote to estimate this model. Note that you could have generated the transformed data before passing it to *Stan* and got the same results. I just wanted to have the program take the data as close as possible to how it is on the journal website.

```
data {
  int<lower=0> n; // number of observations
  vector[n] self_x; // payoff to self with allocation x
  vector[n] other_x; // payoff to other with allocation x
  vector[n] self_y; // payoff to self with allocation y
  vector[n] other_y; // payoff to other with allocation y
  int choice_x[n]; // =1 iff allocation x is chosen

  real prior_lambda[2];
  real prior_alpha[2];
  real prior_beta[2];
}

transformed data {
  vector[n] dX; // disadvantageous inequality at allocation x
  vector[n] dY; // disadvantageous inequality at allocation y
  vector[n] aX; // advantageous inequality at allocation X
  vector[n] aY; // advantageous inequality at allocation X

  dX = fmax(0,other_x-self_x);
  dY = fmax(0,other_y-self_y);
  aX = fmax(0,self_x-other_x);
  aY = fmax(0,self_y-other_y);
}
parameters {
  real alpha;
  real beta;
  real<lower=0> lambda;
```

```
}
model {

  vector[n] Ux; // utility of allocation x
  vector[n] Uy; // utility of allocation y

  Ux = self_x-alpha*dX-beta*aX;
  Uy = self_y-alpha*dY-beta*aY;

  // likelihood
  choice_x ~ bernoulli_logit(lambda*(Ux-Uy));

  // priors
  alpha~normal(prior_alpha[1],prior_alpha[2]);
  beta ~normal(prior_beta[1],prior_beta[2]);
  lambda ~ lognormal(prior_lambda[1],prior_lambda[2]);

}
```

Before moving on, I want to draw your attention to the vectorization used in this program. Specifically, since I have the data variables `self_x`, `dX`, and `aX` stored as vectors, and the model's parameters `alpha` and `beta` are just real numbers (i.e. scalars), I can calculate the entire utility of Option X (and likewise for Option Y) in one line without looping over all observations. That is, the following two methods for calculating the utility of Option X will produce the same result:

```
// the vectorized method
Ux = self_x-alpha*dX-beta*aX;
```

```
// a for loop
for (ii in 1:n) {
  Ux[ii] = self_x[ii]-alpha*dX[ii]-beta*aX[ii];
}
```

However the second method will take longer. This is because *Stan* is geared towards vectorized calculations and matrix operations. The second option may seem more intuitive to you (it does to me), but I encourage you to see the matrix and vectorize things where you can. It's not going to same much time for this model, but as we start estimating more complicated models every bit of speed-up helps.

Now, let's estimate the model for the first participant

```
library(rstan)
rstan_options(auto_write = TRUE)
options(mc.cores = parallel::detectCores())

file<-"RepAgentModels/RepAgentModels_BFS2019"

# only run if I haven't estimated it yet
if (!file.exists(paste0("Outputs/",file,"_subj1.rds"))) {
  ii<-102010050706
  d<-D |> filter(sid==ii)
  dStan<-list(
    n=dim(d)[1],
    self_x=d$self_x,
    other_x=d$other_x,
    self_y=d$self_y,
    other_y=d$other_y,
```

Table 5: Estimates from the first participant in @BFS2019

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%
alpha	0.0889337	0.0038723	0.1932936	-0.2584474	-0.0372583	0.0778896	0.2045905	0.5094519
beta	0.1377641	0.0035096	0.1816480	-0.2167278	0.0241048	0.1406289	0.2529966	0.4902746
lambda	0.0072245	0.0000376	0.0018702	0.0038158	0.0058960	0.0071869	0.0084521	0.0110631
lp__	-40.8879873	0.0382728	1.4314119	-44.5129399	-41.5399276	-40.5390461	-39.8432989	-39.2968401

```
    choice_x=d$choice_x,

    # Stan (and R for that matter) deal with standard deviations,
    # not variances for a normal distribution
    prior_lambda = c(-5.76,2.11),
    prior_alpha=c(0,0.67),
    prior_beta = c(0,0.67)
  )
  Fit<-stan(paste0("Code/",file,".stan"),
            data=dStan,
            seed=42)
  saveRDS(Fit,paste0("Outputs/",file,"_subj1.rds"))
}

Fit<-readRDS(paste0("Outputs/",file,"_subj1.rds"))

rstan::summary(Fit)$summary %>% knitr::kable(caption="Estimates from the first participant in @BFS2019")
```

Table 5 shows the estimates summary, which is not too dissimilar from the point estimates from the logit above (there is lots of uncertainty in both kinds of estimate, so take into consideration the standard errors as well).

```
sim<-tibble(alpha = rstan::extract(Fit)$alpha,beta=rstan::extract(Fit)$beta)
(
  ggplot(sim,
         aes(x=alpha,y=beta))
  +geom_point(size=0.3,alpha=0.1)
  +geom_vline(xintercept=0,linetype="dashed")
  +geom_hline(yintercept=0,linetype="dashed")
  +theme_bw()
  +xlab(expression(alpha))+ylab(expression(beta))
)
```

Figure 23 shows the 4,000 posterior draws from the posterior distribution of α and β. We can use these draws to work out a posterior probability that the parameters lies in each quadrant of this plot, which can help us assign a posterior probability to this participant having the parameter restrictions imposed by Fehr and Schmidt (1999), which are:

- $\alpha > 0$ and $\beta > 0$, so the participant is averse to inequality,
- $\alpha > \beta$, so the participant dislikes being behind (aversion to disadvantageous inequality) more than being ahead, and
- $\beta \in (0, 1)$, so the participant would not be willing to burn money in order to equalize payoffs

To compute the probability that all of these are true, we simply compute the fraction of posterior draws that satisfy these conditions:

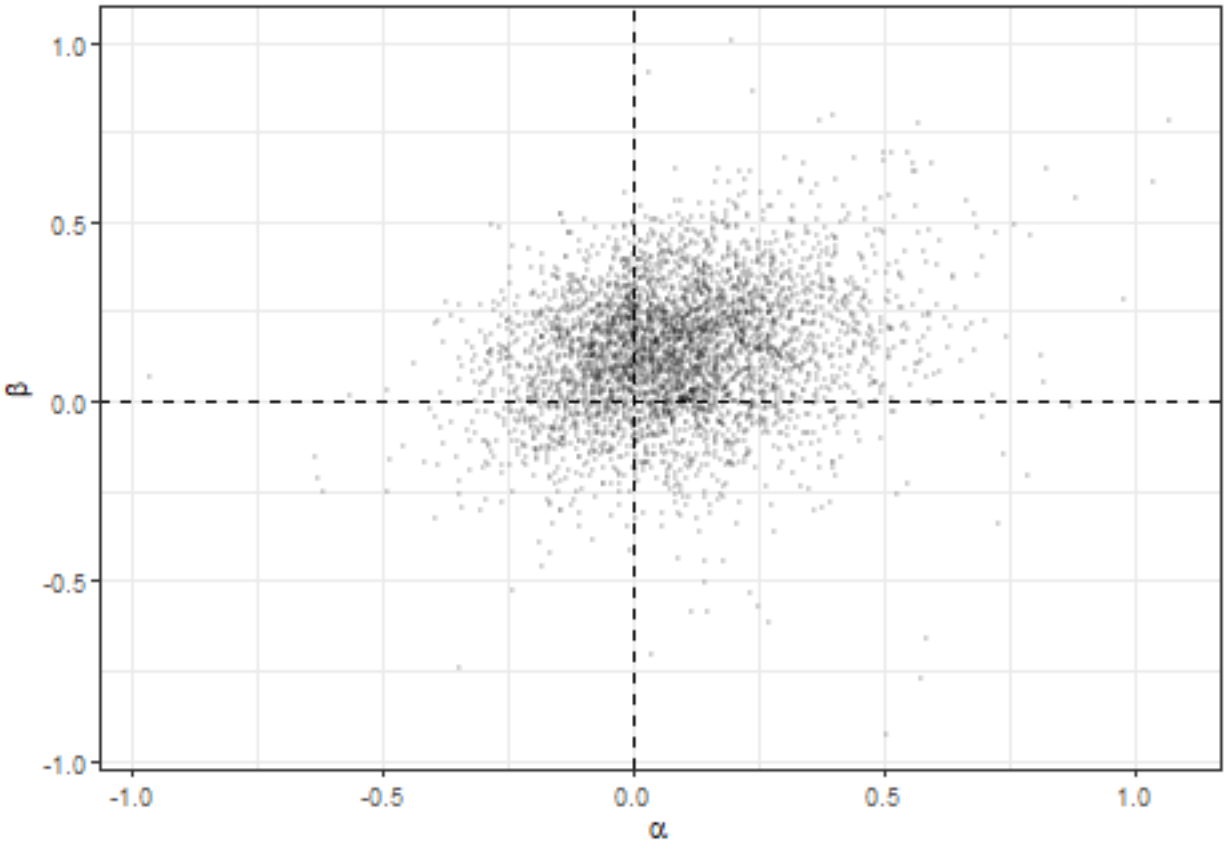


Figure 23: Posterior draws for the first participant of Bruhin, Fehr, and Schunk (2019).

```
sim |> summarize(
  FS1999 = mean(
    alpha>0 & beta>0 & alpha>beta & beta<1
  )
)
```

```
## # A tibble: 1 x 1
##   FS1999
##    <dbl>
## 1  0.233
```

Since I centered the prior distribution on zero for both of these parameters, 23% is not too different from 25%, the prior probability that these conditions are met. Therefore I would not get too excited about this probability. Furthermore, since we have 4,000 posterior draws, the prior standard deviation of this probability is at least:[17]

$$\sqrt{\frac{0.25 \times 0.75}{4000}} \approx 0.007$$

So getting 4,000 draws from the *prior* that implied that this probability was 23% would not be too unlikely. All of this is to say, we actually don't learn much about whether this parameter restriction is relevant for this participant. We actually do resolve quite a bit of parameter uncertainty, though. For example, Figure 24 shows both the prior and posterior distributions of α. We have actually learned quite a bit from these 78 choices, but it is more that we have rules out large values of α, rather than ruled out everything on one side of zero. As such, the uncertainty in whether or not $\alpha > 0$ is largely unchanged.

```
(
  ggplot()
  +geom_density(data=sim,aes(x=alpha,linetype="posterior"))
  +geom_path(data=tibble(alpha=seq(-2,2,length=1000)) |> mutate(den=dnorm(alpha,mean=0,sd=0.67)),aes(x=a
  +theme_bw()
  +xlab(expression(alpha))+ylab("density")
)
```

5.1.6 Estimating the model for all participants

Of course, once we have a model that we can estimate for one participant, we can also use it with the others. So here we go:[18]

```
if (!file.exists(paste0("Outputs/",file,"_ALL.rds"))) {

  # create an empty dataset to dump the posterior draws into
  ESTIMATES<-tibble()

  for (ii in (D$sid |> unique())) {
    print(ii)
    d<-D|> filter(sid==ii)

    dStan<-list(
    n=dim(d)[1],
    self_x=d$self_x,
    other_x=d$other_x,
```

[17] Since the posterior draws come from a Markov Chain, they will likely be correlated, so the standard error on this probability assuming that the draws are independent will likely be an over-estimate.

[18] This one may take a while, so put on a pot of coffee, get off the plateau, or something.

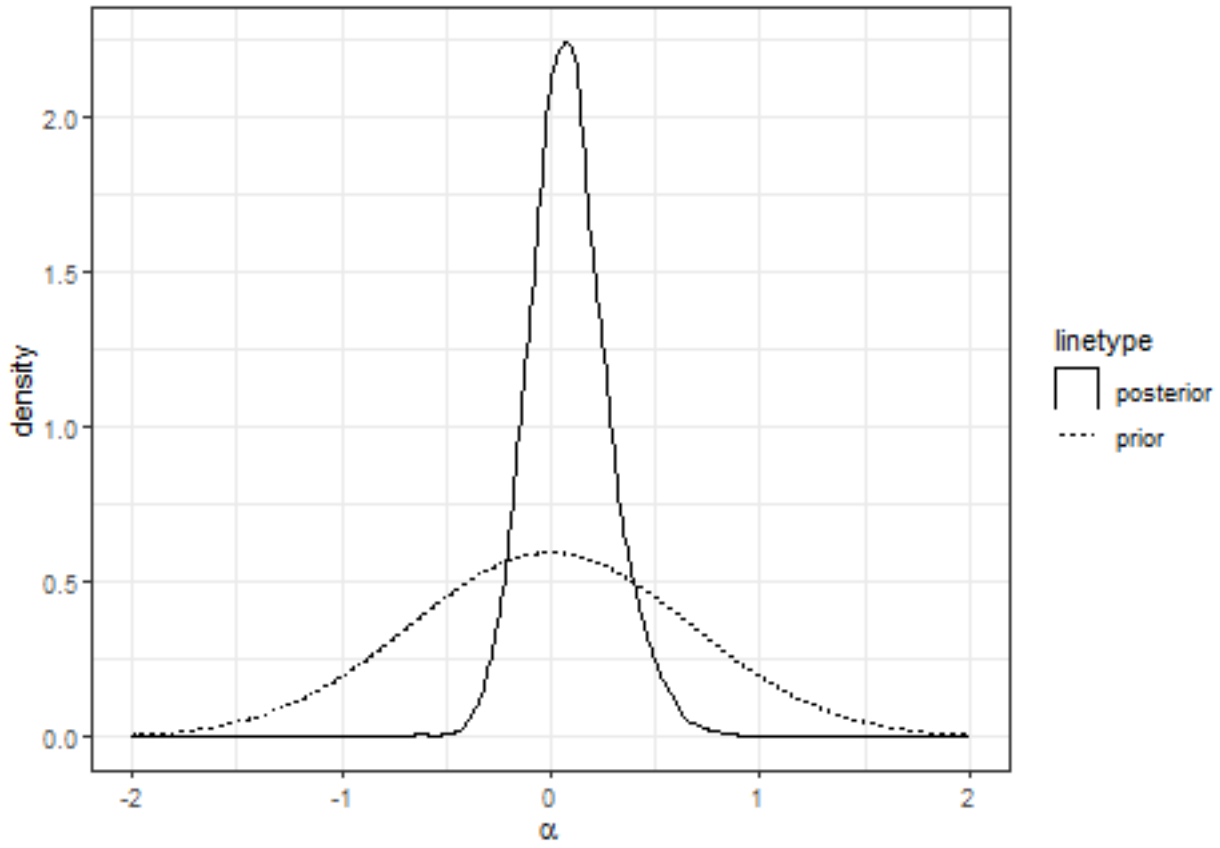


Figure 24: Prior and posterior distributions of parameter α for the first particiapnt in Bruhin, Fehr, and Schunk (2019).

```
    self_y=d$self_y,
    other_y=d$other_y,
    choice_x=d$choice_x,

    # Stan (and R for that matter) deal with standard deviations,
    # not variances for a normal distribution
    prior_lambda = c(-5.76,2.11),
    prior_alpha=c(0,0.67),
    prior_beta = c(0,0.67)
  )
  Fit<-stan(paste0("Code/",file,".stan"),
            data=dStan,
            seed=42,
            # Here I am running into a few "divergent transitions"
            # errors. One of the solutions to this is to increase
            # the "adapt_delta" control parameter
            control=list(adapt_delta=0.999)
            )

  tmp<-tibble(lambda = extract(Fit)$lambda,
              alpha  = extract(Fit)$alpha,
              beta   = extract(Fit)$beta,
              sid    = ii
              )
  ESTIMATES<-rbind(ESTIMATES,tmp)

  }

  saveRDS(ESTIMATES,paste0("Outputs/",file,"_ALL.rds"))
}
```

So that's all very well and good, but what do we *do* with all of these estimates? We don't have the time to look at a summary output table for each one of them individually, so things are best summarized in pictures. What I will do here is wrangle the posterior draws into posterior means and 95% Bayesian credible regions,[19] and also the probability that the Fehr and Schmidt (1999) parameter restriction holds.

```
ESTIMATES<-readRDS(paste0("Outputs/",file,"_ALL.rds"))

# a summary of estimates that I could have got from summary(Fit)$summary
EstSum<-(
  ESTIMATES
  |> pivot_longer(cols=c(lambda,alpha,beta),values_to="value")
  |> group_by(sid,name)
  |> summarize(
    mean = mean(value),
    p025 = quantile(value,probs=0.025),
    p975 = quantile(value,probs=0.975)
  )
)

# Probability that the FS restriction holds
ProbFS1999<-(
```

[19] you can also get these from the summary tables *RStan* can give you, but since I wanted to compute the probability that the Fehr and Schmidt (1999) parameter restriction holds, I needed the whole distribution.

```
  ESTIMATES
  |> group_by(sid)
  |> summarize(PrFS = mean(
    alpha>0 & beta>0 & alpha>beta & beta<1
    )
  )
)
```

First, let's look at the probability that the Fehr and Schmidt (1999) restriction holds. As before this is just a matter of computing the fraction of times the posterior draws land in the right region. The empirical cumulative density of these posterior probabilities is shown in Figure 25.[20] Here we can see a large accumulation of posterior probabilities near zero. About 60% of participants have so few posterior draws of α and β in the region implied by the Fehr and Schmidt (1999) restriction that we cannot distinguish between zero and the actual posterior probability. As such, there are in fact very few participants whose parameters are likely to conform to the Fehr and Schmidt (1999) restriction.

```
(
  ggplot(ProbFS1999,aes(x=PrFS))
  +stat_ecdf()
  +theme_bw()
  +xlab("Posterior probability that the FS restriction on \n parameters holds")
  +ylab("empirical cumulative density")
)
```

Now let's try to visualize the joint distribution of participants' parameters. It can get really messy in these plots if you try to show a credible region around the point estimates, so here I will just plot the posterior means. These are shown in Figure 26. The majority of these point estimates fall in the efficiency-loving region ($\alpha < 0$, $\beta > 0$), and within this most estimates have $\beta > |\alpha|$, meaning that these participants care more about increasing the other person's earnings when the other person is earning less than themself. The majority of parameters are also less than one in absolute value, meaning that they place more weight on their own earnings than they do on the earnings of others. The obvious exception to this is for three competitive and inequality averse participants with $\alpha > 1$. For these participants, their marginal utility of own wealth is less than their marginal utility of the other person's wealth when the other person is earning more. There are no particiapnts whose posterior mean estimates are in the inequality loving quadrant of this plot.

```
regions<-tibble(
  alpha = c(1,-0.25,-0.25,1),
  beta = c(0.5,-0.5,0.8,-0.5),
  label = c("inequality averse",
            "inequality loving",
            "efficiency loving",
            "competitive"
            )
)

pltInd<-(
  ggplot(
    data = (ESTIMATES |> group_by(sid)
                      |> summarize(alpha = mean(alpha),
                                   beta = mean(beta),
                                   lambda = mean(lambda))
```

[20] As I often visualize my results as empirical cumulative distribution functions (ECDF), it may help here to remember that steep parts of a cumulative density function show regions of large probability mass or density, while flat parts are regions where the variable is less likely to occur.

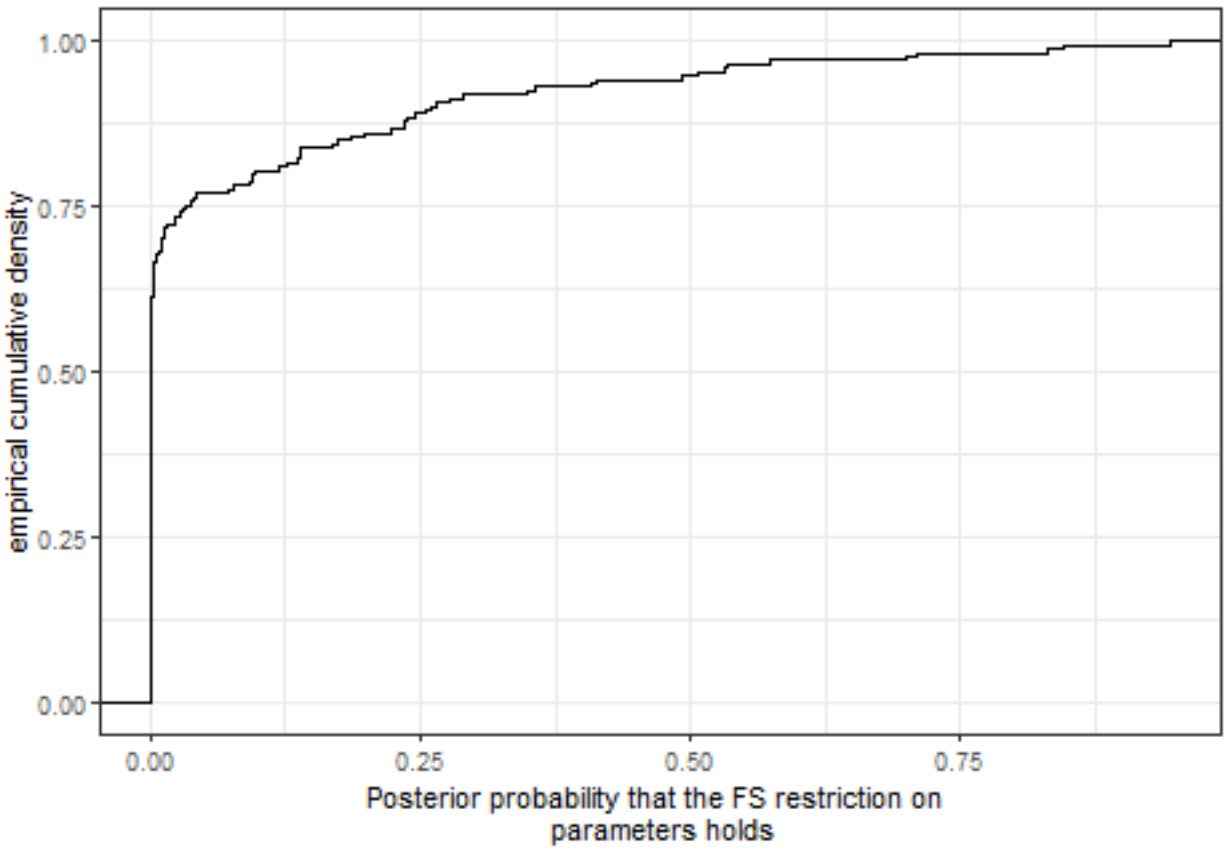


Figure 25: Posterior probability that the Fehr and Schmidt (1999) restrictions hold for each participant.

```
            ),
    aes(x=alpha,y=beta,color=log10(lambda))
  )
  +geom_point()
  +geom_text(data=regions,aes(x=alpha,y=beta,label=label),color="black")
  +theme_bw()
  +xlab(expression(alpha))+ylab(expression(beta))
  +geom_hline(yintercept=0,linetype="dashed")
  +geom_vline(xintercept=0,linetype="dashed")
  #+geom_abline(slope=c(-1,1),intercept=0,linetype="dotted")
)
pltInd |> print()
```

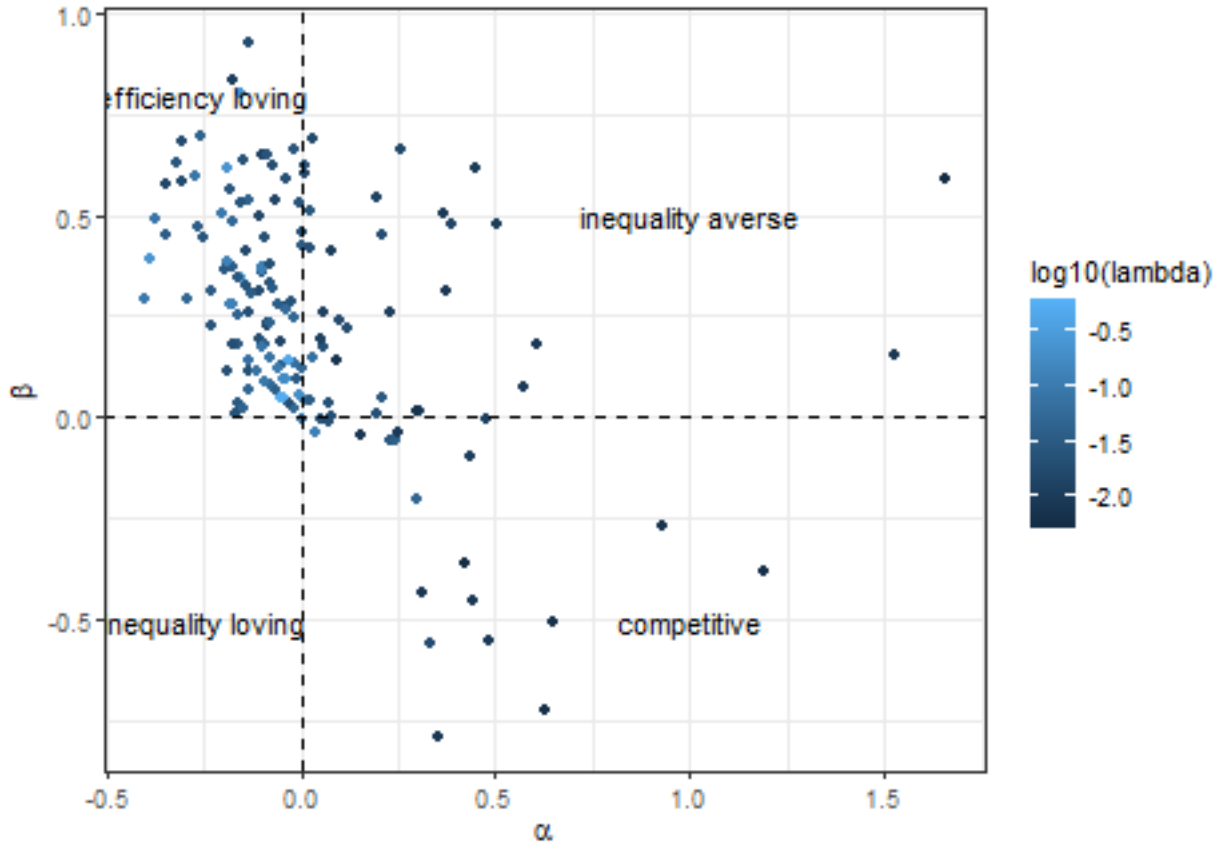


Figure 26: Posterior mean estimates for Fehr and Schmidt (1999) model using data from Bruhin, Fehr, and Schunk (2019). Dashed lines show boundaries for the classification of types of preferences.

Finally, if we want a deeper dive into the parameters individually, we can plot the empirical cumulative density function of the posterior means, overlaying them with their 95% Bayesian credible regions to show how precise the estimates are. This is done in Figure 27. I like these plots both for individual estimation and for hierarchical models because they show us a lot of information about the between-participant heterogeneity in the parameter estimates, as well as giving us an idea about how precise our estimates are. Here we can see that there is substantial heterogeneity both in the posterior mean estimates (seen in the empirical cumulative density functions), as well as in the precision of the estimates (seen in the error bars).

```
(
  ggplot(data=(EstSum
               |> filter(name!="lambda")
               |> group_by(name)
               |> mutate(rank=rank(mean)/length(mean))
               )
  )
  +facet_wrap(~name,scales="free")
  +stat_ecdf(aes(x=mean))
  +geom_errorbar(aes(y=rank,xmin=p025,xmax=p975),alpha=0.5)
  +theme_bw()
  +ylab("empirical cumulative density")
  +xlab("posterior estimate")
)
```

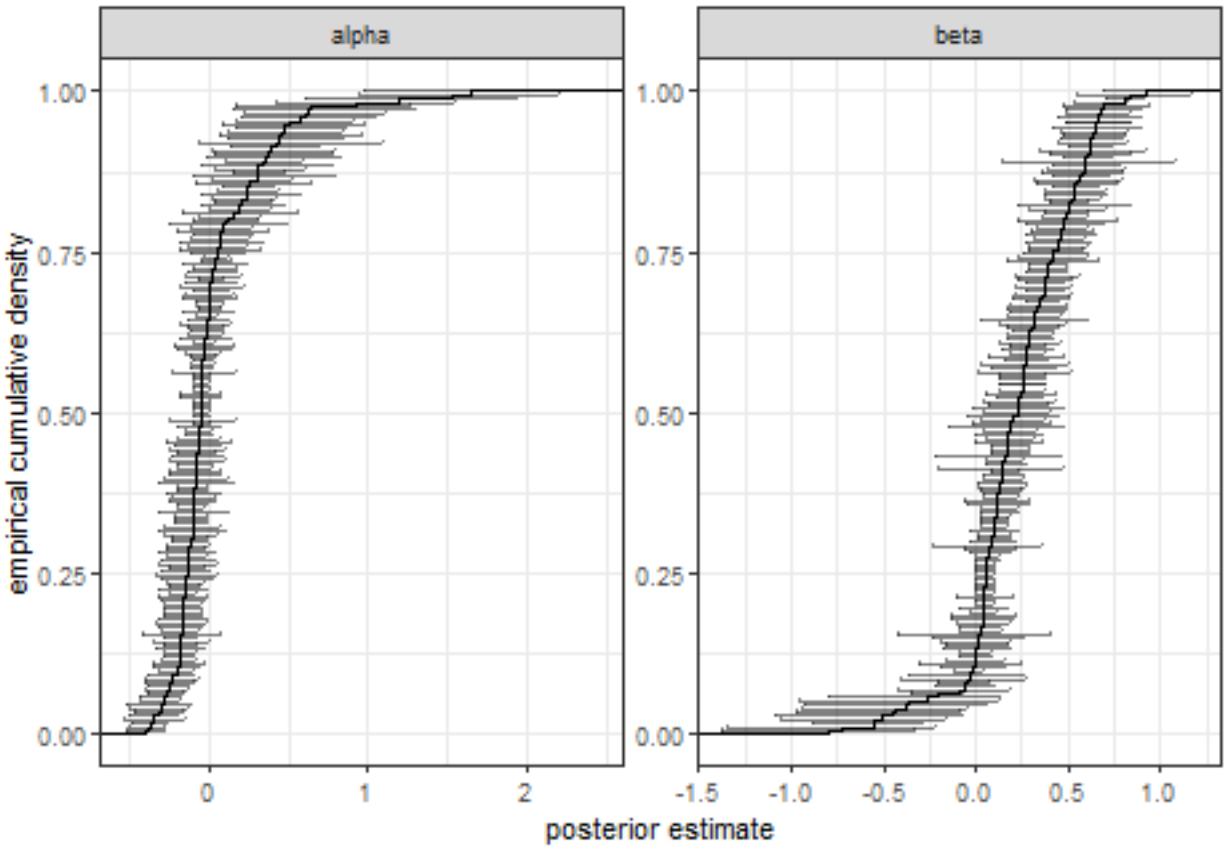


Figure 27: Empirical cumulative density functions of posterior mean estimates for α and β. Error bars show 95% Bayesain credible regions for the parameter.

5.1.7 But we could be learning more!

Hopefully the above analysis is leaving you somewhat disappointed. OK, we have estimated some individual-specific parameters, but they are still quite imprecise. From a practical perspective, why even bother doing Bayesian estimation for this when there is a pre-packaged Frequentist logit function in every statistical

package?[21] Perhaps more annoyingly (for me at least), we haven't yet got a way of estimating a *distribution* of parameters and then commenting on the population of these preferences in any meaningful way. We might have been able to do some scatter plots of our parameters and made qualitative comments about any patterns or clusters that we might see, but what we really need is a way to organize the data of the whole experiment, rather than just for one participant at a time. This will be the focus especially of the next chapter (hierarchical models), and many others to follow will build on this. For the remainder of this chapter, though, we will continue to learn a bit more about representative agent models.

5.2 Actual representative agent models (pooled models)

We can always take our representative agent model to our entire dataset. The real question is: *should* we? As I alluded to in the previous section, the representative agent does not exist unless our models meet oddly specific criteria. Therefore, it is likely that the results we get from pooling all of our data in this way will be misleading. One big concern is heterogeneity bias: models, especially nonlinear models, can be biased in favor of one conclusion over another if the underlying parameters are heterogeneous at the participant level. See Wilcox (2006) for an excellent example of this. Furthermore, pooled models may overstate the certainty we have in our estimates. A pooled model does not understand that we have n participants each making T decisions. Instead it "thinks" we have nT independent observations.

However there are probably diminishing returns to added heterogeneity in a model, and sometimes the optimal amount of heterogeneity might be zero. For example a simple, portable model of Quantal Response Equilibrium (McKelvey and Palfrey 1995) may organize data in games well enough to provide some insights and make useful predictions. If understanding and quantifying heterogeneity is not one's top priority, then models with less or no heterogeneity, but perhaps more complex in other dimensions, may be more useful.

For the purposes of continuing this example, I will now estimate the pooled model for the Bruhin, Fehr, and Schunk (2019) data. I don't expect it will provide any more insight than the participant-specific estimates from the previous section, but hopefully it will provide some insight into *why* it is not going to be particularly useful.

```
if (!file.exists(paste0("Outputs/",file,"_pooled.rds"))) {
  dStan<-list(
    n=dim(D)[1],
    self_x=D$self_x,
    other_x=D$other_x,
    self_y=D$self_y,
    other_y=D$other_y,
    choice_x=D$choice_x,

    # Stan (and R for that matter) deal with standard deviations,
    # not variances for a normal distribution
    prior_lambda = c(-5.76,2.11),
    prior_alpha=c(0,0.67),
    prior_beta = c(0,0.67)
  )
  Fit<-stan(paste0("Code/",file,".stan"),
            data=dStan,
            seed=42)
  saveRDS(Fit,paste0("Outputs/",file,"_pooled.rds"))
}

Fit<-readRDS(paste0("Outputs/",file,"_pooled.rds"))
```

[21] One of the reasons is that for other economic models, the following probabilistic model does not lend itself to a simple representation in our usual reduced-form toolbox of econometric techniques. Once these models become nonlinear, you *will* have to code them up yourself. Hopefully through this book I can show you that there is not much more effort in doing this within a Bayesian framework.

Table 6: Estimates pooling all data in @BFS2019

	mean	se_mean	sd	2.5%	25%	50%	75%	
alpha	-0.0505990	0.0001204	0.0076189	-0.0651262	-0.0558035	-0.0505947	-0.0453162	
beta	0.2622539	0.0001331	0.0076152	0.2471614	0.2572358	0.2622196	0.2673445	
lambda	0.0155982	0.0000042	0.0002505	0.0151177	0.0154292	0.0155959	0.0157644	
lp___	-4022.7625528	0.0295697	1.2356555	-4025.9640038	-4023.3252946	-4022.4349407	-4021.8617346	-40

```
rstan::summary(Fit)$summary |> knitr::kable(caption="Estimates pooling all data in @BFS2019")
```

These estimates are summarized in Table 6. Especially note that the posterior distributions of α and β are *much* more precise than any of the distributions we get from the individual estimates: the posterior standard deviations are two orders of magnitude smaller. This is because have 174 times as many observations compared to when we were doing the participant-specific estimates. Our model thinks that every single decision is a statistically independent observation, and we have a lot of decisions! In reality there is obvious dependence that we are not accounting for within the data for each participant, evidenced by the large variety of parameter estimates. As such, we should not be as confident in those estimates as the posterior standard deviations would have us believe.

6 Hierarchical models

6.1 A random sample of participants walks into your lab

Suppose that your goal was to estimate a parameter θ_i for every participant in your experiment. In an ideal world, they would come into your laboratory with θ_i stamped to their face, and all you would have to do to know it perfectly would be to briefly look at them, pay them a small show-up fee, take note of their θ_i, and send them on their way. Such ideal participants are a pipe dream, but what we *can* do is ask them to make decisions y_i that reveal to us something about their θ_i. If we have a probabilistic model that relates y_i to θ_i, and we have a prior for θ_i, then we can learn about θ_i. But i is not the only participant who comes to the lab. j, k, and l do as well. In fact, we are fortunate enough to get n of them! Is there anything we can learn about θ_i by studying the data from participants j, k, and l? The answer to this is a definitive "yes", if θ_i, θ_j, θ_k, and θ_l are all drawn from the same distribution. In economic experiments this is a reasonably good assumption, as participants are typically drawn from the same subject pool.

So *how* do we learn about θ_i from j's decisions? We note that θ_i and θ_j are both draws from the same distribution, and so decisions y_j are informative of not just θ_j, but the parameters governing this *population-level distribution*, call them $\mathcal{P}$. That is, knowing $\mathcal{P}$ helps us learn about θ_j because $\mathcal{P}$ is the prior we should have on θ_i, and y_j tells us something about θ_j, which in turn tells us something about $\mathcal{P}$. So even if we don't care directly about $\mathcal{P}$, j's decisions can help us learn about i's parameter, *if we let them*. Furthermore, learning about population-level parameters $\mathcal{P}$ might be more interesting to us as researches than whatever is going on at the individual level. These parameters allow us to extrapolate beyond the sample of participants we get, and make statements about our entire subject pool.

This is where the benefits of Bayesian estimation really start to be realized!

Up to this point we have focused solely on representative agent models and participant-specific models. All of this work has not been a waste, as these models are important stepping stones on the way to richer models, but it is these richer models that provide deeper insight into our data. In this chapter, we will be estimating *population-level parameters*. These are parameters that govern the distribution of (what are usually) our participants' individual parameters. That is, if our economic model is about preferences, we will be estimating the *distribution* of preferences. This is an amazingly powerful tool.

We will do this by assuming that our data follows a *hierarchical* structure. By that I mean that we will assume that we can partition our data into chunks, and maybe even that we can further partition these chunks into

sub-chunks. We will then add some structure to how these chunks are related to each other, and estimate these relationships. What we get out of this is a quantification of the heterogeneity that exists between these chunks, and if we have more than one level (i.e. chunks and sub-chunks), the relative contribution to heterogeneity of each level.

6.2 The anatomy of a basic hierarchical model

In the previous chapter, we estimated a participant's parameters using just the data generated by that participant. Specifically, we focused on just their own data y_i, while ignoring the data produced by all our other participants y_{-i}. Just like these participant-specific models, we will continue to have a probabilistic model for the participant's behavior, which we will denote as the likelihood $p(y_i \mid \theta_i)$. However now we will further specify a *distribution* for θ_i. That is, we will assume that each participant's θ_i is an independent draw from the population of potential participants:

$$\theta_i \sim iidG(\mathcal{P})$$

where G is the family of distribution we have chosen to model how the θ_is are generated, and $\mathcal{P}$ are the parameters in the distribution G. For example, probably the most popular hierarchical specification assumes that θ_i are drawn from a multivariate normal distribution, so $\mathcal{P} = (\mu, \Sigma)$ are the mean vector and variance-covariance matrix of this distribution. I will discuss the multivariate normal hierarchical model in much more detail below.

If we were lucky enough to just observe θ_i, then it would be a straightforward task estimating $\mathcal{P}$. Instead though, we get data that are generated according to:

$$y_i \sim g(\theta_i)$$

where g is the family of distribution that we have chosen to model how our data y_i are generated conditional on our participant's parameter θ_i. Putting this together, we observe y_i, which tells us something about θ_i, which tells us something about $\mathcal{P}$.

Note that all we need to simulate this data-generating process is to specify $\mathcal{P}$. That is, we can:

1. Draw $\theta_i \sim G(\mathcal{P})$, then
2. Draw $y_i \sim g(\theta_i)$

Therefore, our likelihood for a participant's data can be defined as follows:

$$\begin{aligned} p(y_i, \theta_i \mid \mathcal{P}) &= p(y_i \mid \theta_i, \mathcal{P}) p(\theta_i \mid \mathcal{P}) \\ &= p(y_i \mid \theta_i) p(\theta_i \mid \mathcal{P}) \end{aligned}$$

where the second line follows if we assume that y_i and $\mathcal{P}$ are independent conditional on θ_i. In plainer English: once we know θ_i, $\mathcal{P}$ will tell us no more information about y_i. The power of this assumption is that it allows us to think separately about how our data are generated given the participant's parameters, (i.e. $p(y_i \mid \theta_i)$), and how participants are distributed across the population (i.e. $p(\theta_i \mid \mathcal{P})$).

Conceptually, the hierarchical structure (a distribution for θ_i) is hopefully easy to grasp. However we have a practical problem: we still don't know θ_i, so any expression that includes it cannot be feasibly used to evaluate our model's likelihood. The next section will develop two techniques for doing this. The first, integrating the likelihood, is popular for Frequentist techniques, but may sometimes be useful in Bayesian applications too; and the second, data augmentation, is the typical Bayesian go-to for this problem, and lends itself especially to the Hamiltonian Monte Carlo of *Stan* (Betancourt and Girolami 2015).

6.3 Accounting for unobserved heterogeneity

One of the hurdles I came accross when I started learning about hierarchical models was how to treat unobservable heterogeneity. For most of the models discussed in this book, this unobservable heterogeneity will mostly be in participant-specific parameters. This problem is not specific to Bayesian implementations of hierarchical models, but it is typically addressed differently depending on whether one adopts a Bayesian or Frequentist (maximum likelihood) approach. In both cases the models can make the same distributional assumptions about θ, but they are typically treated in very different ways.

While maximum likelihood techniques typically integrate out the unobserved heterogeneity θ and just focus on estimating the population-level parameters $\mathcal{P}$, the Bayesian implementation typically jointly estimates θ alongside $\mathcal{P}$. I include an introduction to both techniques, as I believe they can both be useful tools that might be the best option to you in some cases.

6.3.1 The last time you will integrate the likelihood, probably

The typical maximum likelihood implementation of a hierarchical model uses maximum simulated likelihood to eliminate the participant-level parameters θ from the likelihood function. This technique in the context of economic experiments is discussed in Chapter 10 of Moffatt (2015), and is the implementation used in the hierarchical models estimated in, for example, Conte, Hey, and Moffatt (2011).

The likelihood of observing data y_i conditional on only population-level parameters $\mathcal{P}$ can be approximated as:

$$\begin{aligned} p(y_i \mid \mathcal{P}) &= \int_{\Theta} p(y_i, \theta \mid \mathcal{P}) \mathrm{d}\theta \\ &= \int_{\Theta} p(y_i \mid \theta, \mathcal{P}) p(\theta \mid \mathcal{P}) \mathrm{d}\theta \\ &= E_{\theta|\mathcal{P}} \left[p(y_i \mid \theta, \mathcal{P}) \right] \\ &\approx \frac{1}{S} \sum_{s=1}^{S} p(y_i \mid \theta^s, \mathcal{P}), \quad \text{with: } \theta^s \sim iidG(\mathcal{P}) \end{aligned}$$

where the first equality follows because of the relationship between the marginal distribution $y_i \mid \mathcal{P}$ and the joint distribution $y_i, \theta \mid \mathcal{P}$. The second inequality writes this joint distribution function as the product of a conditional and a marginal distribution.[22] The third equality follows by recognizing that the expression in the second line is the expectation of $p(y_i \mid \theta, \mathcal{P})$, where $\theta \mid \mathcal{P}$ is the only random variable, and the approximation is a Monte Carlo integration of this expectation. This approximation can be made more efficient using Halton draws, which are discussed in Section 10.3 of Moffatt (2015).

What all of this means is that you can approximate the likelihood of the entire dataset as:

$$p(y \mid \mathcal{P}) = \prod_{i=1}^{n} p(y_i \mid \mathcal{P}) \approx \prod_{i=1}^{n} \left(\frac{1}{S} \sum_{s=1}^{S} p(y_i \mid \theta^s, \mathcal{P}) \right)$$

So the log-likelihood function, which is what we would pass to *Stan*, is approximated as:

$$\log p(y \mid \mathcal{P}) \approx \sum_{i=1}^{n} \log \left(\frac{1}{S} \sum_{s=1}^{S} p(y_i \mid \theta^s, \mathcal{P}) \right)$$

[22] I.e. $p(A, B) = p(A \mid B) p(B)$

6.3.2 Data augmentation

The other technique for accounting for unobserved heterogeneity is called "data augmentation", and basically comes down to jointly estimating the participant-specific parameters alongside the population-level parameters. This means specifying a posterior distribution in the following form:

$$\begin{aligned} p(\theta, \mathcal{P} \mid y) &\propto p(y \mid \theta, \mathcal{P}) p(\theta, \mathcal{P}) \\ &= p(y \mid \theta, \mathcal{P}) p(\theta \mid \mathcal{P}) p(\mathcal{P}) \end{aligned}$$

Note here the trade-off: we do not have to use Monte Carlo integration to approximate the likelihood (because we are jointly estimating θ), but we now have a whole lot more parameters to estimate! Specifically, while we do not have to evaluate each likelihood S times for the Monte Carlo integration, we have to estimate all of the parameters in θ, not just $\mathcal{P}$. As there will usually be at least one element in θ for every participant, this means that data augmentation can have us going from a few to many parameters very quickly.

Our final product is the posterior draws of $\mathcal{P}$ *and* θ, so not only do we have estimates of the population quantities, we also have estimates of the participants' parameters. These estimates are referred to as *shrinkage* estimates because compared to their participant-specific counterparts from the previous section, they will be "shrunk" or pulled towards the center of the population distribution implied by $\mathcal{P}$. This is a feature, not a bug: by jointly estimating $\mathcal{P}$ alongside θ, we are allowing our data to decide for us what an appropriate prior for θ is. In particular, this prior is the distribution implied by $\mathcal{P}$, which since it is informed by the data, will most likely be more informative than our prior for the participant-specific counterpart.

The computational benefit of this method is that we only have to evaluate the likelihood once, instead of S times if we were integrating it. My intuition for this, albeit only informed through a lot of trial and error trying both methods with both maximum likelihood and Bayesian techniques, is that these methods are common in their respective statistical philosophies for computational practicalities. Hamiltonian Monte Carlo, which is what *Stan* does, seems to scale well with the number of parameters in the model. Hence, adding a whole lot of participant-specific θs to the bill of parameters that we need to estimate is not that much of an additional computational burden, especially when the θs are well informed by the "prior" in the population distribution $\mathcal{P}$. On the other hand, maximum likelihood does not scale so well with the number of parameters, and so Monte Carlo integration of the likelihood function will seriously cut down on the computational burden relative to data augmentation. This is not to say that there will not be times when you will want to integrate the likelihood for your Bayesian estimator. Rather, I would recommend trying data augmentation first, but if there are some pathologies in your likelihood function that make this difficult, integrating the likelihood might be something that you could try.

6.4 A multivariate normal hierarchical model

Although there are many distributional families that you might want to use to model the distribution of your individual parameters, the multivariate normal distribution is a great place to start, and will get you a long way. For this, we will assume that our individual-level parameters θ_i follow a multivariate normal distribution:

$$\theta_i \sim iidN(\mu, \Sigma)$$

Where if θ_i is a $k \times 1$ vector of parameters specific to participant i, then μ is a $k \times 1$ mean vector of means, so:

$$\mu_j = E(\theta_{i,j})$$

and Σ is a $k \times k$ variance-covariance matrix: the diagonal elements of Σ therefore correspond to the variances of each element of θ_i, and the off-diagonal components tell us the covariances. This specification therefore permits correlation between the participants' parameters.

One thing you might be noticing about this specification (and the more general specification discussed above), is that the distribution in (6.4) almost looks like how we would specify a prior. This is certainly a helpful way to think about it: if we were instead estimating a participant-specific model using data from just that participant, $\theta_i \sim N(\mu, \Sigma)$ could exactly be our prior![23] However instead of choosing μ and Σ to express our belief about θ_i, we are instead assigning priors to μ and Σ, then estimating these parameters. As such, instead of specifying priors for θ_i, as we did with the participant-specific estimation, we will be *estimating* what an appropriate prior for θ_i should be.

6.4.1 Decomposing the variance-covariance matrix

While it may be somewhat intuitive to assign a prior for a population mean parameter μ, I always find it difficult thinking about priors for variance-covariance matrices. For me, this is difficult because I don't have good grasp of the implications of a prior for a matrix that must have some restrictions imposed on it (because it is a variance-covariance matrix). Fortunately, as documented in Section 1.13 of the *Stan User's Guide* (Stan Development Team 2022), not only can we decompose this matrix into a vector of standard deviations, and a correlation matrix, two quantities I find *much* easier to think about, but we can assign priors to these quantities separately.

Specifically, note that our variance-covariance matrix Σ can be decomposed into a $k \times 1$ vector τ of standard deviations, and a $k \times k$ correlation matrix Ω:

$$\Sigma = \text{diag}(\tau)\Omega\text{diag}(\tau)$$

I find this immensely useful, because it permits us to think about the prior for Σ in two stages:

1. Choose priors for the elements of τ to reflect our belief about the amount of heterogeneity in each element of θ_i.
2. Choose a prior for Ω to reflect our belief about how these parameters are correlated, without having to worry about the scale of these parameters.

For the priors for τ, the *Stan User's Guide* recommends a half Cauchy distribution. In the first example in this chapter, I discuss how we can use the cumulative density function of this distribution to work out how to assign appropriate priors.

For the correlation matrix Ω, the *Stan User's Guide* recommends an LKJ prior, which has the form:

$$\Omega \sim \text{LKJCorr}(\eta) \iff p(\Omega \mid \eta) \propto \det(\Omega)^{\eta-1}$$

In order to investigate what this means for a correlation between two parameters, I wrote a *Stan* program to draw from this distribution, then varied the parameter η and the matrix size k:

```
data {
  int<lower=0> k; // size of correlation matrix
  real prior_corr; // lkj prior on correlation matrix
}

parameters {
  corr_matrix[k] CORR;
}

model {
  CORR ~ lkj_corr(prior_corr);
}
```

[23] In fact, sometimes these are also referred to as "priors" in discussion of hierarchical models. I like to stay away from this language, as μ and Σ are parameters that we are trying to estimate, rather than parameters describing our prior beliefs about other parameters.

Then I simulated its distribution for a few values of k and η:

```
library(tidyverse)
library(rstan)
  rstan_options(auto_write = TRUE)
  options(mc.cores = parallel::detectCores())

# values to loop over
K<-c(2,3,4) # correlation matrix size
PRIOR_CORR<-c(0.25,0.5,1,2,4,8) # lkj prior parameter

CorrSim<-tibble()
for (k in K) {
  for (prior_corr in PRIOR_CORR) {
    print(paste(k,prior_corr))
    dStan<-list(k=k,prior_corr=prior_corr)
    Fit<-stan("Code/Hierarchical/lkj_corr.stan",data=dStan,seed=42)
    tmp<-tibble(
        prior_corr = prior_corr,
        k = k,
        corr = extract(Fit)$CORR[,1,2]
    )
    CorrSim<-rbind(CorrSim,tmp)

  }
}

saveRDS(CorrSim,"Outputs/Hierarchical/Hierarchica_lkj_corr.Rds")
```

As the LKJ correlation distribution is symmetric, we only need to look at one of the correlations to get the picture, so here it is:

```
library(tidyverse)
corr<-readRDS("Outputs/Hierarchical/Hierarchica_lkj_corr.Rds")
(
  ggplot(corr,aes(x=corr))
  +stat_density(alpha=0.7)
  +facet_grid(paste("eta =",prior_corr)~paste("k =",k))
  +theme_bw()
)
```

Figure 28 confirms main features of the LKH distribution, as outlined in Section 1.13 of the *Stan User's Guide*

> The basic behavior of the LKJ correlation distribution is similar to that of a beta distribution. For $\eta = 1$, the result is a uniform distribution. Despite being the identity over correlation matrices, the marginal distribution over the entries in that matrix (i.e., the correlations) is not uniform between -1 and 1. Rather, it concentrates around zero as the dimensionality increases due to the complex constraints. For $\eta > 1$, the density increasingly concentrates mass around the unit matrix, i.e., favoring less correlation. For $\eta < 1$, it increasingly concentrates mass in the other direction, i.e., favoring more correlation.

That is, for small correlation matrices (say, $k = 2$), the distribution is more spread out, and the modes are more pronounced when $\eta < 1$. Visually, it seems like the matrix size k is less important for larger values of η. All of this is to say, we probably want to choose an $\eta > 1$ (which is also recommended by the *Guide*), but we will also want to choose η differently depending on the number of participant-level parameters k we have in

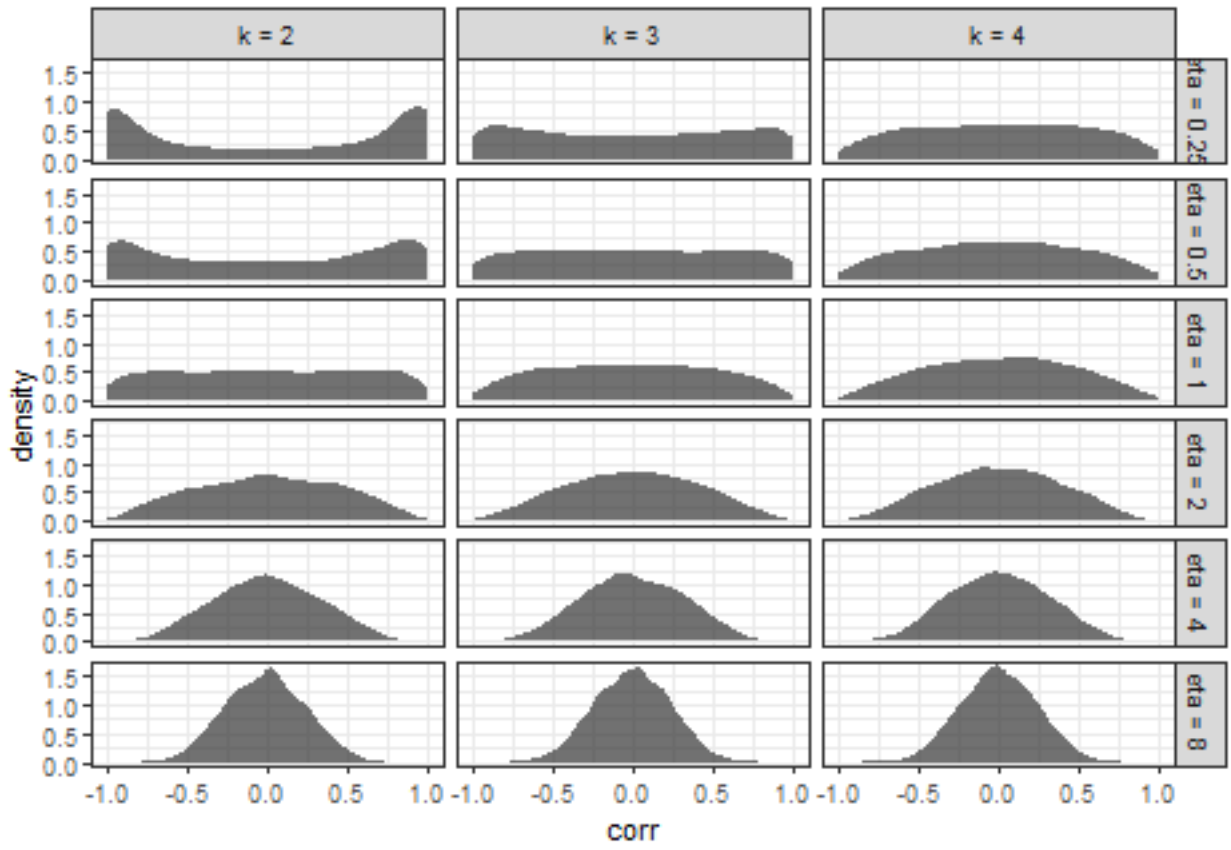


Figure 28: Simulated distributions of correlations from the LKJ distribution. k is the matrix size, and η is the parameter in the LKJ distribution.

our model.

Section 1.13 of the *Stan User's Guide*[24] also suggests using Cholesky factorization to further decompose the correlation matrix Ω. That is, we can represent Ω as the outer product of a triangular matrix L_Ω with itself:

$$\Omega = L_\Omega L_\Omega^\top$$

This is particularly useful because we can then express our participant-level parameters as a linear combination of standard normal random variables:

$$\begin{aligned}\theta_i &= \mu + \text{diag}(\tau) L_\Omega z_i \\ z_{i,j} &\sim iidN(0,1)\end{aligned}$$

This representation speeds up computation because evaluating the multivariate normal density and the LKJ prior both require factorization. By re-parameterizing the model in terms of the already factorized values, we avoid this factorization. This also permits writing the whole collection of individual parameters $\theta = (\theta_1, \theta_2, \ldots, \theta_n)$ as one big matrix operation involving the already factorized parameters, and standard normals (which are very fast to simulate):

$$\begin{aligned}\theta &= \mu 1_n^\top + \text{diag}(\tau) L_\Omega z \\ z_{j,i} &\sim iidN(0,1)\end{aligned}$$

where z is a $k \times n$ matrix of standard normals.

All of these modifications might make your model less intuitive to read, but they will make it easier for *Stan* to simulate your posterior quickly and without errors. What we will end up with is a model whose actual parameters are $\{\mu, \tau, L_\Omega, z\}$ instead of $\{\mu, \Sigma, \theta\}$, however we will always be able to generate the (hopefully) more interpretable parameters of Σ, Ω, and θ, because there is a deterministic relationship between them.

6.4.2 Transformed parameters and normal distributions

One restriction that applies when estimating a hierarchical model with a multivariate normal distribution of individual-level parameters is that these individual-level parameters must be able to take on any real number. This is not a problem for, say, the risk-aversion parameter r in the CRRA utility function $u_i(x) = \frac{x^{1-r_i}}{1-r_i}$, however there are many other cases where our models' parameters must be resrticted to only a subset of the real number line. For these parameters, we are going to have to transform them before they enter the model. That is, instead of estimating (say) logit choice precision parameter λ, which must be positive, we could estimate $\log \lambda$, as $\log(\cdot)$ takes a positive real number, and maps it onto the real number line. Other times, our parameters need to be restricted to the unit interval. In these cases one can use the inverse logit $(\log(x) - \log(1-x))$ or inverse normal cumulative density $\Phi^{-1}(x)$.

6.5 Example: again with Bruhin, Fehr, and Schunk (2019)

```
library(tidyverse)

d1<-(read.csv("Data/BFS2019_choices_exp1.csv")
     |> mutate(experiment=1)
)
d2<-(read.csv("Data/BFS2019_choices_exp2.csv")
     |> mutate(experiment=2)
)
D<-(rbind(d1,d2)
```

[24] This is a really useful section. I have it open in my browser most of the time that I am coding hierarchical models in *Stan*.

```
    # Just use the dictator game data
    |> filter(dg==1)
    |> mutate(
      self_alloc = ifelse(choice_x==1,self_x,self_y),
      other_alloc = ifelse(choice_x==1,other_x,other_y)
    )
)
```

I will begin with again analyzing the modified dictator game decisions in Bruhin, Fehr, and Schunk (2019), which I introduced in the previous chapter. Recall that in this experiment, 174 participants each made 78 pairwise choices over allocations of money between themself and another participant. As with the previous chapter, I will stick with the Fehr and Schmidt (1999) model of inequality-aversion as the deterministic economic model:

$$U(\pi_1, \pi_2; \alpha_i, \beta_i) = \pi_1 - \alpha_i \max\{\pi_2 - \pi_1, 0\} - \beta_i \max\{\pi_1 - \pi_2\}$$

and the logistic choice model to make it a probabilistic model that permits a likelihood representation:

$$X_{i,t} \mid \theta, \mu, \Sigma \sim \text{Bernoulli}\left(\Lambda\left(\lambda_i\left(U(\pi_1^X, \pi_2^X; \alpha_i, \beta_i) - U(\pi_1^Y, \pi_2^Y; \alpha_i, \beta_i)\right)\right)\right)$$

While this statement of the model is almost identical to how I made it in the previous chapter, there are some changes in notation that need to be noted. Firstly, I have explicitly put an i subscript on the individual-level parameters α_i, β_i, and λ_i. This is to make it clear that each participant i has their own parameters. For ease of notation here, I have lumped all of these parameters together into θ, so that we can talk about the population distribution more compactly. That is:

$$\theta_i = \begin{pmatrix} \alpha_i & \beta_i & \log \lambda_i \end{pmatrix}^\top$$

where taking the log of λ ensures that all elements of the (transformed) parameter vector θ can take on any real number. The above sampling statement also conditions on the population level parameters μ and Σ, however we do not see any elements of these parameters on the right-hand side of the sampling statement. This is because it is assumed that θ, and in fact just θ_i, contains all of the information about the distribution of choices $X_{i,t}$ for participant i. Put differently, once we know a participant's parameters θ_i, the population distribution does not tell us anything more about the distribution of their choices.

To fully specify the likelihood, we need to specify how the individual-level parameters θ_i are generated by the population-level parameters μ and Σ. Let's specify an independent multivariate normal distribution for θ_i:

$$\theta_i \sim iidN(\mu, \Sigma)$$

where μ is the population mean of θ_i and Σ is the population variance-covariance matrix of θ_i.

At this point, note that if I told you the population level parameters μ and Σ, then you would be able to simulate data from the experiment. Specifically, you would draw n θ_is from the multivariate normal, and then use these θ_is to simulate the binary choices $\{X_{i,t}\}_{i=1,t=1}^{n,T}$. This means we have all of the information needed to specify the likelihood. What remains is to choose priors for the population-level parameters μ and Σ. As building up priors on variance-covariance matrices can be daunting if handled in one go, I will first take you through a model that assumes that the individual-level parameters are uncorrelated. Fortunately this for us, this is a really useful stepping stone for the correlated model, because *Stan* lets us decompose a variance-covariance matrix into a vector of standard deviations and a correlation matrix.

6.5.1 No correlation between individual-level parameters

Suppose to begin with that our individual-level parameters are not correlated with each other. We can therefore re-write the hierarchical structure of the model as:

$$\theta_{i,\alpha} \sim iidN(\mu_\alpha, \tau_\alpha^2)$$
$$\theta_{i,\beta} \sim iidN(\mu_\beta, \tau_\beta^2)$$
$$\theta_{i,\lambda} \sim iidN(\mu_\lambda, \tau_\lambda^2)$$

6.5.1.1 A quick prior calibration Here it is appropriate to assign normal priors to the mean parameters, and half Cauchy priors (i.e. truncated to be positive only) to the standard deviations. As with the example using this dataset in the previous chapter, let's start by selecting a prior for λ_i. However this time, we need to specify priors for the parameters that govern the distribution of λ_i. Here, I decided to use the following prior for λ in the representative agent model:

$$\log \lambda \sim N(-5.76, 2.11^2)$$

It seems reasonable that we will want the prior mean of μ_λ to be in the same location, so this leaves us with working out an appropriate prior standard deviation for μ_λ. Recall that in this previous example, I used the model's predicted probability for a selfish participant as a gauge for where the economically interesting values of λ were. Again, we can then limit ourselves to looking at twelve different payoff differences:

```
d<-(D
    |> mutate(absDiff = abs(self_x-self_y))
    |> dplyr::select(absDiff)
    |> unique()
    )
d$absDiff |> print()
```

```
##  [1] 140 200 380 240 120  80  40 280 180   0 160 360
```

As this problem will be a bit more complicated, I will just focus on the median value of these differences:

```
(dMedian<-d$absDiff |> median())
```

```
## [1] 170
```

Furthermore, since we are taking the log of λ, $\exp(\mu_\lambda)$ gives us the *median* of the distribution of the predicted choice probability. Figure 29 shows how the distribution of this choice probability, summarized by the median, 2.5th, 25th, 75th, and 97.5th percentiles, changes as we change the prior standard deviation of μ_λ. Here, we don't want to set the prior standard deviation too small, because otherwise it would influence our posterior too much. On the the other hand, we don't want to set it too large, either, because this could mean we are searching for solutions of our model in unlikely regions of the parameter space. For me, choosing a prior standard deviation of one find this happy middle ground. That is, I will go with:

$$\mu_\lambda \sim N(-5.76, 1^2)$$

```
lambdaCheck<-(
  tibble(priorSD = seq(0,4,length=100))
      |> mutate(
        PrMedian = 1/(1+exp(-exp(-5.76)*dMedian)),
        Pr975    = 1/(1+exp(-exp(-5.76+priorSD*1.96)*dMedian)),
        Pr025    = 1/(1+exp(-exp(-5.76-priorSD*1.96)*dMedian)),
        Pr750    = 1/(1+exp(-exp(-5.76+priorSD*0.67)*dMedian)),
        Pr250    = 1/(1+exp(-exp(-5.76-priorSD*0.67)*dMedian)),
```

```
        )
)

(
  ggplot(lambdaCheck,aes(x=priorSD,y=PrMedian))
  +geom_path()
  +geom_ribbon(aes(ymax=Pr975,ymin=Pr025),alpha=0.2)
  +geom_ribbon(aes(ymax=Pr750,ymin=Pr250),alpha=0.4)
  +theme_bw()
  +ylab("median probability of choosing the option with the greatest utility")
  +xlab("prior standard deviation of the population mean parameter for \u03bb")
)
```

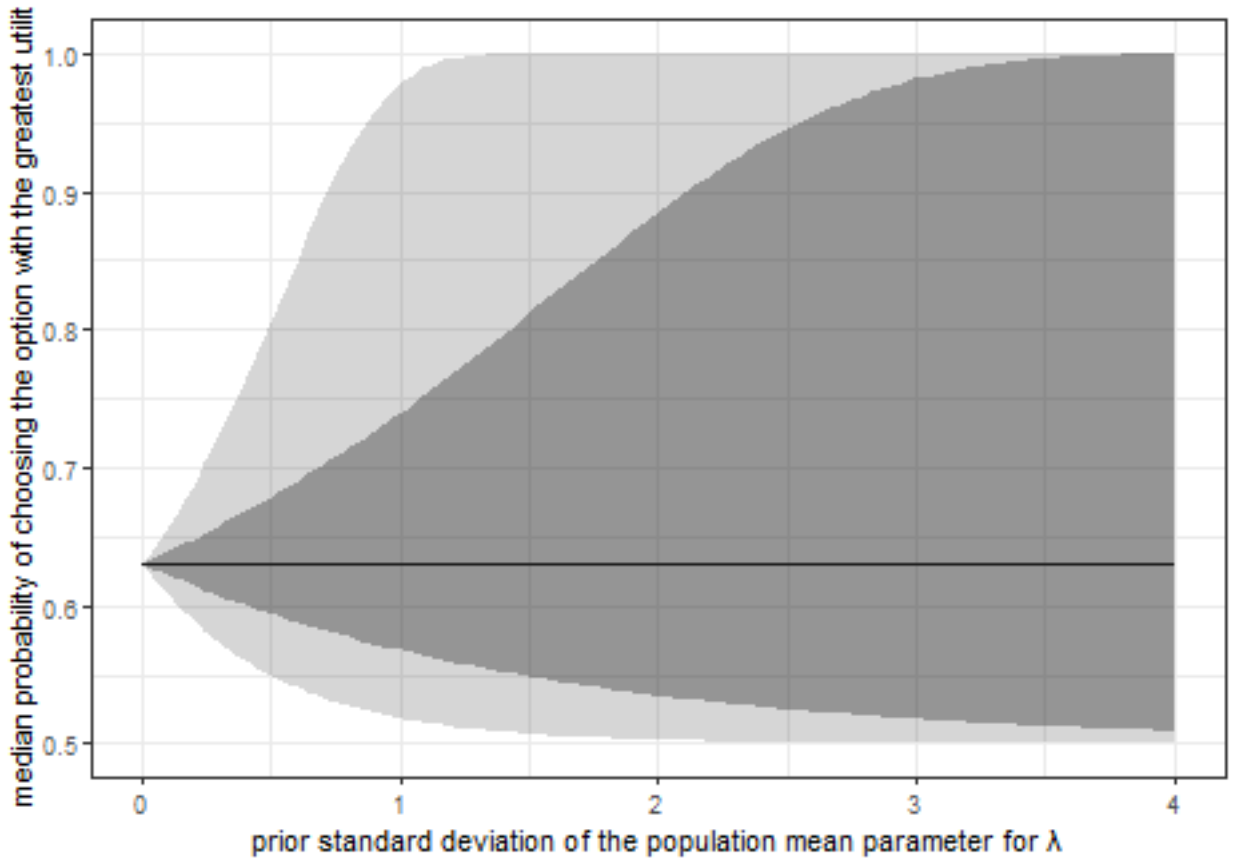


Figure 29: A plot of the distribution of predicted choice probabilities that will help us calibrate the priors for choice precision.

Figure 29 will also be helpful to us in choosing a prior for τ_λ, the standard deviation of $\log \lambda_i$. Firstly, since I have limited myself to priors of the form:

$$\tau_\lambda \sim \text{Cauchy}^+(0, g)$$

it is informative to know that the cumulative distribution function of τ_λ is:

$$F(x) = \frac{2}{\pi} \arctan\left(\frac{x}{g}\right) I(x \geq 0)$$

and so we can write the inverse cumulative density function as:

$$F^{-1}(p) = g \tan\left(\frac{\pi p}{2}\right)$$

so the 95th percentile of the distribution is $F^{-1}(0.95) = g \tan(0.95\pi/2) \approx 12.7g$. Figure 29 shows us, if we fix μ_λ to its prior median of -5.76, what the implications of different values of τ_λ are. That is, we can re-label the horizontal axis as "τ_λ", and the vertical axis as "distribution of choice probabilities if $\mu_\lambda = -5.76$". Since for $\tau_\lambda > 3$ the interquartile range of the distribution of this choice probability (dark shaded region) takes up almost all of the vertical distance, these seem like unlikely values. Therefore I will calibrate the prior so $\tau_\lambda = 3$ is the 95th percentile of its distribution. Therefore $g = 3/12.7 \approx 0.24$, so:

$$\begin{aligned}\mu_\lambda &\sim N(-5.76, 1^2)\\ \tau_\lambda &\sim \text{Cauchy}^+(0, 0.24)\end{aligned}$$

For the priors for the parameters governing the distributions of α_i and β_i, I will keep them centered on zero. This means that I am looking to fill the blanks in the following distributions:

$$\mu_\alpha \sim N(0, s^2), \quad \tau_\alpha \sim \text{Cauchy}^+(0, g)$$

and similarly for μ_β, and τ_β. While in the representative agent model I calibrated the priors for α and β so that 75% of the prior distribution was in the range $(-1, 1)$, this would put too much mass outside this region for their population means: we should expect there to be less prior uncertainty in μ_α than in α. This is because μ_α is a measure of central tendency for α_i, and so it seems reasonable that prior to observing the data, we should be more certain about μ_α than an individual α_i. Therefore, I will calibrate the priors for μ_α and μ_β so that 99% of the probability mass is in this region, so $s = 2.58^{-1} \approx 0.39$. For g, a standard deviation of one seems large to me for both α and β, because these parameters being greater than one means that the participant will place more weight on the other person's earnings than their own. Choosing $g = 0.079$ means that having $\tau_\alpha > 1$ will happen only 5% of the time in the prior. Therefore, all together my priors for this model are:

$$\begin{aligned}\mu_\alpha &\sim N(0, 0.39^2), \quad \tau_\alpha \sim \text{Cauchy}^+(0, 0.79)\\ \mu_\beta &\sim N(0, 0.39^2), \quad \tau_\beta \sim \text{Cauchy}^+(0, 0.79)\\ \mu_\lambda &\sim N(-5.76, 1^2), \quad \tau_\lambda \sim \text{Cauchy}^+(0, 0.24)\end{aligned}$$

6.5.1.2 Implementation and estimation in *Stan* This is where all of the work we did in making the representative agent model comes in handy. Here is all we have to do:

1. Add in another data variable that keeps track of the participant id (i.e. the i index)
2. Change the prior data variables so that they are for priors over the population-level parameters
3. Modify the existing parameters in the parameters block so that they are participant-specific
4. Add in the new population-level parameters in the parameters block
5. Specify the hierarchical structure in the model block

This might seem like a lot, but it will be exactly the same every time we make a model hierarchical. Importantly, we barely have to modify any of the part of the code that does the individual-level likelihood parts. Here is what I came up with after unashamedly copying and pasting my representative agent model code into a new file:[25]

[25]Here I have deliberately tried to leave as much of this *Stan* file as unchanged as possible so you can compare it side-by-side with the representative agent model. There are better ways to do this, which I will get to shortly. Most of them take advantage of the stacked vector θ notation that I have used above.

```
data {
  int<lower=0> n; // number of observations
  vector[n] self_x; // payoff to self with allocation x
  vector[n] other_x; // payoff to other with allocation x
  vector[n] self_y; // payoff to self with allocation y
  vector[n] other_y; // payoff to other with allocation y
  int choice_x[n]; // =1 iff allocation x is chosen

  /* <----- ADDITION: we need a variable to keep track of the participant ids
  We also need an integer telling us how many participants we have
  */
  int id[n];
  int nParticipants;

  /* <---- DELETION: we now have priors over population-level parameters
  real prior_lambda[2];
  real prior_alpha[2];
  real prior_beta[2];
  */

  // <---- ADDITION: data specifying the priors;
  real prior_mu_alpha[2];
  real prior_mu_beta[2];
  real prior_mu_lambda[2];
  real prior_tau_alpha;
  real prior_tau_beta;
  real prior_tau_lambda;

}

transformed data {
  vector[n] dX; // disadvantageous inequality at allocation x
  vector[n] dY; // disadvantageous inequality at allocation y
  vector[n] aX; // advantageous inequality at allocation X
  vector[n] aY; // advantageous inequality at allocation X

  dX = fmax(0,other_x-self_x);
  dY = fmax(0,other_y-self_y);
  aX = fmax(0,self_x-other_x);
  aY = fmax(0,self_y-other_y);
}
parameters {

  /* <--- MODIFICATION: the individual-level parameters are now vectors
  */
  vector[nParticipants] alpha;
  vector[nParticipants] beta;
  vector<lower=0>[nParticipants] lambda;

  // <---- ADDITION: New population-level parameters
  real mu_alpha;
  real mu_beta;
  real mu_lambda;
```

```
  real<lower=0> tau_alpha;
  real<lower=0> tau_beta;
  real<lower=0> tau_lambda;


}
model {

  vector[n] Ux; // utility of allocation x
  vector[n] Uy; // utility of allocation y

  /* <---- MODIFICATION: Since the individual level parameters are now vectors,
  we need to adjust the utility equations so the right parameters are matched
  with the right observations. This is where the new id variable comes in
  */
  Ux = self_x-alpha[id] .* dX-beta[id] .* aX;
  Uy = self_y-alpha[id] .* dY-beta[id] .* aY;

  // <---- Slight modification to the likelihood to bring in the id variable
  choice_x ~ bernoulli_logit(lambda[id] .* (Ux-Uy));

  /* <---- MODIFICATION: priors on individual level parameters are now
  statements about their relationship with population level parameters
  */
  alpha~normal(mu_alpha,tau_alpha);
  beta ~normal(mu_beta,tau_beta);
  lambda ~ lognormal(mu_lambda,tau_lambda);

  // <---- ADDITION: priors for population-level parameters
  mu_alpha ~ normal(prior_mu_alpha[1],prior_mu_alpha[2]);
  mu_beta ~ normal(prior_mu_beta[1],prior_mu_beta[2]);
  mu_lambda ~ normal(prior_mu_lambda[1],prior_mu_lambda[2]);

  tau_alpha ~ cauchy(0,prior_tau_alpha);
  tau_beta ~ cauchy(0,prior_tau_beta);
  tau_lambda ~ cauchy(0,prior_tau_lambda);

}
```

From here, we can pass the data to *Stan* and estimate the model:

```
file<-"Hierarchical/Hierarchical_BFS2019"

# Generate the id variable
D <-(
  D
  |> mutate(id = sid %>% paste() %>% as.factor() %>% as.numeric())
)
# only run if I haven't estimated it yet
if (!file.exists(paste0("Outputs/",file,"_independent.rds"))) {

  dStan<-list(
    n=dim(D)[1],
    self_x=D$self_x,
```

```
    other_x=D$other_x,
    self_y=D$self_y,
    other_y=D$other_y,
    choice_x=D$choice_x,

    id = D$id,
    nParticipants = D$id |> unique() |> length(),

    prior_mu_alpha   = c(0,0.39),
    prior_mu_beta    = c(0,0.39),
    prior_mu_lambda  = c(-5.76,1),
    prior_tau_alpha  = 0.79,
    prior_tau_beta   = 0.79,
    prior_tau_lambda = 0.24

  )
  Fit<-stan(paste0("Code/",file,"_independent.stan"),
            data=dStan,
            seed=42)
  saveRDS(Fit,paste0("Outputs/",file,"_independent.rds"))
}

Fit<-readRDS(paste0("Outputs/",file,"_independent.rds"))

# rstan::summary(Fit)$summary |> View()
```

Interestingly (to me at least), this one did not take as long as I expected (under ten minutes on my laptop). This is compared to about half an hour for the individual-specific estimations in the previous chapter. Think about this: even though we estimated $3n$ parameters in the individual-specific estimations, and $3n + 6$ parameters just now, the second estimation was faster! While I don't have a well-thought-out explanation for this, my intuition is that the population-level parameters are forming a much more informative "prior" over the individual-level parameters, so sampling the individual-level parameters is less burdensome for *Stan*.

Anyway, we may as well look at the estimates. As we have so many individual-level parameters, I will just show a summary in Table 7 for the population-level parameters:

```
s<-rstan::summary(Fit)$summary

# identify the rows that contain the population-level parameters.
# These have row names that contain the character "_"
II<- grepl("_",rownames(s))
s[II,] |> kbl(digits=3,caption="Population estimates of a model assuming a hierarchical independent dis
  kable_classic(full_width=FALSE)
```

6.5.2 Correlation between individual-level parameters

Now let's modify the model so that we can estimate a correlation matrix for our participant-level parameters. Conceptually, this is a fairly straightforward step, as we have already assigned priors to μ and τ: by decomposing Σ into τ and Ω, really the only bit of thinking we need to do is to assign an appropriate prior for Ω. For this, I choose:

$$\Omega \sim \text{LKJCorr}(2)$$

which seems like a reasonably spread-out but still unimodal distribution, as shown in Figure 28 for $k = 3$ participant-level parameters. Here is what I came up with for the code for the hierarchical model with

Table 7: Population estimates of a model assuming a hierarchical independent distribution of individual-level parameters.

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff	R
mu_alpha	-0.030	0.000	0.014	-0.058	-0.040	-0.030	-0.020	-0.001	3933.694	0.
mu_beta	0.233	0.000	0.020	0.195	0.219	0.233	0.246	0.271	6327.612	0.
mu_lambda	-3.498	0.001	0.072	-3.636	-3.546	-3.499	-3.450	-3.353	3126.904	1.
tau_alpha	0.165	0.000	0.015	0.138	0.155	0.165	0.175	0.196	2065.354	1.
tau_beta	0.248	0.000	0.016	0.219	0.237	0.247	0.259	0.283	4071.073	1.
tau_lambda	0.829	0.001	0.066	0.708	0.784	0.827	0.871	0.971	2094.545	1.
lp___	-2345.289	0.439	16.981	-2379.805	-2356.544	-2345.241	-2333.389	-2312.640	1493.791	1.

correlated parameters.

```
data {
  int<lower=0> n; // number of observations
  vector[n] self_x; // payoff to self with allocation x
  vector[n] other_x; // payoff to other with allocation x
  vector[n] self_y; // payoff to self with allocation y
  vector[n] other_y; // payoff to other with allocation y
  int choice_x[n]; // =1 iff allocation x is chosen

  int id[n];
  int nParticipants;



  vector[2] prior_mu[3];
  real prior_tau[3];
  real prior_LKJ;

  // number of simulation steps to approximate demand
  int nsim;
  // Endowment of tokens for demand
  int W;
  // number of prices to evaluate demand at
  int nrho;
  // prices to evaluate demand at
  vector[nrho] rho;


}

transformed data {
  vector[n] dX; // disadvantageous inequality at allocation x
  vector[n] dY; // disadvantageous inequality at allocation y
  vector[n] aX; // advantageous inequality at allocation X
  vector[n] aY; // advantageous inequality at allocation X

  dX = fmax(0,other_x-self_x);
  dY = fmax(0,other_y-self_y);
  aX = fmax(0,self_x-other_x);
  aY = fmax(0,self_y-other_y);
```

```
  // some things used to produce demand
  vector[W+1] X;
    for (xx in 1:(W+1)) {
      X[xx] = xx-1;
    }



}
parameters {

  vector[3] mu; // population mean
  vector<lower=0>[3] tau; // population standard deviation
  cholesky_factor_corr[3] L_Omega; // Cholesky factorization of the correlation matrix

  matrix[3,nParticipants] z; // standard normals determinig participants' parameters

}
model {

  vector[n] Ux; // utility of allocation x
  vector[n] Uy; // utility of allocation y
  vector[n] alpha;
  vector[n] beta;
  vector[n] lambda;

  matrix[3,nParticipants] theta;

  theta = mu*rep_row_vector(1.0,nParticipants)
                  +diag_pre_multiply(tau,L_Omega)*z;

  alpha  = theta[1,id]';
  beta   = theta[2,id]';
  lambda = exp(theta[3,id]');

  Ux = self_x-alpha .* dX-beta .* aX;
  Uy = self_y-alpha .* dY-beta .* aY;

  choice_x ~ bernoulli_logit(lambda .* (Ux-Uy));

  to_vector(z) ~ std_normal();

  for (pp in 1:3) {
    mu[pp] ~ normal(prior_mu[pp][1],prior_mu[pp][2]);
    tau[pp] ~ cauchy(0.0,prior_tau[pp]);
  }

   L_Omega ~ lkj_corr_cholesky(prior_LKJ);
}

generated quantities {
  matrix[3,3] Omega;
  matrix[3,nParticipants] theta;
```

```
theta = mu*rep_row_vector(1.0,nParticipants)
                +diag_pre_multiply(tau,L_Omega)*z;
Omega = L_Omega*L_Omega';

// store demand in this vector
vector[nrho] demand;

{
  // simulated standard normals
  matrix[3,nsim] zsim;
  for (rr in 1:3) {
    zsim[rr,] = to_row_vector(normal_rng(rep_vector(0.0,nsim),rep_vector(1.0,nsim)));
  }

  // simulate parameters
  matrix[3,nsim] thetasim;
  thetasim = mu*rep_row_vector(1.0,nsim)
                +diag_pre_multiply(tau,L_Omega)*zsim;

  vector[nsim] alphasim = thetasim[1,]';
  vector[nsim] betasim = thetasim[2,]';
  vector[nsim] lambdasim = exp(thetasim[3,]');

  // go through each value of rho
  for (rr in 1:nrho) {

    // dump individual-level expected demand in here
    vector[nsim] Y;
    for (ss in 1:nsim) {

      // utility of choosing each allocation
      vector[W+1] U = X
          -alphasim[ss]*fmax(0.0,(W-X*(1.0+rho[rr]))/rho[rr])
          -betasim[ss]*fmax(0.0,(X*(1+rho[rr])-W)/rho[rr]);
      // probability of choosing each allocation
      vector[W+1] pr = softmax(lambdasim[ss]*U);
      // expected tokens kept
      real EX = sum(X.*pr);
      // expected tokens bought for the other participant
      Y[ss] = (W-EX)/rho[rr];

    }

    demand[rr] = mean(Y);

  }



}
```

```
}
```

Here I am implementing both the Cholesky decomposition of Ω and the vectorization as described in Section 1.13 of the *Stan User's Guide*. In particular, note that θ is *not* declared in the `parameters` block. Rather, I use the matrix `z` of standard normals to generate θ in the `model` block:

```
theta = mu*rep_row_vector(1.0,nParticipants)
                    +diag_pre_multiply(tau,L_Omega)*z;
```

Also, since Ω is not a fundamental parameter, I generate it in the `generated quantities` block so that we can look at it later.

Here's how you estimate the model in *RStan*:[26]

```
library(rstan)
rstan_options(auto_write = TRUE)
options(mc.cores = parallel::detectCores())

if (!file.exists(paste0("Outputs/",file,"_correlated.rds"))) {

  dStan<-list(
    n=dim(D)[1],
    self_x=D$self_x,
    other_x=D$other_x,
    self_y=D$self_y,
    other_y=D$other_y,
    choice_x=D$choice_x,

    id = D$id,
    nParticipants = D$id |> unique() |> length(),

    prior_mu = rbind(c(0,0.39),
                     c(0,0.39),
                     c(-5.76,1)
                     ),
    prior_tau = c(0.79,0.79,0.24),
    prior_LKJ = 2,

    # data needed for demand calculation
    # (see the "Doing something with the estimates" section)
    nsim=100,
    W = 1000,
    nrho = 100,
    rho = seq(0.1,10,by=0.1)


  )
  Fit<-stan(paste0("Code/",file,"_correlated.stan"),
```

[26] This one will take a while. Fold some laundry, write some poetry, find Kakariko Village, or something. (about 17 minutes on my laptop)

Table 8: Posterior estimates from the model with correlated parameters.

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff	Rhat
mu[1]	-0.014	0.001	0.015	-0.044	-0.024	-0.014	-0.004	0.017	898.269	1.008
mu[2]	0.230	0.001	0.020	0.191	0.216	0.230	0.243	0.269	1156.227	1.004
mu[3]	-3.531	0.002	0.068	-3.662	-3.576	-3.532	-3.486	-3.394	1593.340	1.004
tau[1]	0.175	0.000	0.014	0.149	0.165	0.174	0.184	0.204	1584.464	1.001
tau[2]	0.242	0.000	0.016	0.211	0.231	0.241	0.252	0.275	1692.600	1.000
tau[3]	0.774	0.002	0.062	0.660	0.731	0.771	0.814	0.902	1292.542	1.001

Table 9: Estimated correlation matrix Ω. Posterior means shown with posterior standard deviations in parentheses.

	α	β	$\log\lambda$
α	1 (0)	-0.689 (0.061)	-0.594 (0.064)
β	-0.689 (0.061)	1 (0)	-0.005 (0.088)
$\log\lambda$	-0.594 (0.064)	-0.005 (0.088)	1 (0)

```
            data=dStan,
            seed=42)
  saveRDS(Fit,paste0("Outputs/",file,"_correlated.rds"))
}

Fit<-readRDS(paste0("Outputs/",file,"_correlated.rds"))
```

There are quite a lot of parameters in this model (because we are using data augmentation), so looking at the entire summary is somewhat daunting and not particularly useful. Fortunately, I set up the parameters block so that the first six parameters listed have the same interpretation as the model in the previous section that assumes no correlation between parameters (except that we are estimating λ as $\log\lambda$):

```
summary(Fit)$summary[1:6,] |> kbl(caption = "Posterior estimates from the model with correlated paramete
```

To read Table 8, recall that the order of parameters in the vectors μ and τ is $(\alpha_i\ ,\beta_i\ ,\ \log\lambda_i)^\top$. These population-level parameters are all quite similar to those estimated in the previous section, where we assumed no correlation between the participant-level parameters. However now we also get estimates of the correlations between these parameters, which can be see in tin Table 9. In particular, note the strong negative correlation between α and β: this is something we would not have beed able to comment on had we just estimated the model without correlation.

```
Omega<-extract(Fit)$Omega

OmegaMean<-Omega |> apply(c(2,3),mean)
OmegaSD  <-Omega |> apply(c(2,3),sd)

table<-paste0(OmegaMean |> round(3),
              "  (",OmegaSD |> round(3),")") |> matrix(nrow=3)

colnames(table)<-c("$\\alpha$","$\\beta$","$\\log\\lambda$")
rownames(table)<-c("$\\alpha$","$\\beta$","$\\log\\lambda$")

table|> kbl(digits=3,caption = "Estimated correlation matrix $\\Omega$. Posterior means shown with poste
  kable_classic(full_width=FALSE)
```

Finally, let's have a look at the model's shrinkage estimates. Recall that these come from the data-augmentation process, and can be thought of as participant-specific estimates, had we used the parameters μ and Σ that we estimated as our prior for these parameters. As with the previous chapter, I will focus on parameters α and β, which are the parameters of the Fehr and Schmidt (1999) inequality-aversion model. Also, as with the previous chapter, I will just plot their posterior means to avoid over-plotting. I show these in Figure 30

```
# extract the right parameters from the Stan Fit object
# and take the mean
theta<-extract(Fit)$theta |> apply(c(2,3),mean)|> t()
dTheta<-tibble(alpha=theta[,1],beta=theta[,2],lambda = exp(theta[,3]))

regions<-tibble(
  alpha = c(0.25,-0.25,-0.25,0.25),
  beta = c(0.5,-0.5,0.8,-0.5),
  label = c("inequality averse",
            "inequality loving",
            "efficiency loving",
            "competitive"
            )
)

(
  ggplot(
    data = dTheta,
    aes(x=alpha,y=beta,color=log(lambda))
  )
  +geom_point()
  +geom_text(data=regions,aes(x=alpha,y=beta,label=label),color="black")
  +theme_bw()
  +xlab(expression(alpha))+ylab(expression(beta))
  +geom_hline(yintercept=0,linetype="dashed")
  +geom_vline(xintercept=0,linetype="dashed")
)
```

6.5.2.1 Doing something with the estimates

Now that we have a model and estimates that respect the heterogeneity we believe is present in the population, how about we use it to make an out-of-sample prediction? For this exercise, we will use the estimated model to predict a demand curve for charitable giving, *a la* Andreoni and Vesterlund (2001) and Andreoni and Miller (2002). To do this, suppose that a participant has 1000 tokens,[27] which she can either keep, or use to buy income for another participant, who starts out with nothing. Kept tokens generate income of 1 experimental currency unit for the participant, and the participant can buy 1 experimental currency unit for the other participant for a price of ρ. Therefore, if x is the income of the participant, and y is the income of the other, then the budget constraint is:

$$1000 = x + \rho y$$

The participant's utility function is then:

$$U_i(x, y) = x - \alpha_i \max\{0, y - x\} - \beta \max\{0, x - y\}$$

And substituting the budget constraint into the utility function yields utility just as a function of x:

[27] I use 1000 tokens to keep the payoff scales similar to the original experiment.

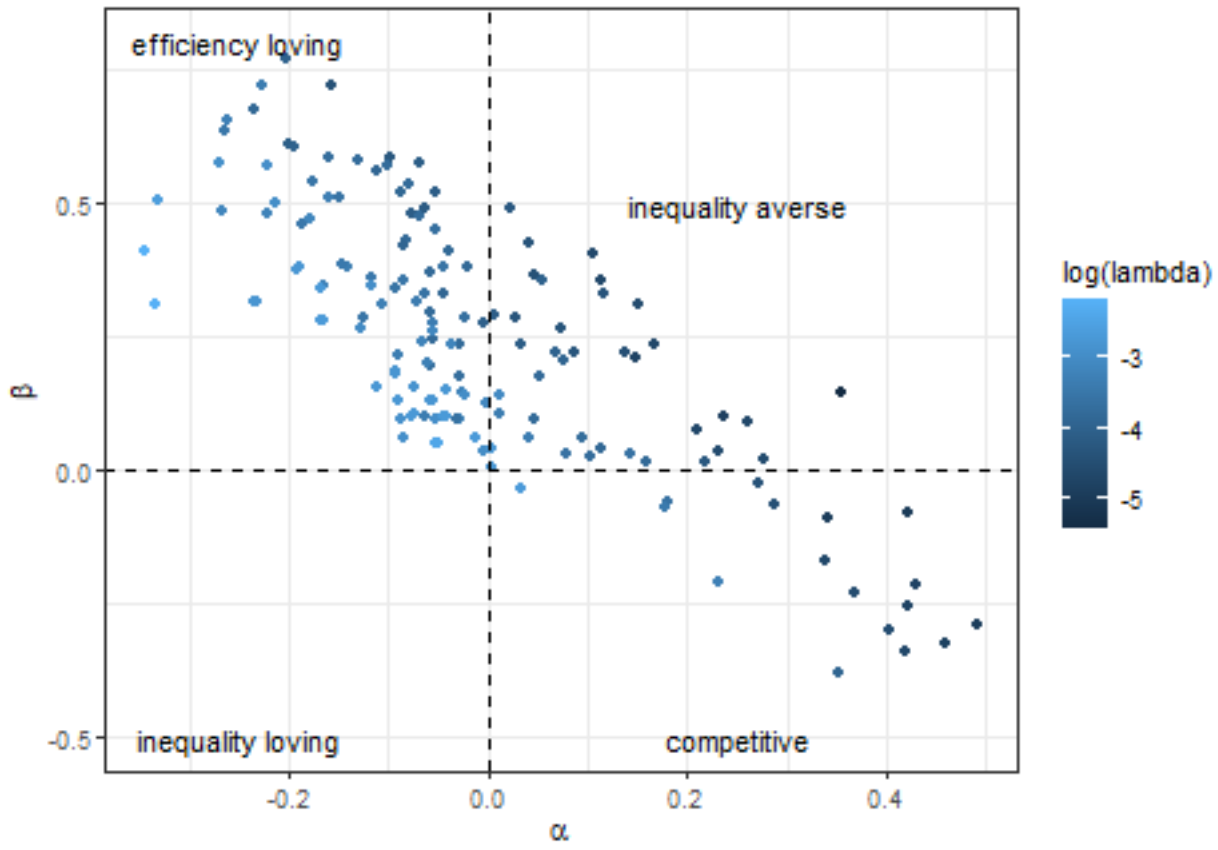


Figure 30: Posterior mean estimates of individual-level parameters α and β from the hierarchical model assuming correlation between the parameters.

$$U_i(x) = x - \alpha_i \max\left\{0, \frac{1000 - x}{\rho} - x\right\} - \beta_i \max\left\{0, x - \frac{1000 - x}{\rho}\right\}$$
$$U_i(x) = x - \alpha_i \max\left\{0, \frac{1000 - x(1+\rho)}{\rho}\right\} - \beta_i \max\left\{0, \frac{x(1+\rho) - 1000}{\rho}\right\}$$

Now assume that participants use logit choice with the same λ_i as in the original experiment.[28] So the distribution of choices given $(\alpha_i, \beta_i, \lambda_i)$ is:

$$\Pr(X = x \mid \alpha_i, \beta_i, \lambda_i, \rho) = \frac{\exp(\lambda_i U_i(x))}{\sum_{k=0}^{1000} \exp(\lambda_i U_i(k))}$$

So:

$$E[X \mid \alpha_i, \beta_i, \lambda_i, \rho] = \frac{\sum_{x=0}^{1000} x \exp(\lambda_i U_i(x))}{\sum_{k=0}^{1000} \exp(\lambda_i U_i(k))}$$

But what we really want to report is the *demand* for giving to the other participant, so we need to convert this into income bought for the other participant (y):

$$E[Y \mid \alpha_i, \beta_i, \lambda_i, \rho] = \frac{1000 - E[X \mid \alpha_i, \beta_i, \lambda_i, \rho]}{\rho}$$

This is participant i's (average, since they use probabilistic choice) demand for giving. But what we really want is a sense of what the *population* would demand. Hence, we need to integrate out the individual-specific parameters α_i, β_i, and λ_i. Mathematically, we can apply the law of iterated expectations like this, where the left-hand side is what we want to report as a function of ρ:

$$E[Y \mid \rho] = E\left[E[Y \mid \alpha_i, \beta_i, \lambda_i, \rho] \mid \rho\right]$$

In practice, this is a really hard expectation to take analytically, so instead we will do it with Monte Carlo integration:

$$E[Y \mid \rho] \approx \frac{1}{S} \sum_{s=1}^{S} E[Y \mid \alpha_s, \beta_s, \lambda_s, \rho]$$
$$\begin{pmatrix} \alpha_s \\ \beta_s \\ \log \lambda_s \end{pmatrix} \sim iidN(\mu, \Sigma)$$

You can see the implementation of this in the *Stan* file above. In particular, see the `generated quantities` block. In general, it is a good idea to use the `generated quantities` block to compute transformations of the parameters like this, as opposed to doing this outside of *Stan* in post-estimation. This is for at least two reasons. Firstly, since *Stan* is fast, this is probably the fastest way for you to compute things. Secondly, *Stan* gives you all of the convergence diagnostics for these transformed quantities, so you don't need to worry about (say) having too small an effective sample size or a bad $\hat{R}$ for these quantities.[29]

Now we can have a look at our estimated demand curve in Figure 31. For perspective, I also include the demand curve if participants chose allocations that equalized payoffs. This is shown with the red dashed line.

[28] This is a somewhat corageous assumption, as in the original experiment participants chose between two things at a time, and now we are asking them to choose between 1,001 things.

[29] I thank Brian Monroe for this insight.

Here we can see that participants are generally less generous than this, but get *more* generous than this for very low prices.

```
demand<-summary(Fit)$summary |>
  data.frame() |>
  rownames_to_column(var = "parameter") |>
  filter(grepl("demand",parameter)) |>
  mutate(
    price = seq(0.1,10,by=0.1),
    equal_split = 1000/(1+price)
  )

(
  ggplot(demand,aes(x=mean,y=price))
  +geom_line()
  +geom_line(aes(x=equal_split,y=price),color="red",linetype="dashed")
  +geom_ribbon(aes(xmin = X2.5.,xmax=X97.5.,y=price),alpha=0.5)
  +coord_cartesian(xlim = c(0,1500))
  +theme_bw()
  +ylab(expression("Price ("*rho*")"))
  +xlab("Quantity of experimental currency units demanded for the other participant")
)
```

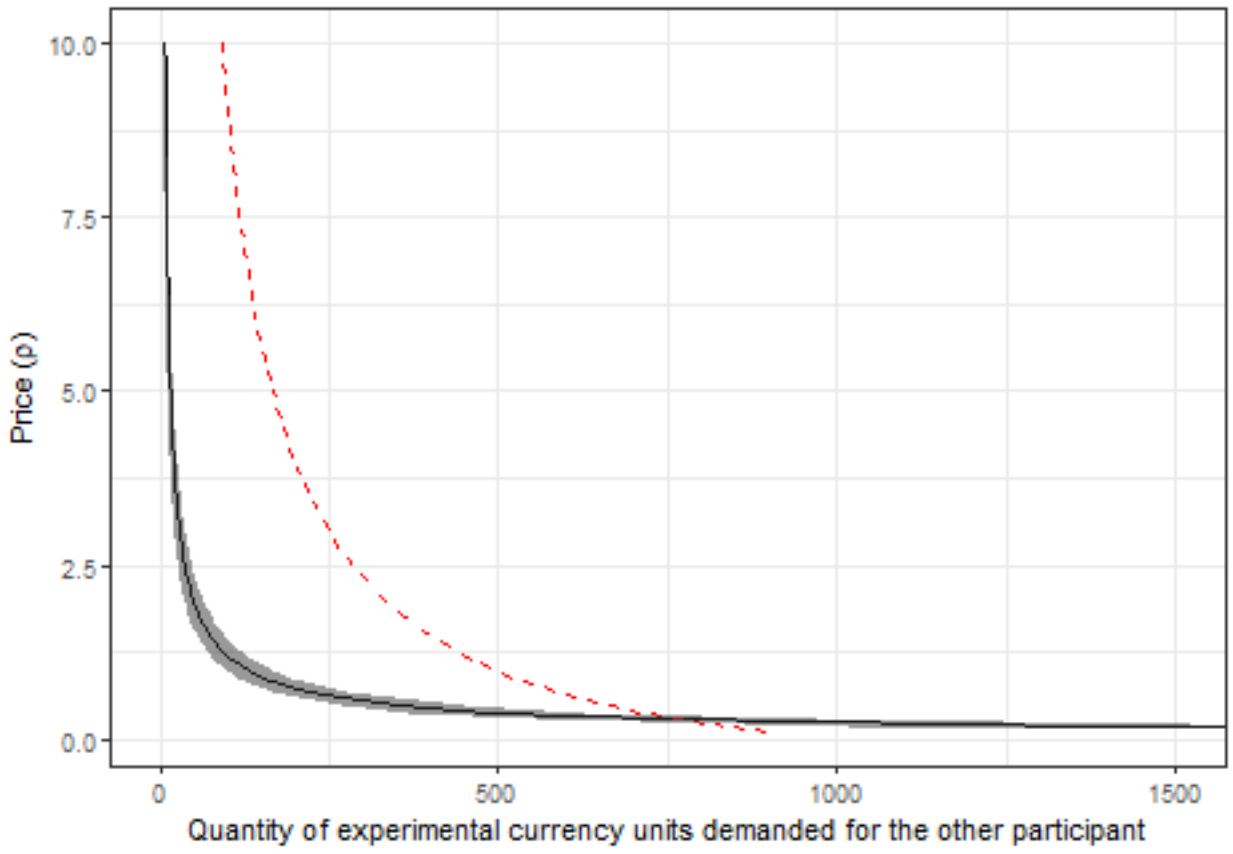


Figure 31: Estimted demand curve for giving. Line shows posterior mean estimate, shaded region shows a 95% Bayesian credible region (2.5th-97.5th percentile). Note that since this is a demand curve, the uncertainty is in the *horizontal* coordinates. Dashed red line shows the demand curve if participants equalized payoffs

Note that here we are making an *out-of-sample* prediction. That is, we are using our parameter estimates to predict how different participants (but drawn from the same population) would behave in a different environment (i.e. the budget set problem, rather than the binary choice task). This is a testable prediction if we run another experiment!

7 Mixture models

Sometimes we are lucky enough to have more than one model of how people could behave in our experiment. While this opens up a whole lot of interesting research questions that we could ask, it also creates a whole lot of problems associated with assigning relative importance to each of these models. One early solution was to take a "horse race" approach: define some measure of goodness of fit, such as the log-likelihood or mean squared prediction error, then pick the model (i.e. the horse) that wins by this measure. Applications of this technique can be seen in the classification of participants in Andreoni and Vesterlund (2001) and Andreoni and Miller (2002) into several types of other-regarding preferences using nonlinear least squares. They can also be seen in the classification of participants into different types of risk preferences in Hey and Orme (1994) using the Akaike information criterion. When applied to data one participant at a time, as is done in these applications, this kind of classification can begin to shed light on the underlying *fractions* of the population that behave according to each model.

However when applied to a whole experiment with many participants, picking a winner makes less sense: it could be that *all*, or at least a fair number of, models are important for substantial fractions of the population, and classifying all of our data into just one will not acknowledge the richness of our data-generating process. The mixture model is a formal way of estimating these fractions in the population. Here, we will take a menu of models that could be important for certain slices of our data, and estimate the fraction of the data that are best characterized by each of these models.

7.1 A menu of models

So we have a menu of models that we would like to include in our estimation. As we are dealing with probabilistic models, each of these on their own will come with a likelihood function. Our job is to take these model-specific likelihood functions, and combine them into a "grand likelihood" for the mixture model. To fix ideas, suppose that for every observation indexed by participant i and decision t, the likelihood for model m is $p(y_{i,t} \mid \theta_i, m)$, where θ_i is a vector of participant-specific parameters and $y_{i,t}$ is the data. Now we introduce the mixing probabilities $\{\rho_m\}_{m=1}^M \in \Delta^M$, which are the fraction of the data that are represented by each model. If this were our only observation, then our distribution of $y_{i,t}$ can be written as (for a mixture of $M = 3$ models):

$$p(y_{i,t} \mid \theta_i, \rho) = \rho_1 p(y_{i,t} \mid \theta_i, m_1) + \rho_2 p(y_{i,t} \mid \theta_i, m_2) + \rho_3 p(y_{i,t} \mid \theta_i, m_3)$$

That is, we have integrated out the uncertainty (measured by ρ) associated with which model observation $y_{i,t}$ is coming from. This representation means that if we can estimate each of these (in this case) three models separately using our data, then in principle at least, we can combine them into a mixture model, and estimate the fraction of data that are generated by each model.

These fractions can be very useful quantities to know. They speak to the relative importance of each model on our menu. From my own use of them, I have been able to show for example that most participants narrowly bracket in games (Bland 2019a) (as opposed to broadly bracketing), and that the majority of participants have Bayesian beliefs (as opposed to using multiple priors model) (Bland and Rosokha 2019).

7.2 Dichotomous and toolbox mixture models

I think of the typical types of structural mixture models taken to data from economic experiments falling into two categories. These relate to how the data may be sliced to divide up decisions as coming from one model or another. That is, data *within* a slice must be generated from the same model, but data *between*

slices can be generated from different models. These two types of mixture model differ as to how the data can be sliced. In particular, they differ in whether data from one participant can be generated by only one, or more than one of the models on our menu.

Conte, Hey, and Moffatt (2011), for example, assumes that the data can be sliced up to the participant level, but no further. An immediate implication of this is that participants have "types": if they make one decision based on model B, then they could *never* make decisions based on models A or C. Hence, we can interpret the mixing probabilities from this kind of model as the fraction of *participants* in the population who behave according to each model. I will refer to this kind of model as a "dichotomous" mixture model. When we are using a dichotomous mixture model with a menu of length M, our grand log-likelihood will be:

$$\log p(y \mid \rho, \beta) = \sum_{i=1}^{n} \log \left(\sum_{m=1}^{M} \rho_m \exp \left(\sum_{t=1}^{T} \log p(y_{i,t} \mid \beta, m) \right) \right)$$

where β is a vector of parameters other than ρ, and $p(y_{i,t} \mid \beta, m)$ is the likelihood for observation $y_{i,t}$ given that it was generated by model m.[30]

Harrison and Rutström (2009), on the other hand, assumes that the data can be sliced at the *decision* level. That is, participants can make some decisions based on model A, some based on model B, and so on. Following Stahl (2018), I refer to these models as *toolbox* mixture models. When we are using a toolbox mixture model with a menu of length M, our grand log-likelihood will be:

$$\log p(y \mid \rho, \beta) = \sum_{i=1}^{n} \sum_{t=1}^{T} \log \left(\sum_{m=1}^{M} \rho_m \exp \left(\log p(y_{i,t} \mid \beta, m) \right) \right)$$

In particular, note how the ordering of the summations is different. However perhaps more importantly, note how *the interpretation of the mixing probabilities is different.* In the dichotomous model, we assume that a participant's data can only be generated from one of the models on the menu. Hence in this case, it is appropriate to say that our participants have "types", and that we can interpret the mixing probabilities ρ as the fraction of *participants* whose decisions are best represented by each model. Hence we can say things like "$100\rho_1$% of participants use model 1". On the other hand in a toolbox mixture model, participants can use all of the models on the menu for different decisions. In the toolbox case we should therefore interpret ρ as the fraction of *decisions* that are best represented by each model. Hence we can say things like "participants use model 1 $100\rho_1$% of the time".

7.3 Coding peculearities

This is a situation where we need to understand a bit about the kind of parameters that *Stan* can simulate. Specifically, they must be continuous random variables. This throws a bit of a spanner in the works when writing our *Stan* model, as the easy way to simulate a posterior for a mixture model is to use data augmentation, with the augmented data being categorical variables for which model the observation is being generated from. However since *Stan* cannot handle categorical parameters, we will instead have to integrate the likelihood.[31]

Another issue we will run into when computing the grand likelihood, we need to exponentiate the log-likelihood in order to incorporate the mixing probabilities, but then we need to then convert the grand likelihood back to log units in order to add it to *Stan*'s `target`. To avoid problems with exponentiating either very small or very large model likelihoods, we will use the following algebra trick:

[30] For simplicity, this assumes that all data are independent conditional on parameters (ρ, β) and all models.

[31] See the code in Bland and Rosokha (2019) for an example of estimating a 4-type mixture model using data augmentation with a Metropolis-Hastings within Gibbs sampler. The mixture model part of this is really simple, but it is not a specification that *Stan* will permit.

$$\log\left(\sum_{m=1}^{M}\rho_m\exp(l_m)\right)=\log\left(\sum_{m=1}^{M}\exp\left(\log\rho_m\right)\exp(l_m)\right)=\log\left(\sum_{m=1}^{M}\exp(l_m+\log\rho_m)\right)$$

which will allow us to use *Stan*'s `log_sum_exp` function to compute the grand likelihood.

7.4 Example experiment: Andreoni and Vesterlund (2001)

Andreoni and Vesterlund (2001) investigate whether there are any differences in other-regarding preferences between male and female participants. In their experiment, each participant was presented with eight different budget sets, which are shown in Figure 32.

```
AV2001choices<-read.csv("Data/AV2001choices.csv")
AV2001parameters<-read.csv("Data/AV2001parameters.csv")

(
  ggplot(AV2001parameters,
         aes(x=0,y=income*pOther,
             xend = income*pSelf,yend=0))
  +geom_segment()
  +theme_bw()
  #+xlim(c(0,100))+ylim(c(0,100))
  +coord_fixed()
  +geom_vline(xintercept=0,linewidth=1)
  +geom_hline(yintercept=0,linewidth=1)
  +xlab("Payoff to self")
  +ylab("Payoff to other")
)
```

For each of the budget sets, participants chose an allocation of money between themselves and another participant. As you can see, there is a good range of different trade-offs between the earnings of the self, and the earnings of the other. They frame this allocation task to participants as having an allocation of "tokens", and allocating these between themselves ("holding" the tokens) and another participant ("passing" the tokens). The tokens are then exchanged for dollars at different rates, hence the different slopes and intercepts of the budget lines above. In keeping with the data as I have it, I will use the following notation to refer to elements of the dataset:

- $\text{income}_{i,t}$ is the endowment measured in tokens
- $\text{pSelf}_{i,t}$ is the value of a token to the decision-maker (the "self")
- $\text{pOther}_{i,t}$ is the value of a token to the other participant (the "other")
- $y_{i,t}$ is the fraction of tokens that the decision-maker kept.

where $i \in \{1, 2, \ldots, N = 142\}$ indexes the participants, and $t \in \{1, 2, \ldots, T = 8\}$ indexes the budget set.

In their Section III.B, Andreoni and Vesterlund (2001) classify their participants into three types of other-regarding preferences. These three types are:

- *Selfish* preferences. These participants only care about their own payoff, and so will choose the allocation that maximizes their own earnings.
- *Perfect substitutes* preferences. These participants care about the total group earnings, and so will choose the allocation that maximizes this quantity. Since the budget lines are linear, this means choosing at one of the endpoints of the budget lines, unless $\text{pSelf}_{i,t} = \text{pOther}_{i,t}$, in which case anything is optimal.[32]

[32]This "no unique optimizer" result is not a problem for the logit choice model I implement below (the distribution of actions will just be uniform), however it *is* a problem for the Euclidean distance classifier. As far as I can tell, Andreoni and Vesterlund (2001) assume that when $\text{pSelf}_{i,t} = \text{pOther}_{i,t}$ the participant will share the tokens equally.

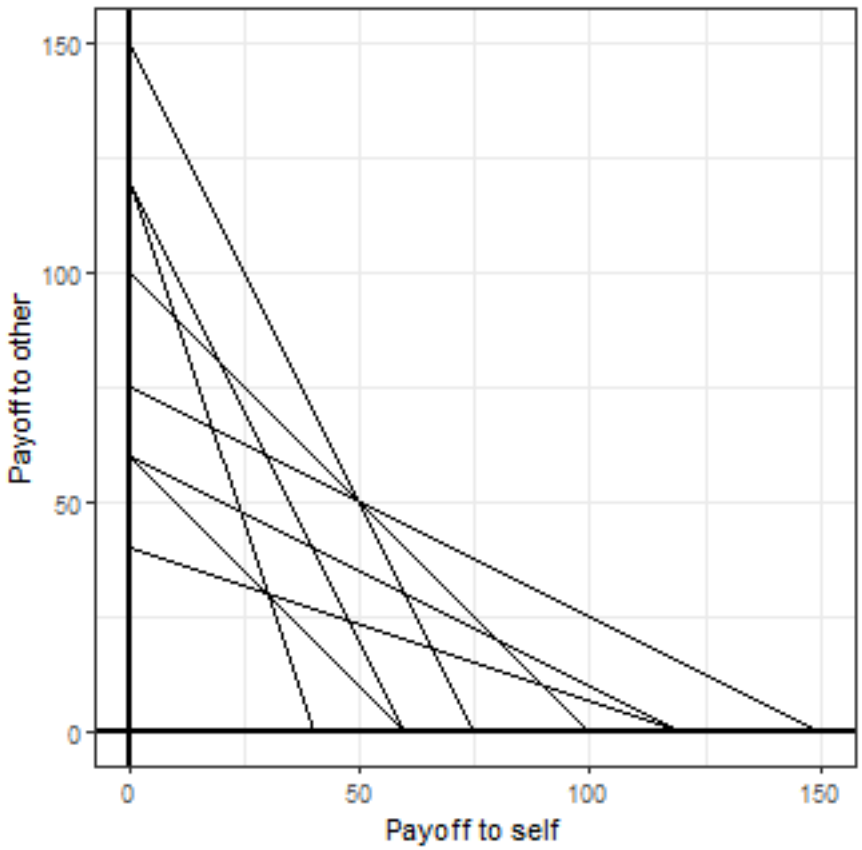


Figure 32: The eight budget sets presented to participants in Andreoni and Vesterlund (2001).

- *Perfect complements*, or *Leontief* preferences. These participants care about the minimum of their own and the other's payoff. They will choose the allocation that is at the intersection of the budget line and the 45° line.

We can represent these preferences as utility functions as follows:

$$\begin{aligned} U^S(\pi_s, \pi_o) &= \pi_s && \text{Selfish} \\ U^{PS}(\pi_s, \pi_o) &= \pi_s + \pi_o && \text{Perfect substitutes} \\ U^{PC}(\pi_s, \pi_o) &= \min\{\pi_s, \pi_o\} && \text{Perfect complements} \end{aligned}$$

These will be the three models that we will use for the mixture models below. Andreoni and Vesterlund (2001) classify their participants into one of these types by choosing the type that minimizes the Euclidean distance between the types' predictions and observed behavior. For this application, I will show you how to specify and estimate a mixture model that estimates the fraction of each type present in the data.[33]

While the above utility functions are sufficient to make deterministic predictions about how a participant might behave in this experiment, in order to turn this into a probabilistic model, we need to specify a probabilistic decision rule. For this, I will assume the logit choice rule, which is:

$$p(y_{i,t} \mid \rho, \lambda, m) = \frac{\exp(\lambda U^m(\text{income}_{i,t} y_{i,t} \text{pSelf}_{i,t}, \text{income}_{i,t}(1 - y_{i,t})\text{pOther}_{i,t}))}{\sum_{j \in \{\text{S,PS,PC}\}} \exp(\lambda U^j(\text{income}_{i,t} y_{i,t} \text{pSelf}_{i,t}, \text{income}_{i,t}(1 - y_{i,t})\text{pOther}_{i,t}))}$$

Noting that all three of the utility functions are homogeneous of degree 1, you will see the following representation show up in my code, which is mathematically equivalent, but saves a bit of time typing out:

$$p(y_{i,t} \mid \rho, \lambda, m) = \frac{\exp(\lambda \text{income}_{i,t} U^m(y_{i,t} \text{pSelf}_{i,t}, (1 - y_{i,t})\text{pOther}_{i,t}))}{\sum_{j \in \{\text{S,PS,PC}\}} \exp(\lambda \text{income}_{i,t} U^j(y_{i,t} \text{pSelf}_{i,t}, (1 - y_{i,t})\text{pOther}_{i,t}))}$$

This probabilistic choice rule, an assumption that all choices are independently distributed, and the choice of whether we are using a dichotomous or toolbox mixture model, pins down the grand likelihood function. For the dichotomous case, this is:

$$\log p(y \mid \rho, \lambda) = \sum_{i=1}^{n} \log \left(\sum_{m \in \{\text{S,PS,PC}\}} \exp \left(\sum_{t=1}^{T} \log p(y_{i,t} \mid \rho, \lambda, m) + \log \rho_m \right) \right)$$

Here we are assuming that participants' decisions can only ever be generated from one of the three models. This is why in dichotomous mixture models the individual models are often referred to as types: a participant is either either a selfish *type*, a perfect substitutes *type*, or a perfect complements *type*. This is not the case for the toolbox specification, where we allow participants to use all three models. In the case of the toolbox mixture model, the grand likelihood is:

$$\log p(y \mid \rho, \lambda) = \sum_{i=1}^{n} \sum_{t=1}^{T} \log \left(\sum_{m \in \{\text{S,PS,PC}\}} \exp(\log p(y_{i,t} \mid \rho, \lambda, m) + \log \rho_m) \right)$$

[33] Using a very similar experiment design, Andreoni and Miller (2002) also do this same classification of types. In and around their footnote 9, they report that "Bayesian algorithms" also yielded similar results.

7.4.1 As basic as it gets

To begin with, I will assume that λ is homogeneous so I can just show you how a mixture model is different from a representative agent model. This will probably lead to some strange results, as it is likely that participants are heterogeneous along this dimension, but it helps to see the actual mixture part of the mixture model.

Here is the *Stan* file I used for the dichotomous mixture model:

```
/*
AV2001_3typesDichotomous.stan
Mixture model assuming each participant is one type forever
*/

data {
  int<lower=0> N;
  int<lower=0> nParticipants;
  int<lower=0> id[N];
  int<lower=0> idStart[nParticipants];
  int<lower=0> idEnd[nParticipants];

  vector[N] income;
  vector[N] pSelf;
  vector[N] pOther;

  int binLength;

  vector[binLength] bin_grid;

  int bin_kept[N];

  vector[3] prior_mix;

  real prior_lambda[2];
}

transformed data {




}


parameters {

  simplex[3] mix;

  real<lower=0> lambda;

}

model {

  matrix[N,3] like_i;
```

```
  for (ii in 1:N) {

    // Selfish
    like_i[ii,1] = log_softmax(lambda*income[ii]*bin_grid*pSelf[ii])[bin_kept[ii]];
    // Perfect substituties
    like_i[ii,2] = log_softmax(lambda*income[ii]*(bin_grid*pSelf[ii]+(1.0-bin_grid)*pOther[ii]))[bin_kep
    // Perfect complements
    vector[binLength] UPC;

    UPC = fmin(bin_grid*pSelf[ii],(1.0-bin_grid)*pOther[ii]);
    like_i[ii,3] = log_softmax(lambda*income[ii]*UPC)[bin_kept[ii]];
  }


  for (ii in 1:nParticipants) {
    int iStart = idStart[ii];
    int iEnd   = idEnd[ii];

    vector[3] like_type = like_i[iStart:iEnd,]'*rep_vector(1.0,iEnd-iStart+1);

    target += log_sum_exp(like_type +log(mix));
  }


  // priors
  mix ~ dirichlet(prior_mix);
  lambda ~ lognormal(prior_lambda[1],prior_lambda[2]);

}
```

And here is the *Stan* file I used for the toolbox mixture model:

```
/*
AV2001_3types_toolbox.stan
Mixture model assuming each decision is one type
*/

data {
  int<lower=0> N;
  int<lower=0> nParticipants;
  int<lower=0> id[N];
  int<lower=0> idStart[nParticipants];
  int<lower=0> idEnd[nParticipants];

  vector[N] income;
  vector[N] pSelf;
  vector[N] pOther;

  int binLength;

  vector[binLength] bin_grid;

  int bin_kept[N];
```

```
  vector[3] prior_mix;

  real prior_lambda[2];
}

transformed data {




}


parameters {

  simplex[3] mix;

  real<lower=0> lambda;

}

model {

  matrix[N,3] like_i;

  for (ii in 1:N) {

    // Selfish
    like_i[ii,1] = log_softmax(lambda*income[ii]*bin_grid*pSelf[ii])[bin_kept[ii]];
    // Perfect substituties
    like_i[ii,2] = log_softmax(lambda*income[ii]*(bin_grid*pSelf[ii]+(1.0-bin_grid)*pOther[ii]))[bin_ke
    // Perfect complements
    vector[binLength] UPC;

    UPC = fmin(bin_grid*pSelf[ii],(1.0-bin_grid)*pOther[ii]);

    like_i[ii,3] = log_softmax(lambda*income[ii]*UPC)[bin_kept[ii]];

    target += log_sum_exp(like_i[ii,]'+log(mix));
  }



  // priors
  mix ~ dirichlet(prior_mix);
  lambda ~ lognormal(prior_lambda[1],prior_lambda[2]);

}
```

As you can see, there isn't much of a difference between them: only the way the model-specific likelihoods enter the grand likelihood. Note that the only difference between them is in the `model` block, and even then after all of the likelihoods for the individual models are calculated.

A problem I ran into when coding this was that the token endowment was different for different decisions.

As participants could choose an endowment in integer multiples of tokens, this means that the size of the choice set changes with the variable `income`. I decided to re-scale the problem by focusing on the derived variable $y_{i,t}$, which is the *fraction* of tokens kept by the participant. I then discretize this fraction into 20 evenly-spaced bins, which I call `bin_grid`, and generate the variable `bin_kept`, which identifies the closest fraction on the grid to the actual fraction $y_{i,t}$.

Now that we have the models coded up, we can move on to thinking about priors. As you can see in the above code, I have chosen a log-normal prior for λ. This makes sense because λ must be greater than zero. For the mixing probabilities, their support is the 3-dimensional simplex. A good choice of prior when we have a parameter that has a simplex support is the *Dirichlet* distribution, which is a generalization of the Beta distribution to more than two categories. That is, the Beta distribution is to binary variables as the Dirichlet distribution is to categorical variables. For ρ then, I choose a prior of:

$$\rho \sim \text{Dirichlet}\left(\begin{pmatrix}1 & 1 & 1\end{pmatrix}^\top\right)$$

which is reasonably spread out. For perspective, the marginal prior for the any of the types implied by this distribution is:

$$\rho^X \sim \text{Beta}(1,2), \quad X \in \{\text{S}, \text{PS}, \text{PC}\}$$

which has a prior mean of $E[\rho^X] = \frac{1}{1+2} = \frac{1}{3}$, so on average each type is equally likely.

While we might be much more interested in our estimates of ρ, we also need to be careful in choosing a prior for λ. To investigate what reasonable values of λ could be, we can plot the implied probability distribution of actions for several different values of λ. To get a feel for an appropriate scale for λ, I do this for several values of λ for pSelf $= 1$, pOther $= 2$ and income $= 50$. This is shown in Figure 33. Here it seems that there is enough plausible behavior happening in all of these panels, so we should choose a prior that does places a fair amount of mass over most of this range. The following prior achieves this, and centers the median prediction on the $\lambda = 1$ panels:

$$\log \lambda \sim N(\log 1, 1^2)$$

```
pSelf<-1
pOther<-2
income<-50

d<-expand_grid(fraction_kept = seq(0.025,0.975,by=0.05),
               lambda = c(0.01,0.1,1,10),
               type = c("S","PS","PC")) |>
  rowwise() |>
  mutate(LU = ifelse(type=="S",lambda*income*fraction_kept*pSelf,
              ifelse(type=="PS",lambda*income*(fraction_kept*pSelf+(1-fraction_kept)*pOther),
                     lambda*income*min(fraction_kept*pSelf,(1-fraction_kept)*pOther))
         )
    ) |>
  ungroup() |>
  arrange(type,lambda,fraction_kept) |>
  group_by(type,lambda) |>
  mutate(expLU = exp(LU-max(LU)),
         p = expLU/sum(expLU),
         lambda = paste("\u03bb =",lambda)
         )
```

```
(
  ggplot(data=d,aes(x=fraction_kept,y=p))
  +geom_path()
  +facet_grid(lambda ~ type,scales="free")
  +theme_bw()

)
```

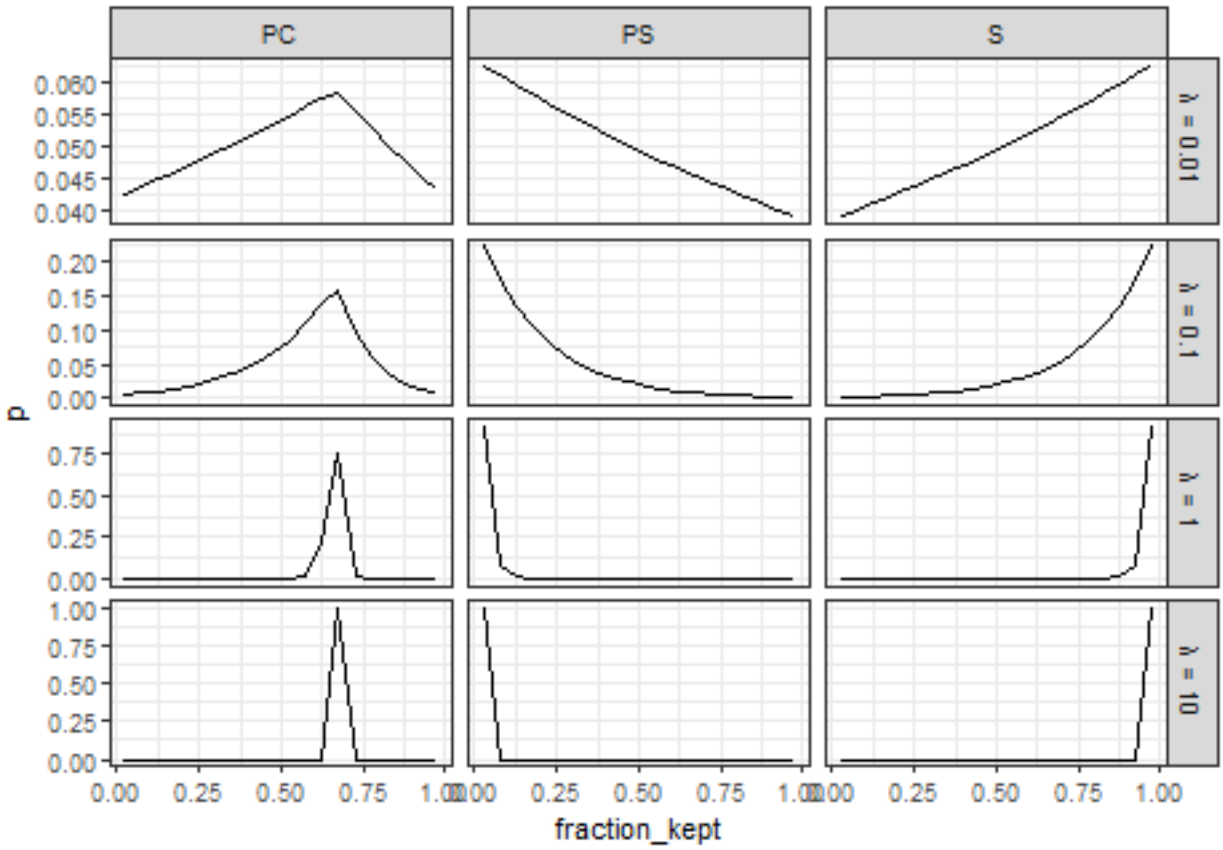


Figure 33: Implied choice probabilities for various values of λ.

Now that we have fully specified the likelihood and priors, we can go and estimate the model! Since Andreoni and Vesterlund (2001) were interested in differences in other-regarding preferences between male and female participants, I estimate both the dichotomous and toolbox models using all of the data, as well as estimating them just with the female participants, and just with the male participants. The estimates can be seen in Table 10.

```
rounding<-3

FileList<-list.files(path = "Code/MixtureModels")

FileList<-FileList[grepl("AV2001Dichotomous",FileList) | grepl("AV2001Toolbox",FileList)]

EstSum<-data.frame()

for (ff in FileList) {
  EstSum<-summary(readRDS(paste0("Code/MixtureModels/",ff)))$summary |>
```

Table 10: Estimates from the 3-type mixture model using data from Andreoni and Vesterlund (2001).

parameter	Toolbox Male	Toolbox Female	Toolbox Both	Dichotomous Male	Dichotomous Female	Dicho
Mix: S	0.468 (0.023)	0.386 (0.03)	0.44 (0.019)	0.01 (0.01)	0.507 (0.291)	0.453
Mix: PS	0.192 (0.022)	0.044 (0.022)	0.148 (0.017)	0.01 (0.01)	0.473 (0.29)	0.153
Mix: PC	0.34 (0.019)	0.57 (0.029)	0.412 (0.017)	0.98 (0.014)	0.02 (0.019)	0.394
lambda	0.19 (0.009)	0.158 (0.011)	0.18 (0.007)	0.377 (0.039)	2.515 (2.131)	0.137
lp___	-1730.788 (1.208)	-919.796 (1.248)	-2666.643 (1.259)	-169.494 (1.271)	-9.075 (1.329)	-2334.

```
    data.frame() |>
    mutate(file=ff,
           parameter = c("Mix: S","Mix: PS","Mix: PC","lambda","lp__")
           ) |>
    rbind(EstSum)
}

EstSum<-EstSum |>
  mutate(
    `Model type` = ifelse(grepl("Dichotomous",file),"Dichotomous","Toolbox"),
    Group = sub(".rds","",sub(".*_","",file)),
    msd = paste0(
      mean |> round(rounding),
      " (",sd |> round(rounding),")"
    )
  ) |>
  dplyr::select(c(parameter,`Model type`,Group,msd))|>
  pivot_wider(names_from = c(`Model type`,Group),values_from = msd,names_sep=" ")

EstSum |> kbl(caption = "Estimates from the 3-type mixture model using data from Andreoni and Vesterlund
```

Since the dichotomous models have a much more similar interpretation to the original nonlinear least squares classification of Andreoni and Vesterlund (2001), I will focus on these estimates. In particular, they are most similar to the "Total" columns in their Table III. Overall, the "Dichotomous Both" model (rightmost column) is estimating that we have about 45% selfish, 15% perfect substitutes, and 40% perfect complements participants. This is compared to 43% selfish, 21% perfect substitutes, and 35% perfect complements classified in Andreoni and Vesterlund (2001). Given some substantial differences in the estimation strategies used, these estimates are remarkably similar. However the similarity does not hold up when we slice the data by male and female participants. My dichotomous mixture model for example, estimates that almost all (98%) male participants are perfect complements types, whereas the equivalent number in Andreoni and Vesterlund (2001) is 25%. Also, for the female participants, I estimate about 47% are perfect substitutes types, but Andreoni and Vesterlund (2001) estimate this to be about 9%.

In this case, if I had to choose one of these estimation strategies, I would have gone with the original nonlinear least squares approach in Andreoni and Vesterlund (2001). This is mainly because I suspect that the assumption in my model that λ is homogeneous is a terrible one. In fact, Andreoni and Vesterlund (2001) note that they can classify 47% of male and 36% of female participants based on their choices *perfectly* matching one of the three types' predictions. This, coupled with the remaining fractions *not* being classified by perfectly matching predictions, suggests that there should be substantial heterogeneity in λ. I will extend the dichotomous model in the next section to account for this.

7.4.2 Adding some heterogeneity

From here, it is not too difficult to add some heterogeneity to λ by including a hierarchical component. Using a log-normal distribution to ensure that each λ_i is positive, the new population-level part of the model is:

$$\log \lambda_i \sim N(\mu_\lambda, \sigma_\lambda^2)$$

Since we now have this hierarchical structure on λ_i, we need to choose priors for μ_λ and σ_λ. For this application, I choose:

$$\mu_\lambda \sim N(0, 1)$$
$$\sigma_\lambda \sim \text{Cauchy}^+(0, 0.05)$$

Here is the *Stan* program that estimates the dichotomous model. For this section, I will just focus on the dichotomous model because it is most similar to the classification in Andreoni and Vesterlund (2001).

```
/*
AV2001_3types_dichotomous_hlambda.stan
Mixture model assuming each participant is one type forever
lambda is participant specific and log-normally distributed
*/

data {
  int<lower=0> N;
  int<lower=0> nParticipants;
  int<lower=0> id[N];
  int<lower=0> idStart[nParticipants];
  int<lower=0> idEnd[nParticipants];

  vector[N] income;
  vector[N] pSelf;
  vector[N] pOther;

  int binLength;

  vector[binLength] bin_grid;

  int bin_kept[N];

  vector[3] prior_mix;

  real prior_lambda_mean[2];
  real prior_lambda_sd;
}

transformed data {




}
```

```
parameters {

  simplex[3] mix;

  real lambdaMean;
  real<lower=0> lambdaSD;

  vector<lower=0>[nParticipants] lambda;

}

model {

  matrix[N,3] like_i;

  for (ii in 1:N) {

    // Selfish
    like_i[ii,1] = log_softmax(lambda[id[ii]]*income[ii]*bin_grid*pSelf[ii])[bin_kept[ii]];
    // Perfect substituties
    like_i[ii,2] = log_softmax(lambda[id[ii]]*income[ii]*(bin_grid*pSelf[ii]+(1.0-bin_grid)*pOther[ii]))
    // Perfect complements
    vector[binLength] UPC;

    UPC = fmin(bin_grid*pSelf[ii],(1.0-bin_grid)*pOther[ii]);
    like_i[ii,3] = log_softmax(lambda[id[ii]]*income[ii]*UPC)[bin_kept[ii]];
  }


  for (ii in 1:nParticipants) {
    int iStart = idStart[ii];
    int iEnd   = idEnd[ii];

    vector[3] like_type = like_i[iStart:iEnd,]'*rep_vector(1.0,iEnd-iStart+1);

    target += log_sum_exp(like_type +log(mix));
  }


  // priors
  mix ~ dirichlet(prior_mix);
  lambdaMean ~ normal(prior_lambda_mean[1],prior_lambda_mean[2]);
  lambdaSD ~ cauchy(0,prior_lambda_sd);


  // Hierarchical structure
  lambda ~ lognormal(lambdaMean,lambdaSD);

}
```

As before, I estimate the model using all of the data, as well as the data for male and female participants separately. Here is a posterior distribution summary table of these models:

Table 11: Estimates from the 3-type dichotomous mixture models with heterogeneous λ using data from Andreoni and Vesterlund (2001). Note that reported λ estimates are reported in log units.

parameter	Male	Female	Both
Mix: S	0.581 (0.055)	0.419 (0.073)	0.531 (0.045)
Mix: PS	0.181 (0.043)	0.072 (0.039)	0.139 (0.03)
Mix: PC	0.238 (0.048)	0.509 (0.074)	0.33 (0.041)
λ mean	-1.112 (0.239)	-1.442 (0.237)	-1.278 (0.169)
λ sd	2.094 (0.255)	1.537 (0.234)	1.868 (0.171)

```
rounding<-3
FileList<-list.files(path = "Code/MixtureModels")
FileList<-FileList[grepl("AV2001hlambdaDichotomous",FileList)]

EstSum<-data.frame()

SelectThis<-c("mix[1]","mix[2]","mix[3]","lambdaMean","lambdaSD")
labels<-c("Mix: S","Mix: PS","Mix: PC","$\\lambda$ mean","$\\lambda$ sd")
for (ff in FileList) {
  EstSum<-summary(readRDS(paste0("Code/MixtureModels/",ff)))$summary[SelectThis,] |>
    data.frame() |>
    mutate(parameter = labels,
           file = ff
           ) |>
    rbind(EstSum)
}

EstSum<-EstSum |>
  mutate(
    `Model type` = ifelse(grepl("Dichotomous",file),"Dichotomous","Toolbox"),
    Group = sub(".rds","",sub(".*_","",file)),
    msd = paste0(
      mean |> round(rounding),
      " (",sd |> round(rounding),")"
    )
  ) |>
  dplyr::select(parameter,Group,msd)|>
  pivot_wider(names_from = c(Group),values_from = msd)

EstSum |> kbl(caption = "Estimates from the 3-type dichotomous mixture models with heterogeneous $\\lam
```

Table 11 shows the estimates from the dichotomous mixture models that permit heterogeneity in choice precision λ. Here we can especially see a change from the homogeneous models in the estimates that slice the data into male and female participants. Very few female participants are now estimated to be perfect substitutes types (7%, compared to 9% in Andreoni and Vesterlund (2001)), but a substantial fraction of male participants are this type (18%, compared to 27% in Andreoni and Vesterlund (2001)). The fraction of selfish types is also similar, with the mixture models estimating 42% of female participants (37% in Andreoni and Vesterlund (2001)) and 58% of male participants (47% in Andreoni and Vesterlund (2001)).

7.5 Some code used to estimate the models

```
rm(list = ls())
library(tidyverse)
library(rstan)
options(mc.cores = parallel::detectCores())
rstan_options(auto_write = TRUE)

rounding_grid<-seq(0.025,0.975,by=0.05)

AV2001parameters<-read.csv("Data/AV2001parameters.csv") |>
  select(-X) |>
  mutate(decision = 1:n())

AV2001<-read.csv("Data/AV2001choices.csv") |>
  mutate(id = 1:n()) |>
  pivot_longer(cols = keep1:keep8,
               names_prefix = "keep",
               names_to = "decision",
               values_to = "amount_kept") |>
  mutate(decision = decision |> as.numeric()) |>
  left_join(AV2001parameters,by="decision") |>
  mutate(rownum = 1:n(),
         fraction_kept = amount_kept/income
         ) |>
  rowwise() |>
  mutate(bin_kept = which(abs(rounding_grid-fraction_kept)==min(abs(rounding_grid-fraction_kept)))[1])

idstartend<-AV2001 |> group_by(id) |>
  summarize(start = min(rownum),end=max(rownum))









Dichotomous<-stan_model("Code/MixtureModels/AV2001_3types_dichotomous.stan")
Toolbox<-stan_model("Code/MixtureModels/AV2001_3types_toolbox.stan")
Dichotomous_hlambda<-stan_model("Code/MixtureModels/AV2001_3types_dichotomous_hlambda.stan")
############################################################
#  mixture models using all of the data
############################################################

dStan<-list(
  N = dim(AV2001)[1],
  nParticipants = AV2001$id |> unique() |> length(),
  id = AV2001$id,
  idStart = idstartend$start,
  idEnd = idstartend$end,

  income = AV2001$income,
  pSelf = AV2001$pSelf,
  pOther = AV2001$pOther,
```

```
  binLength = length(rounding_grid),
  bin_grid = rounding_grid,
  bin_kept = AV2001$bin_kept,

  prior_mix = c(1,1,1),
  prior_lambda = c(log(1),1),

  prior_lambda_mean = c(log(1),1),
  prior_lambda_sd = 0.05

)

file<-"Code/MixtureModels/AV2001Dichotomous_Both.rds"
if (!file.exists(file)) {
  Fit<-sampling(Dichotomous,data=dStan,seed=42)
  Fit |> saveRDS(file)
}

file<-"Code/MixtureModels/AV2001Toolbox_Both.rds"
if (!file.exists(file)) {
  Fit<-sampling(Toolbox,data=dStan,seed=42)
  Fit |> saveRDS(file)
}

file<-"Code/MixtureModels/AV2001hlambdaDichotomous_Both.rds"
if (!file.exists(file)) {
  Fit<-sampling(Dichotomous_hlambda,data=dStan,seed=42)
  Fit |> saveRDS(file)
}

############################################################
# Mixture models using just the data from female participants
############################################################

d<-AV2001 |>
  filter(female==1) |>
  # I need to re-number the id variables so they go from 1:n only
  rename(idFull = id) |>
  ungroup() |>
  mutate(id = paste("id",idFull) |> as.factor() |> as.integer())

idstartend<-d |>
  mutate(rownum = 1:n()) |>
  group_by(id) |>
  summarize(start = min(rownum),end=max(rownum))

dStan<-list(
  N = dim(d)[1],
  nParticipants = d$id |> unique() |> length(),
  id = d$id,
  idStart = idstartend$start,
  idEnd = idstartend$end,
```

```
  income = d$income,
  pSelf = d$pSelf,
  pOther = d$pOther,

  binLength = length(rounding_grid),
  bin_grid = rounding_grid,
  bin_kept = d$bin_kept,

  prior_mix = c(1,1,1),
  prior_lambda = c(log(1),1),

  prior_lambda_mean = c(log(1),1),
  prior_lambda_sd = 0.05

)

file<-"Code/MixtureModels/AV2001Dichotomous_Female.rds"
if (!file.exists(file)) {
  Fit<-sampling(Dichotomous,data=dStan,seed=42)
  Fit |> saveRDS(file)
}

file<-"Code/MixtureModels/AV2001Toolbox_Female.rds"
if (!file.exists(file)) {
  Fit<-sampling(Toolbox,data=dStan,seed=42)
  Fit |> saveRDS(file)
}

file<-"Code/MixtureModels/AV2001hlambdaDichotomous_Female.rds"
if (!file.exists(file)) {
  Fit<-sampling(Dichotomous_hlambda,data=dStan,seed=42)
  Fit |> saveRDS(file)
}

############################################################
# Mixture models using just the data from male participants
############################################################

d<-AV2001 |>
  filter(female==0) |>
  # I need to re-number the id variables so they go from 1:n only
  rename(idFull = id) |>
  ungroup() |>
  mutate(id = paste("id",idFull) |> as.factor() |> as.integer())

idstartend<-d |>
  mutate(rownum = 1:n()) |>
  group_by(id) |>
  summarize(start = min(rownum),end=max(rownum))

dStan<-list(
  N = dim(d)[1],
  nParticipants = d$id |> unique() |> length(),
```

```
  id = d$id,
  idStart = idstartend$start,
  idEnd = idstartend$end,

  income = d$income,
  pSelf = d$pSelf,
  pOther = d$pOther,

  binLength = length(rounding_grid),
  bin_grid = rounding_grid,
  bin_kept = d$bin_kept,

  prior_mix = c(1,1,1),
  prior_lambda = c(log(1),1),

  prior_lambda_mean = c(log(1),1),
  prior_lambda_sd = 0.05

)

file<-"Code/MixtureModels/AV2001Dichotomous_Male.rds"
if (!file.exists(file)) {
  Fit<-sampling(Dichotomous,data=dStan,seed=42)
  Fit |> saveRDS(file)
}

file<-"Code/MixtureModels/AV2001Toolbox_Male.rds"
if (!file.exists(file)) {
  Fit<-sampling(Toolbox,data=dStan,seed=42)
  Fit |> saveRDS(file)
}

file<-"Code/MixtureModels/AV2001hlambdaDichotomous_Male.rds"
if (!file.exists(file)) {
  Fit<-sampling(Dichotomous_hlambda,data=dStan,seed=42)
  Fit |> saveRDS(file)
}
```

8 Model evaluation

It is often the case that we want to take more than one model to the data. While it is sometimes feasible to include all of your models in one big estimation (e.g. in a mixture model, or a model that nests all models under consideration), sometimes it is not computationally feasible to do this. In these cases, one approach is to compare your models in some way, and assign them some kind of measure of goodness-of-fit. These could include, but are certainly not limited to, assigning posterior probabilities to each model, and cross-validation, which I will tell you about in this chapter.

An important questions you will need to answer for yourself when deciding between the tools described below (or some secret other thing) is which technique best speaks to your research question? While it might seem that, philosophically, you might want to *always* use model posterior probabilities, because this is how Bayes' rule says we should do things, if we are trying to make out-of-sample predictions, then cross-validation might be more appropriate. It really depends on what you want to get out of your analysis.

8.1 Example dataset and models

Here I will take three models of other-regarding preferences to just the first participant to appear in the Bruhin, Fehr, and Schunk (2019) dataset. As a reminder from previous chapters, this experiment consists of 78 pair-wise choices of allocations between a decision-maker (the "self") and an inactive participant (the "other"). I will consider three models that could be generating the data.

The first, and most general of the three, is the Fehr and Schmidt (1999) model of inequality-aversion. This is the same model as I use in the "representative agent" chapter. For this, I took the *Stan* file from the representative agent models chapter, and made a few modifications that would allow me to do the model evaluations shown in this chapter. First, I added a data element in called `UseData`, which is a vector of ones and zeros the same length as the data (in this case 78 choices). If (say) the 36th element of this vector is equal to one, then the 36th decision in the data is counted toward the likelihood. On the other hand if this element is zero, then the model is estimated ignoring this observation. We will need this functionality to perform cross-validation. Hence, setting `UseData` to a vector of ones will exactly replicate the estimations in the representative agent models chapter, and setting `UseData` to a vector of zero will give you draws from the prior (because it will ignore all the data). Second, I explicitly computed the log-likelihood associated with each decision in a vector called `log_lik` in the `transformed parameters` block. Again, this is needed for cross-validation. Finally, I replaced sampling statements with a tilde (`~`) with an explicit `target += ...` statement. That is, whereas in the representative agent models chapter I would have had:

```
lambda ~ lognormal(prior_lambda[1],prior_lambda[2]);
```

I now have:

```
target += lognormal_lpdf(lambda | prior_lambda[1],prior_lambda[2]);
```

The difference between these sampling statements is that the former drops the log-normal density's constant of proportionality, while the latter does not. We will need this when we use `bridgesampling` to evaluate model posterior probabilities.

Here is the *Stan* program that estimates the Fehr and Schmidt (1999) model:

```
// FS.stan
data {
  int<lower=0> n; // number of observations
  vector[n] self_x; // payoff to self with allocation x
  vector[n] other_x; // payoff to other with allocation x
  vector[n] self_y; // payoff to self with allocation y
  vector[n] other_y; // payoff to other with allocation y
  int choice_x[n]; // =1 iff allocation x is chosen

  real prior_lambda[2];
  real prior_alpha[2];
  real prior_beta[2];

  vector[n] UseData;
}

transformed data {
  vector[n] dX; // disadvantageous inequality at allocation x
  vector[n] dY; // disadvantageous inequality at allocation y
  vector[n] aX; // advantageous inequality at allocation X
  vector[n] aY; // advantageous inequality at allocation X

  dX = fmax(0,other_x-self_x);
  dY = fmax(0,other_y-self_y);
```

```
  aX = fmax(0,self_x-other_x);
  aY = fmax(0,self_y-other_y);
}
parameters {
  real alpha;
  real beta;
  real<lower=0> lambda;
}
transformed parameters {

  vector[n] log_lik;

  for (ii in 1:n) {
    real Ux = self_x[ii]-alpha*dX[ii]-beta*aX[ii];
    real Uy = self_y[ii]-alpha*dY[ii]-beta*aY[ii];
    log_lik[ii] = bernoulli_logit_lpmf(choice_x[ii] | lambda*(Ux-Uy));
  }

}

model {


  // likelihood
  target += log_lik.*UseData;

  // priors
  target += normal_lpdf(alpha | prior_alpha[1],prior_alpha[2]);
  target += normal_lpdf(beta | prior_beta[1],prior_beta[2]);
  target += lognormal_lpdf(lambda | prior_lambda[1],prior_lambda[2]);

}
```

For this chapter, I consider two simplifications of the Fehr and Schmidt (1999) model. First, a model of pure selfishness that sets the aversion to inequality parameters equal to zero:

```
// Selfish.stan
data {
  int<lower=0> n; // number of observations
  vector[n] self_x; // payoff to self with allocation x
  vector[n] self_y; // payoff to self with allocation y
  int choice_x[n]; // =1 iff allocation x is chosen

  real prior_lambda[2];

  vector[n] UseData;
}


parameters {
  real<lower=0> lambda;
}
transformed parameters {
```

```
  vector[n] log_lik;

  for (ii in 1:n) {
    real Ux = self_x[ii];
    real Uy = self_y[ii];
    log_lik[ii] = bernoulli_logit_lpmf(choice_x[ii] | lambda*(Ux-Uy));
  }

}

model {


  // likelihood
  target += log_lik.*UseData;

  // priors
  target += lognormal_lpdf(lambda | prior_lambda[1],prior_lambda[2]);

}
```

And second, as an in-between model, I estimated a one-parameter ustility model that essentially imposes a restriction of $\alpha = -\beta$. This can capture efficiency-loving or efficiency-averse preferences, but not inequality-averse preferences:

```
// Linear.stan
data {
  int<lower=0> n; // number of observations
  vector[n] self_x; // payoff to self with allocation x
  vector[n] other_x; // payoff to other with allocation x
  vector[n] self_y; // payoff to self with allocation y
  vector[n] other_y; // payoff to other with allocation y
  int choice_x[n]; // =1 iff allocation x is chosen

  real prior_lambda[2];
  real prior_gamma[2];

  vector[n] UseData;
}

parameters {
  real gamma;
  real<lower=0> lambda;
}
transformed parameters {

  vector[n] log_lik;

  for (ii in 1:n) {
    real Ux = self_x[ii]+gamma*other_x[ii];
    real Uy = self_y[ii]+gamma*other_y[ii];
    log_lik[ii] = bernoulli_logit_lpmf(choice_x[ii] | lambda*(Ux-Uy));
  }
```

```
}

model {


  // likelihood
  target += log_lik.*UseData;

  // priors
  target += normal_lpdf(gamma | prior_gamma[1],prior_gamma[2]);
  target += lognormal_lpdf(lambda | prior_lambda[1],prior_lambda[2]);

}
```

For all of these models, I use the priors calibrated in the representative agent models chapter, which are:

$$\begin{aligned}\alpha, \beta, \gamma &\sim N(0, 0.67^2) \\ \log \lambda &\sim N(-5.76, 2.11^2)\end{aligned}$$

8.2 Model posterior probabilities

Bayes' rule actually provides us with a clear way of assigning posterior probabilities to models. To see this, take Bayes' rule in its general form for any two events A and B:

$$p(A \mid B) = \frac{p(B \mid A)p(A)}{p(B)}$$

and then substitute $A = M$ (i.e. the model) and $B = y$ (the data):

$$p(M \mid y) = \frac{p(y \mid M)p(M)}{p(y)}$$

That is, if we have a prior for each model, $p(M)$, and a likelihood of observing the data conditional on each model, $p(y \mid M)$, we can calculate the posterior probability of a model. The denominator can be integrated out by noting that:

$$\begin{aligned} p(M \mid y) &\propto p(y \mid M)p(M) \\ \implies p(M \mid Y) &= \frac{p(y \mid M)p(M)}{\sum_{m \in \mathcal{M}} p(y \mid m)p(m)} \end{aligned}$$

So we can assign a posterior probability to a model M in the set of models under consideration $\mathcal{M}$ if we can compute $p(y \mid M)$ and have formed a prior $p(M)$ for each model.

While a prior for a model is something that you need to come up with on your own, the likelihood component $p(y \mid M)$ comes straight from the marginal likelihood of each individual model. In principle, if you can estimate each model, then computing the marginal likelihood is feasible. To see this, note that we can notate Bayes' rule for a specific model as follows:

$$p(\theta \mid y, M) = \frac{p(y \mid \theta, M)p(\theta \mid M)}{p(y \mid M)}$$

So there is the marginal likelihood we are after right there in the denominator. A bit of re-arranging yields:

$$p(y \mid M) = \frac{p(y \mid \theta, M)p(\theta \mid M)}{p(\theta \mid y, M)}$$

Therefore in principle at least, if we can draw from the posterior $\theta \mid y, M$, we can compute the marginal likelihood $p(y \mid M)$.

Unfortunately, in practice doing this is somewhat difficult. To see this, the most intuitive (for me at least) way of evaluating $p(y \mid M)$ is to integrate out θ from $p(y \mid \theta, M)$, that is:

$$\begin{aligned} p(y \mid M) &= \int_{\Theta} p(y, \theta \mid, M) \mathrm{d}\theta \\ &= \int_{\Theta} p(y \mid \theta, M) p(\theta) \mathrm{d}\theta \\ &= E_{\theta}\left[p(y \mid \theta, M)\right] \end{aligned}$$

where the last line takes the expectation over prior θ. Therefore, we could approximate this marginal likelihood through simulation as:

$$\begin{aligned} p(y \mid M) &\approx \frac{1}{S} \sum_{s=1}^{S} p(y \mid \theta_s, M) \\ \{\theta_s\}_{s=1}^{S} &\sim p(\theta) \end{aligned}$$

where the θ_Ss are draws from the prior. However if the posterior is very different to the prior (and we hope that it would be because we hope to learn from our data), then this simulation will be very inefficient. This is because there will be many draws of θ_s that return very small likelihoods.

8.2.1 Implementation using bridge sampling and the `bridgesampling` library

A more efficient way to evaluate $p(y \mid M)$ is to use *bridge sampling* (Meng and Wong 1996). While we fortunately have a package in R that will just do this for us, it may be useful to see how it works. For my own understanding, I found Gronau et al. (2017) immensely useful in learning how this worked, and what follows closely mimics their explaination.

We start, as is the case with many algebraic problems, by constructing fancy but trivial identity:

$$1 = \frac{\int_{\Theta} p(y \mid \theta, M)p(\theta \mid M)g(\theta)h(\theta)\mathrm{d}\theta}{\int_{\Theta} p(y \mid \theta, M)p(\theta, M)g(\theta)h(\theta)\mathrm{d}\theta}$$

where $g(\theta)$ is a user-specified proposal distribution, and $h(\theta)$ is a user-specified "bridge function".

Next, we multiply both sides by $p(y \mid M)$:

$$
\begin{aligned}
p(y \mid M) &= p(y \mid M) \frac{\int_\Theta p(y \mid \theta, M) p(\theta \mid M) g(\theta) h(\theta) \mathrm{d}\theta}{\int_\Theta p(y \mid \theta, M) p(\theta, M) g(\theta) h(\theta) \mathrm{d}\theta} \\
&= \frac{\int_\Theta p(y \mid \theta, M) p(\theta \mid M) g(\theta) h(\theta) \mathrm{d}\theta}{\frac{1}{p(y|M)} \int_\Theta p(y \mid \theta, M) p(\theta, M) g(\theta) h(\theta) \mathrm{d}\theta} \\
&= \frac{\int_\Theta p(y \mid \theta, M) p(\theta \mid M) g(\theta) h(\theta) \mathrm{d}\theta}{\int_\Theta \frac{p(y|\theta,M)p(\theta|M)}{p(y|M)} g(\theta) h(\theta) \mathrm{d}\theta} \\
&= \frac{\int_\Theta p(y \mid \theta, M) p(\theta \mid M) g(\theta) h(\theta) \mathrm{d}\theta}{\int_\Theta \frac{p(y,\theta|M)}{p(y|M)} g(\theta) h(\theta) \mathrm{d}\theta} \\
&= \frac{\int_\Theta p(y \mid \theta, M) p(\theta \mid M) g(\theta) h(\theta) \mathrm{d}\theta}{\int_\Theta p(\theta \mid y, M) g(\theta) h(\theta) \mathrm{d}\theta}
\end{aligned}
$$

Note now that both the numerator and denominator can be written as expectations. For the numerator we will view this as an expectation relative to the proposal density $g(\theta)$, and for the denominator we will view this as an expectation relative to the posterior $p(\theta \mid y, M)$, so we can write:

$$
p(y \mid M) = \frac{E_{\theta \sim g(\theta)} \left[p(y \mid \theta, M) p(\theta \mid M) h(\theta) \right]}{E_{\theta \sim p(\theta|y,M)} \left[g(\theta) h(\theta) \right]}
$$

Therefore we can approximate $p(y \mid M)$ as follows:

$$
\begin{aligned}
p(y \mid M) &\approx \frac{\frac{1}{S} \sum_{s=1}^{S} p(y \mid \hat{\theta}_s, M) p(\hat{\theta}_s \mid M) h(\hat{\theta}_s)}{\frac{1}{T} \sum_{t=1}^{T} g(\tilde{\theta}_t) h(\tilde{\theta}_t)} \\
\hat{\theta}_s &\sim g(\theta) \\
\tilde{\theta}_t &\sim p(\theta \mid y, M)
\end{aligned}
$$

That is, the $\hat{\theta}$s are draws from the proposal distribution, and the $\tilde{\theta}$s are draws from the posterior.

Fortunately for us, the `bridgesampling` library (Gronau, Singmann, and Wagenmakers 2020) does all of this for us. The only thing we need to be careful of in writing our *Stan* files is that we do not drop constants of proportionality, which I do in the *Stan* programs shown above.[34] Here is how we can assign posterior probabilities to our three models using the `bridgesampling` library:

```
library(tidyverse)
library(bridgesampling)

d1<-(read.csv("Data/BFS2019_choices_exp1.csv")
     |> mutate(experiment=1)
)
d2<-(read.csv("Data/BFS2019_choices_exp2.csv")
     |> mutate(experiment=2)
)
D<-(rbind(d1,d2)
    # Just use the dictator game data
    |> filter(dg==1)
    |> mutate(
      self_alloc = ifelse(choice_x==1,self_x,self_y),
      other_alloc = ifelse(choice_x==1,other_x,other_y)
```

[34] When using a sampling statement with a tilde (~), *Stan* drops constants of proportionality because they are not needed to simulate the posterior. However they *are* needed for computing the marginal likelihood of a model.

```
    )
) |>
  filter(
    sid==102010050706
  )



model_FS<-"Code/Validation/FS.stan" |>
  stan_model()
model_Linear<-"Code/Validation/Linear.stan" |>
  stan_model()
model_Selfish<-"Code/Validation/Selfish.stan" |>
  stan_model()

dStan<-list(
  n = dim(D)[1],
  self_x = D$self_x,
  other_x = D$other_x,
  self_y = D$self_y,
  other_y = D$other_y,

  choice_x = D$choice_x,

  prior_alpha = c(0,0.67),
  prior_beta = c(0,0.67),
  prior_gamma = c(0,0.67),
  prior_lambda = c(-5.76,2.11),

  UseData = rep(1,dim(D)[1])
)

# Estimate the models

Fit_FS<-model_FS |>
  sampling(data=dStan,seed=42)

Fit_Selfish<-model_Selfish |>
  sampling(data=dStan,seed=42)

Fit_Linear<-model_Linear |>
  sampling(data=dStan,seed=42)

# Apply the bridge sampler to the estimated models

FS<-Fit_FS |>
  bridge_sampler()

Selfish<-Fit_Selfish |>
  bridge_sampler()

Linear<-Fit_Linear |>
  bridge_sampler()
```

Table 12: Posterior probabilities of the three models assuming equal prior probabilities.

	x
FS	0.0742
Selfish	0.7706
Linear	0.1552

```
# Calculate a posterior probability for each model assuming an equal prior

post_prob(FS,Selfish,Linear) |>
  saveRDS("Code/Validation/PostProbs.rds")
```

```
"Code/Validation/PostProbs.rds" |>
  readRDS() |>
  kbl(digits = 4,caption = "Posterior probabilities of the three models assuming equal prior probabilit
  kable_classic(full_width=FALSE)
```

Table 12 shows the three models' posterior probabilities assuming equal prior probabilities. Here we can see that most of the probability mass is assigned to the selfish model. That is, if we wanted to pick one model out of these three, we would choose the selfish model. Another, more Bayesian interpretation of the results in this Table is as follows: suppose that these three models are the only three models that could be generating the data, then after observing the data our belief that the selfish model is the true model is 77% (assuming equal prior probabilities).

8.3 Cross-validation

Cross-validation techniques, while not strictly Bayesian,[35] are useful when you are interested in making out-of-sample predictions. Cross-validation mimics an out-of-sample prediction by splitting the dataset into a "training" dataset and a "testing" dataset. The training dataset is used to estimate the model, and then the testing dataset is used to evaluate the model. That is, we estimate the model ignoring the testing data, and then see how well the estimated model performs in predicting it.

8.3.1 Expected Log Predicted Density (ELPD) and other measures of goodness of fit

An important component to cross-validation is the choice of goodness-of-fit: we need to have a measure of how well, or how badly, our models fit the data. If we were in the realm of linear regression, a good starting point could be out-of-sample residual sum of squares:

$$RSS_M = \sum_{i,t \in \text{testing}} (y_{i,t} - X_{i,t}\hat{\beta}_M)^2$$

That is, we would aim to select the model $M \in \mathcal{M}$ that minimized RSS_M.

Things become a little more complicated when we are dealing with choice data $y_{i,t}$ that are not real numbers. For instance, many of the examples I use in this book, and many experiments in general, have *binary* choices. It is not clear whether minimizing the sum of squared residuals for binary data is what we want to be doing. Furthermore, even if we had continuous data, RSS_M only evaluates the model's *point predictions* (i.e. $\hat{y}_{i,t,M} = X_{i,t}\hat{\beta}_M$), and completely ignores that our model makes *probabilistic* predictions. From the linear regression model's perspective we are ignoring that, when coupled with (say) a distributional assumption of normal errors, the model also says something about the spread of data.

[35] I mean two things here. Firstly, unlike posterior probabilities, we are not using Bayes' rule to quantify uncertainty in our models. Secondly, cross-validation can equally be applied using Frequentist techniques.

In the absence of a specific predictive goal,[36] a good all-encompassing measure of goodness-of-fit is Expected Log Predicted Density (ELPD), which is defined as:

$$\text{ELPD}_M = E_{\theta|y,M}\left[\log p(y^* \mid \theta, M)\right]$$

where $\theta \mid y, M$ is the posterior distribution of parameters θ from model M, and $p(y^* \mid \theta, M)$ is the probability (mass or density) of observing (out-of-sample) observation y^* given model M and parameters θ. My (somewhat) lay-person's translation of ELPD_M is: "If we estimate and use model M, how un-surprised will we be by new data?" That is, if ELPD_M is large, then our model on average assigns a lot of probability (mass or density) to the new data that we will get, and so we are often not surprised by our new data.

Since y^* is unknown at the time of estimation, in practice we approximate it using cross-validation. We use the testing dataset to compute it:

$$\begin{aligned}
\text{ELPD}_M &\approx \frac{1}{n_{\text{testing}}} \sum_{i,t \in \text{testing}} E_{\theta|M}\left[\log p(y_{i,t} \mid \theta, M)\right] \\
&\approx \frac{1}{n_{\text{testing}}} \sum_{i,t \in \text{testing}} \left[\frac{1}{S}\sum_{s=1}^{S} \log p(y_{i,t} \mid \theta_s, M)\right] \\
\theta_s &\sim p(\theta \mid y_{\text{training}}, M)
\end{aligned}$$

That is, we are evaluating the out-of-sample (i.e. testing) log-likelihood of our model with respect to the posterior (conditional on the training data only). Hence you will see in the in the *Stan* programs above, that I have explicitly created a `log_lik` vector in the `transformed parameters` block. This will make it easy to extract the relevent elements of this when we do cross-validation.

I will proceed with the remainder of this chapter using ELPD as the measure of goodness-of-fit, but realize that what follows applies equally to any other valid measure.

8.3.2 1-round cross-validation

The simplest and fastest form of cross-validation in 1-round cross-validation. Here, we do the cross-validation process exactly once. We partition the sample into a "training" dataset, where we estimate the model, and a "testing" dataset, which we use to evaluate the model. We do this exactly once. The other cross-validation techniques mentioned later on in this chapter will involve repeating this many times.

Here is my code to evaluate ELPD for the three models using 1-round cross-validation. Note that I achieve this by setting ten random elements of the `UseData` vector to zero. This means that these observations are not used in incrementing the target (see the `target += log_like .* UseData;` line in the *Stan* programs), and so don't contribute to the estimation process. I then extract the log-likelihoods corresponding to these testing data to compute ELPD.

```
library(tidyverse)
library(rstan)
options(mc.cores = parallel::detectCores())
rstan_options(auto_write = TRUE)


d1<-(read.csv("Data/BFS2019_choices_exp1.csv")
     |> mutate(experiment=1)
)
d2<-(read.csv("Data/BFS2019_choices_exp2.csv")
```

[36] For example, if you want to make predictions that minimize the expected squared error, then RSS_M may be a great choice for you.

```
    |> mutate(experiment=2)
)
D<-(rbind(d1,d2)
    # Just use the dictator game data
    |> filter(dg==1)
    |> mutate(
      self_alloc = ifelse(choice_x==1,self_x,self_y),
      other_alloc = ifelse(choice_x==1,other_x,other_y)
    )
) |>
  filter(
    sid==102010050706
  )

model_FS<-"Code/Validation/FS.stan" |>
  stan_model()
model_Linear<-"Code/Validation/Linear.stan" |>
  stan_model()
model_Selfish<-"Code/Validation/Selfish.stan" |>
  stan_model()

dStan<-list(
  n = dim(D)[1],
  self_x = D$self_x,
  other_x = D$other_x,
  self_y = D$self_y,
  other_y = D$other_y,

  choice_x = D$choice_x,

  prior_alpha = c(0,0.67),
  prior_beta = c(0,0.67),
  prior_gamma = c(0,0.67),
  prior_lambda = c(-5.76,2.11),

  UseData = rep(1,dim(D)[1])
)

# split the data into training and testing subsamples
# Here we will randomly select 10 observations to hold out for the testing data
set.seed(42)
training<-(1:78)[order(runif(78))][1:10]
dStan$UseData[training]<-0


# Estimate the models

Fit_FS<-model_FS |>
  sampling(data=dStan,seed=42)

Fit_Selfish<-model_Selfish |>
  sampling(data=dStan,seed=42)
```

Table 13: ELPD evaluated using 1-round cross validation.

model	ELPD
FS	-6.30
Selfish	-6.32
Linear	-6.25

```
Fit_Linear<-model_Linear |>
  sampling(data=dStan,seed=42)

# extract the relevant parts of the log-likelihood matrix for the training data

ELPD_FS<-mean(extract(Fit_FS)$log_lik[,training]%*% rep(1,10))
ELPD_Selfish<-mean(extract(Fit_Selfish)$log_lik[,training]%*% rep(1,10))
ELPD_Linear<-mean(extract(Fit_Linear)$log_lik[,training]%*% rep(1,10))

d<-tibble(
  model = c("FS","Selfish","Linear"),
  ELPD  = c(ELPD_FS,ELPD_Selfish,ELPD_Linear)
)

d |>
  write.csv("Code/Validation/OneRound.csv")
```

```
"Code/Validation/OneRound.csv" |>
  read.csv() |>
  dplyr::select(-X) |>
  kbl(digits=2,caption = "ELPD evaluated using 1-round cross validation.") |>
  kable_classic(full_width=FALSE)
```

Table 13 shows the ELPD evaluated using 1-round cross-validation. More positive numbers indicate better out-of-sample predictions. Here we can see that we (marginally) select the FS model over the others.

8.3.3 Leave-one-out cross-validation (LOO)

Leave-one-out (LOO) cross-validation is *much* more computationally intensive than 1-round, but is preferable because it better mimics the estimation process and does not depend on the definition of the testing and training datasets. Here, we estimate the model n times, once for each observation in the dataset. Each time, we put just one observation into the testing dataset and evaluate its goodness-of-fit. We then take the average goodness-of-fit over these n estimation runs. In the code below, you can see I loop over the observations in the data (`for (ii in 1:dim(D)[1]) { ...`). Each time through the loop, element `ii` of `UseData` is set to zero. We then extract the log-likelihood for this left-out observation and store it in a matrix (e.g. `extract(Fit_FS)$log_lik[,ii]`) I then use the `loo` library to summarize the results.

```
library(tidyverse)
library(loo)

d1<-(read.csv("Data/BFS2019_choices_exp1.csv")
     |> mutate(experiment=1)
)
d2<-(read.csv("Data/BFS2019_choices_exp2.csv")
     |> mutate(experiment=2)
)
D<-(rbind(d1,d2)
```

```
    # Just use the dictator game data
    |> filter(dg==1)
    |> mutate(
      self_alloc = ifelse(choice_x==1,self_x,self_y),
      other_alloc = ifelse(choice_x==1,other_x,other_y)
    )
) |>
  filter(
    sid==102010050706
  )



model_FS<-"Code/Validation/FS.stan" |>
  stan_model()
model_Linear<-"Code/Validation/Linear.stan" |>
  stan_model()
model_Selfish<-"Code/Validation/Selfish.stan" |>
  stan_model()

dStan<-list(
  n = dim(D)[1],
  self_x = D$self_x,
  other_x = D$other_x,
  self_y = D$self_y,
  other_y = D$other_y,

  choice_x = D$choice_x,

  prior_alpha = c(0,0.67),
  prior_beta = c(0,0.67),
  prior_gamma = c(0,0.67),
  prior_lambda = c(-5.76,2.11),

  UseData = rep(1,dim(D)[1])
)

ll_FS<-c()
ll_Selfish<-c()
ll_Linear<-c()

for (ii in 1:dim(D)[1]) {

  print(paste(ii,"of",dim(D)[1]))

  dStan<-list(
    n = dim(D)[1],
    self_x = D$self_x,
    other_x = D$other_x,
    self_y = D$self_y,
    other_y = D$other_y,

    choice_x = D$choice_x,
```

Table 14: Model comparison using exact LOO

model	elpd_diff	se_diff	elpd_loo	se_elpd_loo	p_loo	se_p_loo	looic	se_looic
Selfish	0.00000	0.0000000	-41.71713	6.029357	1.794600	0.8102173	83.43425	12.05871
Linear	-2.14477	0.7093951	-43.86190	5.823461	2.979981	0.9124914	87.72379	11.64692
FS	-4.65318	1.4090278	-46.37031	6.316032	4.604015	1.3196564	92.74061	12.63206

```
    prior_alpha = c(0,0.67),
    prior_beta = c(0,0.67),
    prior_gamma = c(0,0.67),
    prior_lambda = c(-5.76,2.11),

    UseData = rep(1,dim(D)[1])
  )

  dStan$UseData[ii]<-0

  Fit_FS<-model_FS |>
    sampling(data=dStan,seed=42)
  ll_FS<-cbind(ll_FS,extract(Fit_FS)$log_lik[,ii])

  Fit_Linear<-model_Linear |>
    sampling(data=dStan,seed=42)
  ll_Linear<-cbind(ll_Linear,extract(Fit_Linear)$log_lik[,ii])

  Fit_Selfish<-model_Selfish |>
    sampling(data=dStan,seed=42)
  ll_Selfish<-cbind(ll_Selfish,extract(Fit_Selfish)$log_lik[,ii])

}

ll_FS |>
  write.csv("Code/Validation/LOO_ll_FS.csv")
ll_Linear |>
  write.csv("Code/Validation/LOO_ll_Linear.csv")
ll_Selfish |>
  write.csv("Code/Validation/LOO_ll_Selfish.csv")
```

```
models<-c(model1="FS",model2="Selfish",model3="Linear")

LOO_ll_FS<-"Code/Validation/LOO_ll_FS.csv" |> read.csv() |> dplyr::select(-X) |> as.matrix()
LOO_ll_Linear<-"Code/Validation/LOO_ll_Linear.csv" |> read.csv() |> dplyr::select(-X) |> as.matrix()
LOO_ll_Selfish<-"Code/Validation/LOO_ll_Selfish.csv" |> read.csv() |> dplyr::select(-X) |> as.matrix()

loo_compare(loo(LOO_ll_FS,cores=8),loo(LOO_ll_Selfish,cores=8),loo(LOO_ll_Linear,cores=8)) |>
  data.frame()|>
  rownames_to_column(var = "model") |>
  mutate(
    model = models[model]
  ) |>
  kbl( caption = "Model comparison using exact LOO") |>
  kable_classic(full_width=FALSE)
```

Table 14 shows a summary of the `loo` output. Recall that larger (i.e. more positive) values of ELPD indicate a better fit. Here we select the selfish model.

8.3.4 Approximate LOO

Vehtari, Gelman, and Gabry (2017) describes methods for approximating LOO using Pareto-smoothed importance sampling. The advantage of this process is that you only need to estimate each model once, rather than n times. This can be done in R using the `loo` library.

```
library(tidyverse)
library(loo)
library(rstan)
options(mc.cores = parallel::detectCores())
rstan_options(auto_write = TRUE)

d1<-(read.csv("Data/BFS2019_choices_exp1.csv")
     |> mutate(experiment=1)
)
d2<-(read.csv("Data/BFS2019_choices_exp2.csv")
     |> mutate(experiment=2)
)
D<-(rbind(d1,d2)
    # Just use the dictator game data
    |> filter(dg==1)
    |> mutate(
      self_alloc = ifelse(choice_x==1,self_x,self_y),
      other_alloc = ifelse(choice_x==1,other_x,other_y)
    )
) |>
  filter(
    sid==102010050706
  )



model_FS<-"Code/Validation/FS.stan" |>
  stan_model()
model_Linear<-"Code/Validation/Linear.stan" |>
  stan_model()
model_Selfish<-"Code/Validation/Selfish.stan" |>
  stan_model()

dStan<-list(
  n = dim(D)[1],
  self_x = D$self_x,
  other_x = D$other_x,
  self_y = D$self_y,
  other_y = D$other_y,

  choice_x = D$choice_x,

  prior_alpha = c(0,0.67),
  prior_beta = c(0,0.67),
  prior_gamma = c(0,0.67),
  prior_lambda = c(-5.76,2.11),
```

```
  UseData = rep(1,dim(D)[1])
)

# Estimate the models

Fit_FS<-model_FS |>
  sampling(data=dStan,seed=42)

Fit_Selfish<-model_Selfish |>
  sampling(data=dStan,seed=42)

Fit_Linear<-model_Linear |>
  sampling(data=dStan,seed=42)

# apply approximate LOO cross-validation

FS<-extract(Fit_FS)$log_lik |>
  loo(cores=8)


Selfish<-extract(Fit_Selfish)$log_lik |>
  loo(cores=8)

Linear<-extract(Fit_Linear)$log_lik |>
  loo(cores=8)

# Compare the models

loo_compare(FS,Selfish,Linear) |>
  saveRDS("Code/Validation/ApproxLOO.rds")
```

The downside of this process is that it is not guaranteed that the sampling is reliable. Fortunately there is a diagnostic of this, which 'loo' checks for us. For example, for the "Selfish" model, I got the following warning:

```
Pareto k diagnostic values:
                         Count Pct.    Min. ESS
(-Inf, 0.7]   (good)     72    92.3%   2270
   (0.7, 1]   (bad)       0     0.0%   <NA>
   (1, Inf)   (very bad)  6     7.7%   <NA>
See help('pareto-k-diagnostic') for details.
```

So there are six observations for which this process did not work well. In these cases one can instead do LOO as described in the previous section.

```
"Code/Validation/ApproxLOO.rds" |>
  readRDS() |>
  data.frame()|>
  rownames_to_column(var = "model") |>
  mutate(
    model = models[model]
  ) |>
  kbl( caption = "Model validation using approximate LOO") |>
  kable_classic(full_width=FALSE)
```

Table 15 shows the `loo` library's summary of the model evaluation. Here we select the "Selfish" model.

Table 15: Model validation using approximate LOO

model	elpd_diff	se_diff	elpd_loo	se_elpd_loo	p_loo	se_p_loo	looic	se_looic
Selfish	0.0000000	0.0000000	-39.93907	5.310491	1.467190	0.6213825	79.87814	10.62098
Linear	-0.9970267	0.7318856	-40.93610	5.009543	2.454233	0.6737671	81.87220	10.01909
FS	-1.7310278	1.0247346	-41.67010	5.095779	3.476072	0.8952323	83.34020	10.19156

8.3.5 k-fold cross-validation

Perhaps a happy medium between LOO and approximate LOO is k-fold cross-validation. Here, we partition the data into k "folds". For each fold, we estimate the model using data not in that fold, then evaluate how well the estimated model predicts data in that fold. That is, when $k = n$ we are back to LOO.

Here is my R code for implementing k-fold cross-validation. In this script, I partition the data into $k = 10$ folds.

```
library(tidyverse)
library(rstan)
options(mc.cores = parallel::detectCores())
rstan_options(auto_write = TRUE)

d1<-(read.csv("Data/BFS2019_choices_exp1.csv")
     |> mutate(experiment=1)
)
d2<-(read.csv("Data/BFS2019_choices_exp2.csv")
     |> mutate(experiment=2)
)
D<-(rbind(d1,d2)
    # Just use the dictator game data
    |> filter(dg==1)
    |> mutate(
      self_alloc = ifelse(choice_x==1,self_x,self_y),
      other_alloc = ifelse(choice_x==1,other_x,other_y)
    )
) |>
  filter(
    sid==102010050706
  )



model_FS<-"Code/Validation/FS.stan" |>
  stan_model()
model_Linear<-"Code/Validation/Linear.stan" |>
  stan_model()
model_Selfish<-"Code/Validation/Selfish.stan" |>
  stan_model()

dStan<-list(
  n = dim(D)[1],
  self_x = D$self_x,
  other_x = D$other_x,
  self_y = D$self_y,
  other_y = D$other_y,
```

```
  choice_x = D$choice_x,

  prior_alpha = c(0,0.67),
  prior_beta = c(0,0.67),
  prior_gamma = c(0,0.67),
  prior_lambda = c(-5.76,2.11),

  UseData = rep(1,dim(D)[1])
)

ll_FS<-matrix(NA,4000,78)
ll_Selfish<-matrix(NA,4000,78)
ll_Linear<-matrix(NA,4000,78)

set.seed(42)


# set up an index for the folds.
folds<-rep(1:10,8)
folds<-folds[order(runif(80))][1:78]



for (ff in unique(folds)) {

  print(ff)

  dStan<-list(
    n = dim(D)[1],
    self_x = D$self_x,
    other_x = D$other_x,
    self_y = D$self_y,
    other_y = D$other_y,

    choice_x = D$choice_x,

    prior_alpha = c(0,0.67),
    prior_beta = c(0,0.67),
    prior_gamma = c(0,0.67),
    prior_lambda = c(-5.76,2.11),

    UseData = rep(1,dim(D)[1])
  )

  dStan$UseData[folds==ff]<-0

  Fit_FS<-model_FS |>
    sampling(data=dStan,seed=42)
  ll_FS[,ff==folds]<-extract(Fit_FS)$log_lik[,ff==folds]

  Fit_Linear<-model_Linear |>
    sampling(data=dStan,seed=42)
  ll_Linear[,ff==folds]<-extract(Fit_Linear)$log_lik[,ff==folds]
```

Table 16: Model comparison using k-fold cross-validation

model	elpd_diff	se_diff	elpd_loo	se_elpd_loo	p_loo	se_p_loo	looic	se_looic
Selfish	0.000000	0.0000000	-42.31369	6.220406	2.098414	0.9582934	84.62739	12.44081
Linear	-1.810640	0.8332116	-44.12433	5.871708	3.168026	0.9462041	88.24867	11.74342
FS	-4.303829	1.6452414	-46.61752	6.330098	4.929662	1.3753267	93.23504	12.66020

```
  Fit_Selfish<-model_Selfish |>
    sampling(data=dStan,seed=42)
  ll_Selfish[,ff==folds]<-extract(Fit_Selfish)$log_lik[,ff==folds]

}

ll_FS |>
  write.csv("Code/Validation/kfold_ll_FS.csv")
ll_Linear |>
  write.csv("Code/Validation/kfold_ll_Linear.csv")
ll_Selfish |>
  write.csv("Code/Validation/kfold_ll_Selfish.csv")
```

```
kfold_ll_FS<-"Code/Validation/kfold_ll_FS.csv" |> read.csv() |> dplyr::select(-X) |> as.matrix()
kfold_ll_Linear<-"Code/Validation/kfold_ll_Linear.csv" |> read.csv() |> dplyr::select(-X) |> as.matrix()
kfold_ll_Selfish<-"Code/Validation/kfold_ll_Selfish.csv" |> read.csv() |> dplyr::select(-X) |> as.matri


loo_compare(loo(kfold_ll_FS,cores=8),loo(kfold_ll_Selfish,cores=8),loo(kfold_ll_Linear,cores=8)) |>
  data.frame()|>
  rownames_to_column(var = "model") |>
  mutate(
    model = models[model]
  ) |>
  kbl( caption = "Model comparison using $k$-fold cross-validation") |>
  kable_classic(full_width=FALSE)
```

Table 16 shows the `loo` library's summary of the model evaluation using k-fold cross-validation. Again, we select the Selfish model.

9 Speeding up your *Stan* code

While Bayesian computation certainly isn't for the impatient, there are some things you can do to make sure you are not waiting any longer than you have to for your results. I this chapter, I will show you some ways to write code that runs faster. In particular, we will learn about recognizing opportunities to pre-compute values, how to vectorize operations, and how to take advantage of *Stan*'s within-chain parallel processing capabilities. The first two of these will help on any machine that you might be using, whereas parallelization might help in specific instances where you have some idle cores that could be put to use.

9.1 Example dataset and model

In this chapter we will be estimating a two-parameter Expected Utility Theory (EUT) model for each of the 327 participants of Harrison and Swarthout (2023). In this experiment, each participant made 100 binary lottery choices. Each lottery pair can be described by three monetary prizes (which were common for both lotteries in the pair), and the probabilities of winning each prize. Let $x_{t,k}$ be the kth prize in lottery pair t,

and $q^L_{t,k}$ and $q^R_{t,k}$ are the probabilities of winning prize k for the "Left" and "Right" lottery in pair t. Assuming a constant relative risk-aversion specification, we can then write down the expected utility of the Left lottery in pair t as:

$$Eu(x_t, q^L_t \mid r) = \sum_{k=1}^{3} q^L_{t,k} \frac{x_{t,k}^{1-r}}{1-r}$$

where $r \in \mathbb{R}$ is the coefficient of relative risk-aversion.

I assume that the prizes are inclusive of the experiment's \$10 show-up fee, and the "endowment" for the lottery, which was used to frame some lottery pairs in the loss domain.

Using the contextual utility normalization (Wilcox 2011) and a logit choice rule with precision λ means that the probability of choosing the Left lottery in pair t is:

$$p(y_t = \text{Left} \mid r, \lambda) = \Lambda \left(\lambda \frac{\sum_{k=1}^{3} q^L_{t,k} \frac{x_{t,k}^{1-r}}{1-r} - \sum_{k=1}^{3} q^R_{t,k} \frac{x_{t,k}^{1-r}}{1-r}}{\frac{\overline{x}_t^{1-r}}{1-r} - \frac{\underline{x}_t^{1-r}}{1-r}} \right)$$

where $\overline{x}_t$ and $\underline{x}_t$ are the largest and smallest prizes in lottery pair t, respectively.

I will use the priors suggested in Bland (2023b) for this model, which are:

$$r \sim N(0.27, 0.36^2)$$
$$\log \lambda \sim N(\log 30, 0.5^2)$$

The prior for r was calibrated to match estimates form Holt and Laury (2002), and the prior for λ was calibrated to achieve some plausible predictions based on the probabilistic choice rule.

9.2 A really slow way to estimate the model

I start out with a *Stan* program that looks like how I would code in *Matlab* in 2010: completely unaware of what makes code fast, but with a good understanding of how a `for` loop works! Here I compute the expected utility difference for each decision separately (in the `for (ii in 1:N)` loop), *and* for good measure also compute the sums in the expected utility formula using a `for` loop. There is nothing mathematically wrong with any of this, but as you will see below, we can speed things up considerably.

```
data {
  int<lower=0> N; // number of decisions

  int Left[N]; // logical =1 if left lottery was chosen

  matrix[N,3] prizes; // the 3 monetary prizes
  matrix[N,2] prizerange; // The minimum and maximum prize


  matrix[N,3] probL; // prob. distribution for Left lottery
  matrix[N,3] probR; // prob. distribution for Right lottery

  vector[2] prior_r; // normal prior on CRRA parameter
  vector[2] prior_lambda; // log-normal prior on logit choice precision
}

parameters {
```

```
  real r;
  real<lower=0> lambda;
}
model {

  // priors
  r ~ normal(prior_r[1],prior_r[2]);
  lambda ~ lognormal(prior_lambda[1],prior_lambda[2]);

  for (ii in 1:N) { // loop over lottery pairs

    real EUleft = 0.0;
    real EUright = 0.0;

    for (kk in 1:3) { // loop over lottery pairs
      // add each component of the sum individually
      EUleft += probL[ii,kk]*pow(prizes[ii,kk],1.0-r)/(1.0-r);
      EUright += probR[ii,kk]*pow(prizes[ii,kk],1.0-r)/(1.0-r);

    }
    // contextual utility normalization
    real normalizedUtility = (EUleft-EUright)/
                             (pow(prizerange[ii,2],1.0-r)/(1.0-r)-pow(prizerange[ii,1],1.0-r)/(1.0-r));

    target += bernoulli_logit_lpmf(Left[ii] | lambda*normalizedUtility);

  }

}
```

9.3 Pre-computing things

Probably the lowest-hanging fruit for speeding up your code is realizing when something can be calculated once, rather than every time *Stan* does an iteration. That is, if you can move something from the `model` or `transformed parameters` blocks into the `transformed data` block, then *Stan* will only have to do it once, instead of thousands of times.

For the EUT model presented above, note the following re-arrangement of the likelihood:

$$
\begin{aligned}
p(y_t = \text{Left} \mid r, \lambda) &= \Lambda\left(\lambda \frac{\sum_{k=1}^{3} q_{t,k}^{L} \frac{x_{t,k}^{1-r}}{1-r} - \sum_{k=1}^{3} q_{t,k}^{R} \frac{x_{t,k}^{1-r}}{1-r}}{\frac{\overline{x}_t^{1-r}}{1-r} - \frac{\underline{x}_t^{1-r}}{1-r}}\right) \\
&= \Lambda\left(\lambda \frac{\sum_{k=1}^{3} \frac{x_{t,k}^{1-r}}{1-r}\left(q_{t,k}^{L} - q_{t,k}^{R}\right)}{\frac{\overline{x}_t^{1-r}}{1-r} - \frac{\underline{x}_t^{1-r}}{1-r}}\right)
\end{aligned}
$$

That is, an EUT participant only cares about the *difference* in probabilities between two lotteries, and since these probabilities are not a function of the model's parameters, we can compute their difference in the `transformed data` block. Here is my modification to the program that does this (the new data variable is called `dprob`):

```
data {
  int<lower=0> N; // number of decisions
```

```
  int Left[N]; // logical =1 if left lottery was chosen

  matrix[N,3] prizes; // the 3 monetary prizes
  matrix[N,2] prizerange; // The minimum and maximum prize


  matrix[N,3] probL; // prob. distribution for Left lottery
  matrix[N,3] probR; // prob. distribution for Right lottery

  vector[2] prior_r; // normal prior on CRRA parameter
  vector[2] prior_lambda; // log-normal prior on logit choice precision
}

transformed data {

  matrix[N,3] dprob = probL-probR;

}

parameters {
  real r;
  real<lower=0> lambda;
}
model {

  // priors
  r ~ normal(prior_r[1],prior_r[2]);
  lambda ~ lognormal(prior_lambda[1],prior_lambda[2]);

  for (ii in 1:N) { // loop over lottery pairs

    real DEU = 0.0;

    for (kk in 1:3) { // loop over lottery pairs
      // add each component of the sum individually
      DEU += dprob[ii,kk]*pow(prizes[ii,kk],1.0-r)/(1.0-r);

    }
    // contextual utility normalization
    real normalizedUtility = DEU/
                             (pow(prizerange[ii,2],1.0-r)/(1.0-r)-pow(prizerange[ii,1],1.0-r)/(1.0-r));

    target += bernoulli_logit_lpmf(Left[ii] | lambda*normalizedUtility);

  }

}
```

9.4 Vectorization

Now we will eliminate that nasty double `for` loop by replacing it with a matrix operation. This will speed things up because *Stan* is very fast at vector and matrix operations. Furthermore, a lot of *Stan*'s in-built functions are vectorized, and so we can even take advantage of this for non-linear functions (see below where

I use `pow(,)`).

The trick here is to note that our expression for expected utility is a sum, and we can write a sum as an inner product. Since we have `N` of these sums, if we can write the terms of the sum as a matrix, we can compute *all* of the expected utilities as a matrix multiplication. First, note that all of the terms in the expression for $Eu(x_t, p_t^L \mid r)$ can be written as:

$$q^L \cdot \frac{x^{1-r}}{1-r}$$

where q^L is an $N \times 3$ matrix representing the probabilities in the L lotteries, x is an $N \times 3$ matrix representing the prizes in these lotteries, and $\cdot$ indicates an element-wise matrix multiplication. Here we are raising x to the power of $1-r$ element-wise. The result of this is an $N \times 3$ matrix, where each row represents a lottery, and each column represents a prize. Now we need to add up the rows. This can be done by multiplying this matrix by a vector of ones:

$$Eu(x, q^L \mid r) = \left(q^L \cdot \frac{x^{1-r}}{1-r}\right) \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

Carrying through our insight that we can pre-compute the difference in probabilities, we can further speed this up by just calculating the expected utility *difference*, which is all we need to compute the logit choice probability:

$$Eu(x, q^L \mid r) - Eu(x, p^R \mid r) = \left((q^L - q^R) \cdot \frac{x^{1-r}}{1-r}\right) \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

Here is how I implement this in the *Stan* program:

```
vector[N] DEU = (
     dprob
     .*pow(prizes,1.0-r)/(1.0-r)
   )*rep_vector(1.0,3);
```

Note the use of `.*` for the element-wise multiplication, and that the `pow(.,.)` function also does an element-wise operation (raising every element of `prizes` to the power of `1-r`).

Finally, we can take advantage of the fact that `bernoulli_logit_lpmf(.|.)` is a *reduction*, so instead of writing a `for` loop to increment the likelihood:

```
for (ii in 1:N) {
  target += bernoulli_logit_lpmf(Left[ii] | lambda * DEU[ii] ./
          (pow(prizerange[ii,2],1.0-r)-pow(prizerange[ii,1],1.0-r))/(1.0-r)
      );
}
```

We can just input the whole vector `DEU` and integer array `Left`, and it will compute the sum of the individual log-likelihoods.

```
target += bernoulli_logit_lpmf(Left | lambda * DEU ./
        (pow(prizerange[,2],1.0-r)-pow(prizerange[,1],1.0-r))/(1.0-r)
    );
```

Here is the whole *Stan* program:

```
data {
  int<lower=0> N; // number of decisions

  int Left[N]; // logical =1 if left lottery was chosen

  matrix[N,3] prizes; // the 3 monetary prizes
  matrix[N,2] prizerange; // The minimum and maximum prize


  matrix[N,3] probL; // prob. distribution for Left lottery
  matrix[N,3] probR; // prob. distribution for Right lottery

  vector[2] prior_r; // normal prior on CRRA parameter
  vector[2] prior_lambda; // log-normal prior on logit choice precision
}

transformed data {

  matrix[N,3] dprob = probL-probR;

}

parameters {
  real r;
  real<lower=0> lambda;
}
model {

  // priors
  r ~ normal(prior_r[1],prior_r[2]);
  lambda ~ lognormal(prior_lambda[1],prior_lambda[2]);

  vector[N] DEU = (
    dprob
    .*pow(prizes,1.0-r)/(1.0-r)

    )*rep_vector(1.0,3);

    target += bernoulli_logit_lpmf(Left | lambda * DEU ./
        (pow(prizerange[,2],1.0-r)-pow(prizerange[,1],1.0-r))/(1.0-r)
    );

}
```

9.5 Within-chain parallelization with `reduce_sum()`

If you have run any *Stan* program at all, you will probably be aware that it automatically gives you *between*-chain parallelization.[37] That is, if your computer has more than one logical processor, and you use the recommended option `options(mc.cores = parallel::detectCores())`, *RStan* will run as many of the required chains as it can in parallel, which is four chains by default. This is great, because things should run about four times faster (if you have at least four logical processors).[38] *Stan* also supports *within*-chain

[37] If you *haven't* run one yet, what are you doing? Go back and try out some of the examples that come before this chapter!

[38] In reality it will be a bit worse than four times faster because your computer needs to devote some time to managing the parallization. Anecdotally for me doing things on my laptop, I know that running chains in parallel on my laptop comes with a

parallelization, which means that within a chain more than one computation can be done at a time, if your computer can support it.

In this section, I will show you how to use *Stan*'s `reduce_sum()` function (documentation here) to parallelize the computation of the log-likelihood. The trick here is realizing that since the log-likelihood is a sum, we can compute subsets of the sum in parallel (i.e. on separate processors), and then add up the results to get the grand total. That is, we can write the log-likelihood as the sum of the individual likelihoods of the 100 decisions made by the participant, then split these up into (say) four chunks of 25 decisions:

$$\begin{aligned}\log p(y \mid \theta) &= \sum_{t=1}^{100} \log p(y_t \mid \theta) \\ &= \underbrace{\sum_{t=1}^{25} \log p(y_t \mid \theta)}_{\text{to processor 1}} + \underbrace{\sum_{t=26}^{50} \log p(y_t \mid \theta)}_{\text{to processor 2}} + \underbrace{\sum_{t=51}^{75} \log p(y_t \mid \theta)}_{\text{to processor 3}} + \underbrace{\sum_{t=76}^{100} \log p(y_t \mid \theta)}_{\text{to processor 4}}\end{aligned}$$

In principle if you have four processors, this should take a a bit more than a quarter of the time that the non-paralellized addition takes, but in practice there is substantial overhead in managing the parallelization. This means that you want to think carefully about which operations you parallelize, and how you chose tuning parameters for the parallelization (more on this later). In fact in the example below, I had to give *Stan* a more computationally intensive problem to showcase the benefits of within-chain parallelization: there really are no noticeable gains to be made if we just want to do the participant-specific estimation (at least on my machine).

In order to use `reduce_sum()`, you will need to write a user-defined function in the `functions` block that computes a chunk of the sum, returning a real number. In my code below, this function is called `sum_likelihood()`. `reduce_sum()` needs this function to have a specific *signature*, meaning that its arguments need to appear in a specific order, which is:

1. `x_slice`, the slice of the data that we will use to compute this chunk of the likelihood
2. `start`, an integer identifying the first term of the data going into this chunk
3. `end`, an integer identifying the last term of the data going into this chunk
4. Any remaining quantities we need to compute the sum

Here is the user-defined function I wrote to compute this chunk:

```
real sum_likelihood(
  int[] Left,
  int start, int end,
  real lambda, real r,
  matrix dprob, matrix prizes, matrix prizerange
) {

  int n = end-start+1;

  vector[n] lambdaDEU = lambda*((dprob[start:end,] .*pow(prizes[start:end,],1.0-r))*rep_vector(1.0,3)
                                ./
                                (pow(prizerange[start:end,2],1.0-r)-pow(prizerange[start:end,1],1.0-r
  ;

  return bernoulli_logit_lpmf(Left | lambdaDEU);

}
```

noticeable start-up time (usually 10-30 seconds) that is not present if I run the chains in series. Therefore I sometimes shut down the between-chain parallelization if each chain only takes a couple of seconds to run. This start-up time is not noticeable if I am running things on my server.

That is, we could use this function *without* any within-chain parallelization to increment the likelihood as follows:

```
target += sum_likelihood(Left,
                         1,N,
                         lambda,r,
                         dprob,prizes,prizerange
                           );
```

But since we *do* want to use within-chain paralleloization, we input this user-defined function into `reduce_sum()` like this:

```
target += reduce_sum(
      sum_likelihood,
      Left,
      grainsize,
      lambda, r,
      dprob, prizes, prizerange
    );
```

Here `grainsize` is a (positive integer) tuning parameter for `reduce_sum()`, and refers to the maximum number of elements in a chunk of the sum to be sent off to a processor. *Stan*'s documentation for choosing `grainsize` recommends choosing `grainsize = 1` unless you spend some time investigating how this parameter affects performance. With this option selected, *Stan* will try to find a good way to split up the sum. For any other `grainsize > 1`, it follows your recommendation.

Here is the entire *Stan* program. I have coded it with an option `doparallel`, which allows for toggling whether or not within-chain parallelization is used. That way when I look for any improvements due to parallelization, I will be running everything from the same *Stan* program.

```
functions {

  real sum_likelihood(
    int[] Left,
    int start, int end,
    real lambda, real r,
    matrix dprob, matrix prizes, matrix prizerange
  ) {

    int n = end-start+1;

    vector[n] lambdaDEU = lambda*((dprob[start:end,] .*pow(prizes[start:end,],1.0-r))*rep_vector(1.0,3)
                                  ./
                                  (pow(prizerange[start:end,2],1.0-r)-pow(prizerange[start:end,1],1.0-r
    ;




    return bernoulli_logit_lpmf(Left | lambdaDEU);

  }

}
data {
```

```
  int<lower=0> N; // number of decisions

  int Left[N]; // logical =1 if left lottery was chosen

  matrix[N,3] prizes; // the 3 monetary prizes
  matrix[N,2] prizerange; // The minimum and maximum prize


  matrix[N,3] probL; // prob. distribution for Left lottery
  matrix[N,3] probR; // prob. distribution for Right lottery

  vector[2] prior_r; // normal prior on CRRA parameter
  vector[2] prior_lambda; // log-normal prior on logit choice precision

  int grainsize;

  int doparallel; // if ==1 then do within-chain parallelization
}

transformed data {

  matrix[N,3] dprob = probL-probR;

}

parameters {
  real r;
  real<lower=0> lambda;
}
model {

  // priors
  r ~ normal(prior_r[1],prior_r[2]);
  lambda ~ lognormal(prior_lambda[1],prior_lambda[2]);


  if (doparallel==1) {

    target += reduce_sum(
      sum_likelihood,
      Left,
      grainsize,
      lambda, r,
      dprob, prizes, prizerange
    );
  } else {

    target += sum_likelihood(Left,
                          1,N,
                          lambda,r,
                          dprob,prizes,prizerange
                            );
  }
```

```
}
```

9.6 Evaluating the implementations

As explained in more detail below, there are not any gains to be made from parallelization of the participant-specific estimation. I therefore start with evaluating the the performance of the other models, then turn to evaluating parallelization from a slightly different angle.

9.6.1 Pre-computing and vectorization

Here I estimate the three un-parallelized models for each participant in Harrison and Swarthout (2023), running *RStan*'s default options except for:

- `chains = 1`, i.e. running only one chain, and
- `refresh = 1000`, which only displays progress every 1,000 iterations

The computation times are summarized in Figure 34, which shows the empirical cumulative distributions of these times. The vectorized model runs almost four times faster than the original "slow" model.

```
BenchmarkSummary<-"Code/SpeedyCode/BenchmarkSummarySeries.rds" |>
  readRDS()

d<-BenchmarkSummary |>
  group_by(id,model) |>
  summarize(
    time = mean(time)
  )

(
  ggplot(d ,aes(x=time,color=model))
  +stat_ecdf()
  +theme_bw()
  +xlab("computation time (s)")
  +ylab("empirical cumulative density")
  +coord_cartesian(xlim = c(0,max(d$time)))
  +scale_color_discrete(breaks = c("slow","precompute","vectorized","parallel"))
)
```

9.6.2 Within-chain parallelization

To evaluate the performance improvements due to within-chain paralleization, I give *Stan* a more difficult problem to solve. This is because there is a substantial fixed time cost associated with managing more than one core, and so there might not be much to see if we just looked at the participant-specific estimations, which only had 100 observations each. Instead, we can increase the number of things that *Stan* needs to add up to compute the likelihood by estimating a representative-agent model pooling all of the data. This means that instead of having to add up 100 things to compute the likelihood, *Stan* will need to add up 32,700 things (i.e. 327 participants each making 100 decisions).

In order to showcase the benefits of parallelization, I estimate the model with just one chain. This means that I can devote all eight cores of my server to within-chain parallelization. I estimate the model for five different values of the tuning parameter `grainsize`, including `grainsize = 1`, which allows *Stan*'s scheduler to determine how the sum is split up. These were chosen to roughly split the summation up into 8ths, 16ths, 32nds and so on on my 8-core server. I also estimate the model without parallelization (setting the data input `doparallel` to zero). The computation times are summarized in Table 17. For the values of `grainsize` explored, the computation time is about ten times faster for the parallelized process.

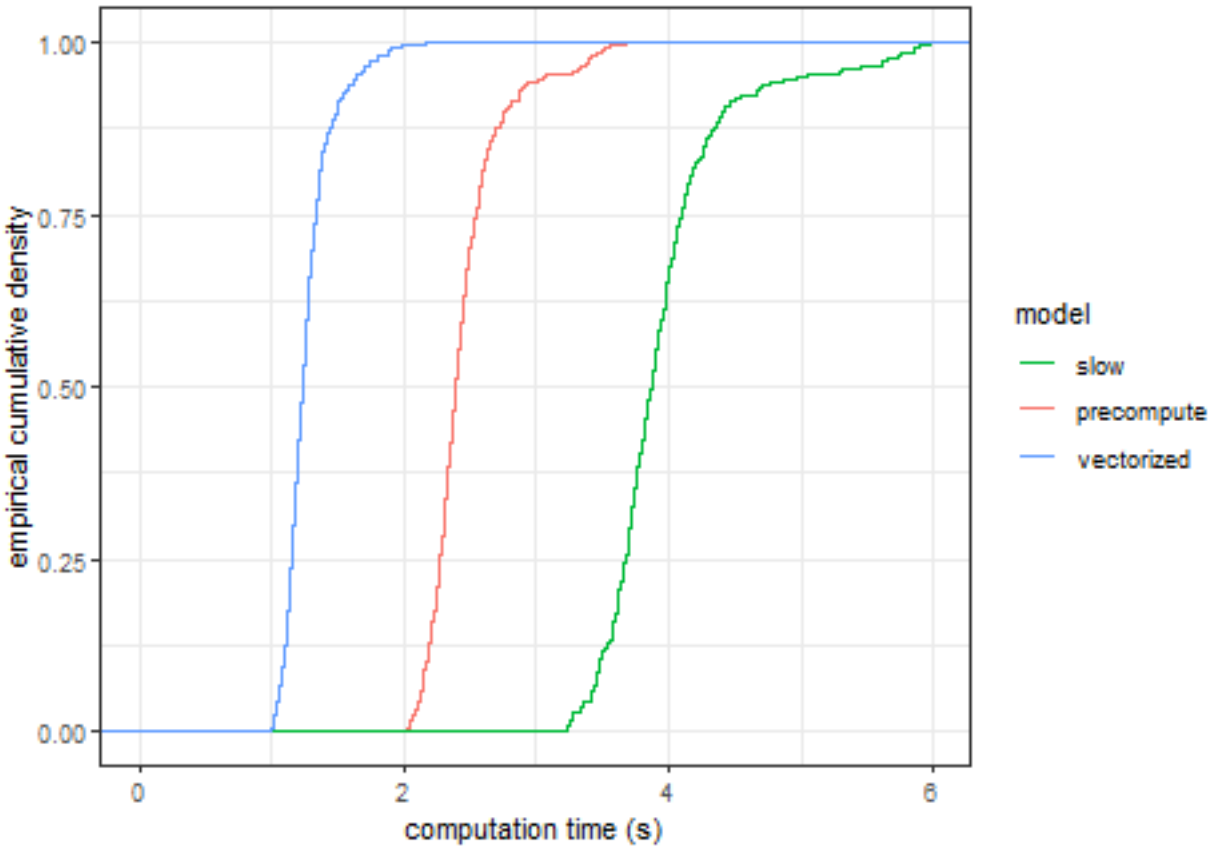


Figure 34: Computation time for individual-level estimation of models, excluding the paralleized program.

Table 17: Computation time for parallelized model. ‘grainsize = NA‘ indicates the un-parallelized process. ‘grainsize = 1‘ allows *Stan*’s scheduler to choose how to split up the summation.

grainsize	Time to estimate model (s)
1	50.9
511	50.1
1022	50.4
2044	56.3
4088	86.6
NA	539.5

```
Parallel<-"Code/SpeedyCode/BenchmarkSummaryParallel.rds" |>
  readRDS() |>
  group_by(grainsize) |>
  summarize(
    `Time to estimate model (s)` = mean(time)
  )

Parallel |>
  round(1) |>
  kbl(caption = "Computation time for parallelized model. `grainsize = NA` indicates the un-parallelize
  kable_classic(full_width=FALSE)
```

While we should not be particularly interested in the estimates from this representative agent model, it is important to note that we can probably benefit from these improvements in other, more interesting models, too. For example if we were estimating a hierarchical model using these data, we would also need to add up 32,700 things to compute the likelihood. If we can expect similar improvements in computational time for this model, then this could turn a 3-day slog into an over-nighter!

9.7 *R* code to estimate models

9.7.1 Slow, pre-computed, and vectorized models

```
library(tidyverse)
library(haven)
library(rstan)
  rstan_options(auto_write = TRUE)
  options(mc.cores = parallel::detectCores())
  rstan_options(threads_per_chain = 8)


numreps<-1

# Note: The show-up fee for the undergrads was $10 and $40 for the MBA students

BenchmarkSummary<-tibble()

D<-"Data/HS2023.dta" |>
  read_dta() |>
  select(id,choice,prize1L:prob4R,endowment) |>
  # add in the endowment
  mutate(
```

```
    show_up_fee = ifelse(mba,40,10),
    Left = 1-choice,
    prize1L = prize1L+endowment+show_up_fee,
    prize2L = prize2L+endowment+show_up_fee,
    prize3L = prize3L+endowment+show_up_fee,
    prize4L = prize4L+endowment+show_up_fee,
    prize1R = prize1R+endowment+show_up_fee,
    prize2R = prize2R+endowment+show_up_fee,
    prize3R = prize3R+endowment+show_up_fee,
    prize4R = prize4R+endowment+show_up_fee
  ) |>
  # there were never four possible prizes
  select(-contains("4")) |>
  rowwise() |>
  mutate(
    prizerangeLow = min(c(prize1L,prize2R,prize3L)),
    prizerangeHigh = max(c(prize1L,prize2R,prize3L))
  ) |>
  filter(!is.na(Left))


ModelList<-list(
  slow = "Code/SpeedyCode/EUT_slow.stan" |> stan_model(),
  precompute = "Code/SpeedyCode/EUT_precompute.stan" |> stan_model(),
  vectorized = "Code/SpeedyCode/EUT_vectorized.stan" |> stan_model()
)


for (rr in 1:numreps) {
  for (ii  in unique(D$id)) {

    print(paste("rep",rr,"of",numreps,"id",ii,"of",length(D$id |> unique())))

    d<- D|> filter(id==ii)

    dStan<-list(
      N = dim(d)[1],
      Left = d$Left,

      prizes = cbind(d$prize1L,d$prize2L,d$prize3L),
      prizerange=cbind(d$prizerangeLow,d$prizerangeHigh),

      probL = cbind(d$prob1L,d$prob2L,d$prob3L),
      probR = cbind(d$prob1R,d$prob2R,d$prob3R),

      prior_r = c(0.27,0.36),
      prior_lambda = c(log(30),0.5)
    )

    for (mm in names(ModelList)) {

      Fit<-ModelList[[mm]] |>
        sampling(seed=42,chains=1,data=dStan,refresh=1000)
```

```
      BenchmarkSummary<- BenchmarkSummary |>
        rbind(
          tibble(id=ii,
                 model = mm,
                 rep = rr,
                 grainsize=NA,
                 time = get_elapsed_time(Fit) |> sum()
                 )
        )



    }

    BenchmarkSummary |>
      saveRDS("Code/SpeedyCode/BenchmarkSummarySeries.rds")

  }


}
```

9.7.2 Parallelized model

```
library(tidyverse)
library(haven)
library(rstan)
  rstan_options(auto_write = TRUE)
  options(mc.cores = parallel::detectCores())
  rstan_options(threads_per_chain = 8)


numreps<-10



# Note: The show-up fee for the undergrads was $10 and $40 for the MBA students

BenchmarkSummary<-tibble()

D<-"Data/HS2023.dta" |>
  read_dta() |>
  select(id,choice,prize1L:prob4R,endowment) |>
  # add in the endowment
  mutate(
    show_up_fee = ifelse(mba,40,10),
    Left = 1-choice,
    prize1L = prize1L+endowment+show_up_fee,
    prize2L = prize2L+endowment+show_up_fee,
    prize3L = prize3L+endowment+show_up_fee,
    prize4L = prize4L+endowment+show_up_fee,
    prize1R = prize1R+endowment+show_up_fee,
    prize2R = prize2R+endowment+show_up_fee,
```

```
    prize3R = prize3R+endowment+show_up_fee,
    prize4R = prize4R+endowment+show_up_fee
  ) |>
  # there were never four possible prizes
  select(-contains("4")) |>
  rowwise() |>
  mutate(
    prizerangeLow = min(c(prize1L,prize2R,prize3L)),
    prizerangeHigh = max(c(prize1L,prize2R,prize3L))
  ) |>
  filter(!is.na(Left))

GRAINSIZE<-c(1,4088,2044,1022,511)

model<-"Code/SpeedyCode/EUT_parallel.stan" |>
  stan_model()



for (rr in 1:numreps) {

    print(paste("rep",rr,"of",numreps))

    dStan<-list(
      N = dim(D)[1],
      Left = D$Left,

      prizes = cbind(D$prize1L,D$prize2L,D$prize3L),
      prizerange=cbind(D$prizerangeLow,D$prizerangeHigh),

      probL = cbind(D$prob1L,D$prob2L,D$prob3L),
      probR = cbind(D$prob1R,D$prob2R,D$prob3R),

      prior_r = c(0.27,0.36),
      prior_lambda = c(log(30),0.5),

      grainsize=1,
      doparallel = 0
    )

    Fit<-model |>
      sampling(seed=42,chains=1,data=dStan,refresh=1000)
    BenchmarkSummary<- BenchmarkSummary |>
      rbind(
        tibble(id=ii,
               model = "parallel",
               rep = rr,
               grainsize=NA,
               time = get_elapsed_time(Fit) |> sum()
        )
      )

    dStan$doparallel<-1
```

```
    for (gg in GRAINSIZE) {
      dStan$grainsize<-gg
      Fit<-model |>
        sampling(seed=42,chains=1,data=dStan,refresh=1000)

      BenchmarkSummary<- BenchmarkSummary |>
        rbind(
          tibble(id=ii,
                 model = "parallel",
                 rep = rr,
                 grainsize=gg,
                 time = get_elapsed_time(Fit) |> sum()
          )
        )

    }



    BenchmarkSummary |>
      saveRDS("Code/SpeedyCode/BenchmarkSummaryParallel.rds")



}
```

Applications

10 Application: Experience-Weighted Attraction

In this application, I will run you through my take on a Bayesian version of Table 3 of Cason, Friedman, and Hopkins (2010), which reports estimates of parameters in the Experience-Weighted Attraction model (Camerer and Ho 1999). In this experiment, participants played 80 rounds of a modification of Rock Paper Scissors called "Rock Paper Scissors Dumb", which is shown in Table 18. Here the added strategy "Dumb" is never a best response to a pure strategy, but is played with probability $\frac{1}{2}$ in mixed strategy Nash equilibrium. While the games share the same mixed strategy Nash equilibrium prediction, they diverge in their predictions assuming fictitious play-like learning: in the Stable game play will converge to the Nash equilibrium, whereas in the Unstable game there will be cycles. Participants played in groups of 12, and were randomly re-matched each period with another participant in their group each round.

```
dStan<-readRDS("Data/CFH2010TASP/CFH2010TASP_dStan.rds")

TAB<-list(
  Unstable = dStan$payoffs[1,,],
  Stable   = dStan$payoffs[2,,]
)

for (gg in c("Stable","Unstable")) {
  colnames(TAB[[gg]])<-c("rock","paper","scissors","dumb")
  rownames(TAB[[gg]])<-c("Rock","Paper","Scissors","Dumb")
}
```

Table 18: The "Unstable" and "Stable" Rock-Paper-Scissors-Dumb games. Just the payoffs of the row player are shown. The payoffs of the column player are the transpose of the row player.

	Unstable treatment				Stable treatment			
	rock	paper	scissors	dumb	rock	paper	scissors	dumb
Rock	90	0	120	20	60	0	150	20
Paper	120	90	0	20	150	60	0	20
Scissors	0	120	90	20	0	150	60	20
Dumb	90	90	90	0	90	90	90	0

```
tab<-cbind(TAB[[1]],TAB[[2]])


tab |> kbl(caption = 'The "Unstable" and "Stable" Rock-Paper-Scissors-Dumb games. Just the payoffs of th
```

In addition to the Stable and Unstable games, participants were also assigned either to a high payoffs or low payoffs treatment, where the "Experimental Francs" in the above table were multiplied by 0.05 and 0.02 US dollars respectively.

In their Table 3, Cason, Friedman, and Hopkins (2010) estimate several permutations of the Experience-Weighted Attraction (EWA) model (Camerer and Ho 1999). This model nests some important special cases of learning models, including Stochastic Fictitious Play (SFP), which assumes that a player's beliefs in period t are equal to the (potentially weighted) average of empirical choice frequencies observed up to that point. As noted by Wilcox (2006) however, the EWA model suffers from *heterogeneity bias*: if participants are heterogeneous in their parameters, then pooled estimates of these parameters will be biased. In particular, the pooled estimator favors reinforcement learning over belief learning. Fortunately, one remedy to this is relatively straightforward and lends itself to Bayesian estimation: model the heterogeneity! As such, estimating a hierarchical model makes a lot of sense here.

10.1 The model at the individual level

The EWA model (Camerer and Ho 1999) is a 4-parameter model that describes how players form expectations about payoffs based on the history of play. These expectations about payoffs are called "attractions", which evolve according to the following equations:

$$A_{i,t}^{j} = \frac{\phi N_{i,t-1} A_{i,t-1}^{j} + [\delta + (1-\delta) I_{j,t-1}]\, \pi_{j,t-1}}{N_{i,t}}$$

$$N_{i,t} = \rho N_{i,t-1} + 1$$

where:

- $\pi_{j,t-1}$ is the payoff the payer would have received had they taken action j in the previous period. That is, it is a function of the action that their opponent took in the previous period.
- $I_{j,t-1} = 1$ iff action j was played in the previous period, zero otherwise.

This part of the model has three parameters, all of which are constrained to the unit interval:

- ϕ and ρ are "recency" parameters. When both are equal to one, the participant takes a simple average of the payoffs from all previous periods (this is the fictitious play model). If they are both equal and less than one, the participant takes a weighted average of payoffs from all of the previous rounds (this is the weighted fictitious play model).

- δ is an "imagination factor". When $\delta = 1$ the player can "imagine" what would have happened if they played action j, even if they had in fact not played action j. On the other hand, when $\delta = 0$, they only consider the payoff that they get from the action they took in the previous period.

Players then probabilistically best respond to these attractions. Specifically, the probability of player i taking action j in period t follows a logit choice rule:

$$p_{i,t}^j = \frac{\exp(\lambda A_{i,t}^j)}{\sum_{k=1}^n \exp(\lambda A_{i,t}^k)}$$

where $\lambda > 0$ is the choice precision parameter.

Cason, Friedman, and Hopkins (2010) set their initial conditions to $A_{i,0}^j = 1$ and $N_{i,0} = 0$, noting that their results are "robust to alternative initial attraction and experience weights" (see their footnote 8). I will stick with this assumption for my analysis.

10.2 Some computational and coding issues

While we are mostly very lucky in experiments to collect our data in a very clean format that usually requires very little tidying, sometimes we get something special. In this case, there was a tornado during one of the sessions, and so these participants only got to play the game for 70 periods, instead of the planned 80.[39] This is why you will see me pass a binary variable to *Stan* called `UseData`, which tells it whether or not to let an observation contribute to the likelihood. As you will see in other examples where I use cross-validation, having a variable like `UseData` is generally helpful anyway, as it allows us to easily turn on or off whether or not we are using some data, without modifying our *Stan* program. It is also useful if we want to get draws from the prior: just set all of its elements to zero!

Perhaps more importantly for your own applications, I want you to note that there is *a lot* we can do here to speed up computational time by pre-processing some of the variables in the model. Specifically, the variables $I_{j,t}$ and $\pi_{j,t}$ do not depend on the model's parameters, and so we can pre-calculate these before passing them to *Stan*, which means we only have to evaluate these once.[40] This will greatly reduce the computational time compared to doing these in the `model` or `transformed parameters` block, which are evaluated thousands of times.

Also, the recursive equation specifying $A_{i,t}^j$ means that we will not be able to avoid looping over the t dimension of this problem: we cannot compute $A_{i,t}^j$ without knowing $A_{i,t-1}^j$. Since there were 80 periods per participant, this could be a considerable time suck for *Stan*. That being said, with some careful thought to vectorization, we *will* be able to avoid looping over the i and j dimensions. This will greatly speed things up compared to a triple `for` loop. I have left my inefficient code looping over i in the *Stan* file for the representative agent model, commented out, to show you what the double `for` loop would look like. Spotting these vectorized representations of the operations we need to do will usually come in handy, and while it might not save too much time for the representative agent model, the time savings will be carried through when we are estimating the hierarchical model. As you will see in my code for the representative agent model, I don't give a hoot about eliminating triple `for` loops in the `transformed data` block. This block will only run once, so the time savings will be negligible. On the other hand, the `transformed parameters`, `model`, and to a lesser extent `generated quantities`[41] blocks will have to run thousands of times to get your posterior simulation, so for these blocks it is usually worth vectorizing things.

Finally, while datasets from economic experiments are not usually that big, our posterior simulations from hierarchical models can get large. This is because the augmented data will have several individual-level

[39] If you're playing along at home, these participants are coded as `session = 4`.

[40] Another alternative would be to calculate these in the `transformed data` block. For some reason *Stan* was crashing on me when I did this.

[41] *Stan* does not have to evaluate derivatives in the `generated quantities` block, so things should compute faster here anyway.

parameters per participant.[42] While I don't run into any RAM constraints on my laptop in running this, I *do* like to back all of my work up in the cloud somewhere, so you might want to think about how much you save and where it goes. There is nothing in this chapter that cannot be re-run in a day if you have to, so saving just the code to get you the estimates in the cloud may be an appropriate choice if your Dropbox is getting full. This is why I use the `pars = ..., include = FALSE` options when calling `sampling` for the hierarchical model.

10.3 Representative agent models

Let's start with estimating some representative agent models. I do this not because we should take them as seriously as the hierarchical model, but because they are a useful stepping stone for getting us to the hierarchical model. We can also compare these to the hierarchical model to see how much heterogeneity bias we might get. Additionally, as Cason, Friedman, and Hopkins (2010) also estimate these representative agent models, I will have an opportunity to verify that my code is working correctly before going to the more complicated model.

10.3.1 Prior calibration

To begin with, let's think about priors. We have four parameters: $\phi \in (0,1)$, $\rho \in (0,1)$, $\delta \in (0,1)$, and $\lambda \in (0,\infty)$. Since the first three are all constrained to the unit interval, a good starting place is to use the uniform distribution. In my code, I allow this to be a Beta distribution, which generalizes the uniform, but I will stick to the uniform to keep things simple.

For λ, we want to constrain it to positive real numbers, so a log-normal distribution is appropriate:

$$\log \lambda \sim N(m_\lambda, s_\lambda^2)$$

The difficulty in choosing λ is choosing an appropriate scale, which since we are dealing with a *log*-normal distribution, is as much about its median as it is about its standard deviation. Small λ implies choices are not very precise, so participants will not be very sensitive to the attractions $A_{i,t}^j$. On the other hand if λ is large, choices will be precise, and small changes in $A_{i,t}^j$ will imply large changes in choice probabilities. But what exactly are "small" and "large" values of λ? I find it helps here to plot some predicted choice probabilities over a relevant scale of payoffs. Note that in Table 18, the smallest payoff is zero points, and the largest payoff is 150 points. Therefore payoff differences must be in this range. Converting this to dollars by using the average of the two exchange rates, we therefore have a range of $[0, 150 \times 0.035] = [0, 5.25]$. We can use the logit Quantal Response Equilibrium (QRE) predictions to further narrow down this scale. Fortunately in this game, computing the Quantal Response Equilibrium is really easy, because Cason, Friedman, and Hopkins (2010) derive the results for us, and it turns out that we can characterize the Quantal Response Equilibrium as $\hat{p} = \big(m, m, m, k\big)$, where $k \in [0.25, 0.5]$ and $m = (1-k)/3$. Because there is a unique λ for each QRE, we can back it out once we know the equilibrium probabilities:

```
pD<-seq(0.25,0.5-1e-6,length=100)
ExRate<-0.035
Uunstable<-dStan$payoffs[1,,]
Ustable<-dStan$payoffs[2,,]

QRE<-tibble()
for (pp in pD) {
  p<-c(rep((1-pp)/3,3),pp)
  U<-Uunstable%*%p*ExRate
  lambda<-(log(p[1])-log(p[4]))/(U[1]-U[4])
  QRE<-rbind(QRE,
             tibble(game="Unstable",pD = pp,lambda = lambda)
```

[42] The hierarchical model using all of the data produced a *Stan* fit object that took up about 140MB, and this was after I told it to drop the normalized augmented data. This is certainly not huge by any modern hard drive or RAM requirements, but it is 7% of what Dropbox gives you for free (at the time of writing).

```
            )
  U<-Ustable%*%p*ExRate
  lambda<-(log(p[1])-log(p[4]))/(U[1]-U[4])
  QRE<-rbind(QRE,
             tibble(game="Stable",pD = pp,lambda = lambda)
             )
}

(
  ggplot(QRE,aes(x=lambda,y=pD,linetype = game))
  +geom_path()
  +scale_x_continuous(trans="log10")
  +theme_bw()
  +ylab("Probability of playing 'Dumb'")
  +xlab(expression(lambda))
)
```

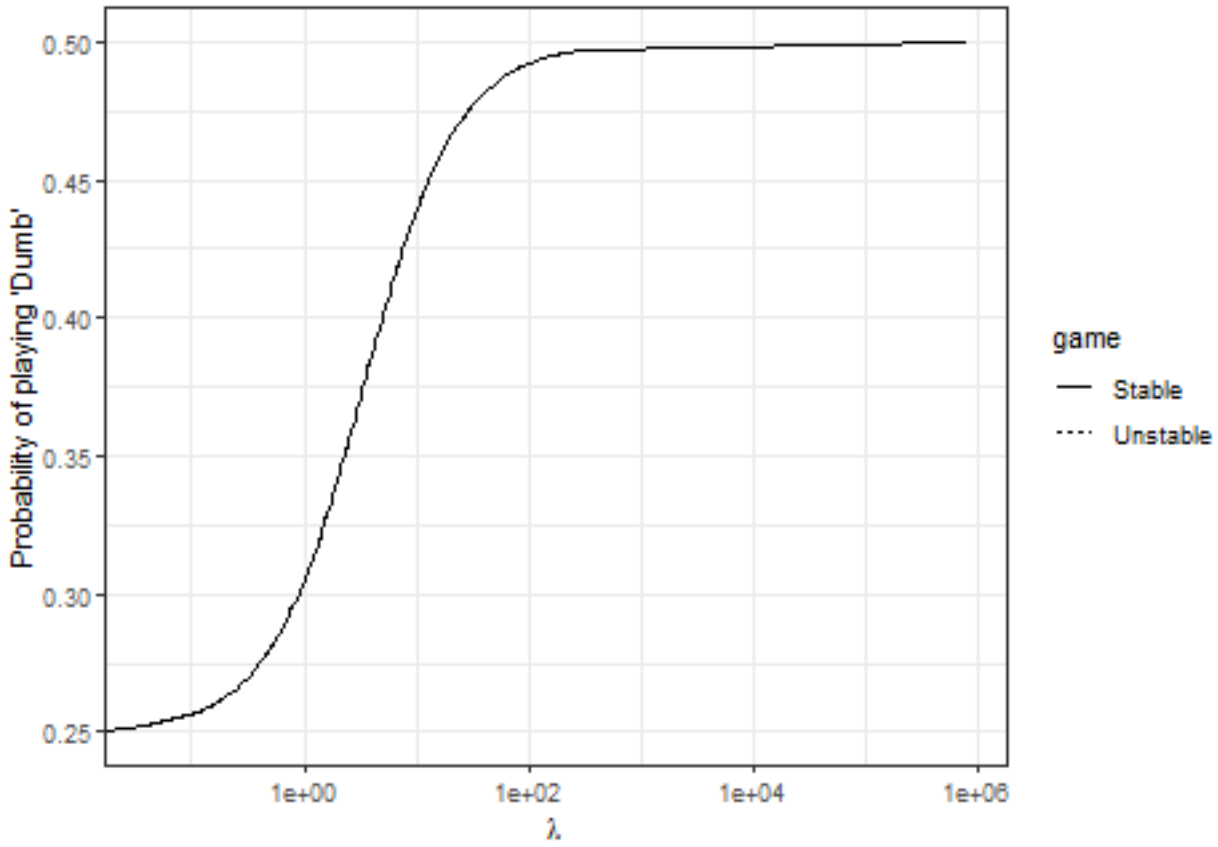


Figure 35: Quantal Response Equilibrium probabilities of choosing 'Dumb'

Some things to note here:

1. The Quantal Response Equilibrium probability of playing Dumb is the same for both games (this is derived in Cason, Friedman, and Hopkins (2010)). We can see this in the plot because both curves lie on top of each other.
2. All of the interesting stuff seems to be happening between $\lambda = 0.1$ and $\lambda = 100$.

Let's target our prior so that we place 95% prior probability mass in this "interesting" range. This sets up the following system of equations:

$$\begin{aligned} -1.96 s_\lambda &= \log(0.1) - m_\lambda \\ 1.96 s_\lambda &= \log(100) - m_\lambda \end{aligned}$$

$$\begin{aligned} m_\lambda &= \frac{1}{2}\left(\log(100) + \log(0.1)\right) \approx 1.15 \\ s_\lambda &= \frac{\log(100) - \log(0.1)}{2 \times 1.96} \approx 1.76 \end{aligned}$$

And so our prior for λ is:

$$\log \lambda \sim N(1.15, 1.76^2)$$

10.3.2 The *Stan* model

Here is the code I came up with to implement the model in *Stan*.

```
data {
  int nPeriods;
  int nParticipants;
  int nGames;
  int nActions;

  // action chosen. 1=Rock, 2=Paper, 3=Scissors, 4=Dumb
  int direction[nPeriods,nParticipants];

  // =1 if low payoffs used, =2 if high payoffs used
  int HiPay[nPeriods,nParticipants];

  // first element: low payoffs exchange rate
  // second element: high payoffs exchange rate
  real payoff_multipliers[2];

  // =1 if the observation should contribute to the likelihood
  int UseData[nPeriods,nParticipants];


  // Pre-calculated components of the EWA model
  real PI[nPeriods,nParticipants,nActions];
  real I[nPeriods,nParticipants,nActions];

  // priors
  real prior_phi[2]; // Beta
  real prior_delta[2]; // Beta
  real prior_rho[2]; // Beta
  real prior_lambda[2]; // log-normal

  real max_payoff;
}

transformed data {
```

```
  matrix[nParticipants,nActions] DIRECTION[nPeriods];

  for (tt in 1:nPeriods) {
    DIRECTION[tt] = rep_matrix(0.0,nParticipants,nActions);
    for (ii in 1:nParticipants) {
      for (aa in 1:nActions) {
        if (direction[tt,ii]==aa) {
          DIRECTION[tt][ii,aa] = 1.0;
        }
      }
    }
  }
}

parameters {

  real<lower=0,upper=1> phi; // recency
  real<lower=0,upper=1> rho; // recency
  real<lower=0,upper=1> delta; // imagination
  real<lower=0> lambda; // precision

}

model {

  // initial conditions. See CFH201 footnote 8
  matrix[nParticipants,nActions] A = rep_matrix(1.0,nParticipants,nActions);
  vector[nParticipants] N = rep_vector(0.0,nParticipants);



  for (tt in 1:nPeriods) {

    // increment the likelihood
    matrix[nParticipants,nActions] lA = lambda*(to_vector(payoff_multipliers[HiPay[tt,]])*rep_row_vecto
    vector[nParticipants] lsum_lA = log(exp(lA)*rep_vector(1.0,nActions));

    /*
    Note here the for loop that would do the same thing as the uncommented line below it
    I construct DIRECTION in the transformed data block
    */

    //for (ii in 1:nParticipants) {
    //  target+= UseData[tt,ii]*(lA[tt,direction[tt,ii]]-lsum_lA[ii]);
    //}

    target+=to_vector(UseData[tt,]) .* (((lA-lsum_lA*rep_row_vector(1.0,nActions)) .* DIRECTION[tt])*re



    // Update A and N
```

```
    matrix[nParticipants,nActions] I_t = to_matrix(I[tt,,]);
    matrix[nParticipants,nActions] pi_t = to_matrix(PI[tt,,]);

    A = (phi*(N*rep_row_vector(1.0,nActions)).*A+(delta+(1.0-delta)*I_t).*pi_t)
          ./ (rho * N * rep_row_vector(1.0,nActions)+1.0);

    N = rho*N+1.0;

  }

  // priors

  phi ~ beta(prior_phi[1],prior_phi[2]);
  rho ~ beta(prior_rho[1],prior_rho[2]);
  delta ~ beta(prior_delta[1],prior_delta[2]);
  lambda ~ lognormal(prior_lambda[1],prior_lambda[2]);

}
```

One thing to note from this code is that I have subtracted the variable `max_payoff` from all of the attractions before exponentiating them to compute choice probabilities. Mathematically this does absolutely nothing, because:

$$\frac{\exp(x-a)}{\exp(x-a)+\exp(y-a)} = \frac{\exp(x)}{\exp(y)+\exp(y)}$$

does not depend on a. However this doesn't do nothing for *computing* the choice probabilities. In particular, if x is large, then $\exp(x)$ is *really* large, and our poor computer may have some trouble handling it. The solution I use is to subtract a large constant (i.e. the maximum payoff in the game) from all attractions. This way we are always exponentiating numbers that are less than zero. Since this is the functional form we are using for choice probabilities, we are taking logs of them when we increment the likelihood. If (in this simplified example) y is very large and positive, then `exp(y)` might be "rounded" to `Inf`, in which case the fraction becomes indistinguishable from zero, and the log-likelihood is `-Inf`. Keeping the numbers that we are exponentiating in a `softmax` small will avoid this problem. In fact, I suspect that *Stan*'s inbuilt `softmax` function will do this for you, but unfortunately in this application it did not lend itself to my vectorization of the problem.

10.3.3 Results

Here I replicate the first panel Table 3 of Cason, Friedman, and Hopkins (2010), which estimates a pooled model for each of the four treatments. I also include a fifth column pooling all of the data.

```
rounding<-3
RAEstimatesList<-dir(path="Code/CFH2010TASP",pattern="Estimates_RA_")
TABLE<-c()
names<-c()
for (ee in RAEstimatesList) {
  estimates<-readRDS(paste0("Code/CFH2010TASP/",ee)) |> round(rounding)
  msd<-paste0(estimates[,"mean"]," (",estimates[,"sd"],")")
  TABLE<-cbind(TABLE,msd)
  names<-c(names,ee |> str_remove("Estimates_RA_") |> str_remove(".rds"))
}
colnames(TABLE)<-names
rownames(TABLE)<-c("$\\psi$","$\\rho$","$\\delta$","$\\lambda$","lp__")
TABLE |> kbl(caption="Posterior moments (means with standard deviations in parentheses) of the represen
```

Table 19: Posterior moments (means with standard deviations in parentheses) of the representative agent models.

	Pooled	StableHi	StableLow	UnstableHi	UnstableLow
ψ	0.909 (0.004)	0.934 (0.006)	0.91 (0.007)	0.882 (0.011)	0.888 (0.009)
ρ	0.564 (0.063)	0.437 (0.184)	0.59 (0.106)	0.404 (0.157)	0.474 (0.166)
δ	0.007 (0.006)	0.018 (0.016)	0.012 (0.011)	0.032 (0.023)	0.021 (0.018)
λ	0.355 (0.046)	0.257 (0.082)	0.42 (0.088)	0.295 (0.074)	0.365 (0.102)
lp___	-12448.586 (1.498)	-3070.514 (1.575)	-3118.543 (1.622)	-3152.688 (1.551)	-3115.559 (1.617)

Comparing this to the left panel in Table 3 of Cason, Friedman, and Hopkins (2010), these estimates are very similar. This is especially the case for ϕ, whose posterior means match their MLE counterparts to at least two decimal places. Since ρ is estimated with less precision than ϕ in both tables, it should be unsurprising that there is less agreement between these estimates, however everything is accurate to one standard deviation/error. The biggest discrepancy is for choice precision λ, which I estimate to be much larger than do Cason, Friedman, and Hopkins (2010). I suspect that this is due to my model using US dollars as the unit for utility, whereas it looks like Cason, Friedman, and Hopkins (2010) used experimental francs. Multiplying my estimates of λ by the exchange rate at least achieves estimates in the same order of magnitude as their Table 3.

10.4 Hierarchical model

So now that we have a representative agent model that works, we can move on to extending it to a hierarchical model. To modify the *Stan* program, we will need to:

1. Convert the scalar individual-level parameters ϕ, ρ, δ, and λ into vectors of length `nParticipants`.
2. Specify the population level parameters and choose some appropriate priors for them
3. Link the distribution of these individual-level parameters to the new population-level parameters.

To speed up computation, I will also use Cholesky factorization of the covariance matrix, which I discuss in the hierarchical models chapter of this book.

The hierarchical specification I will use for this model is:

$$\begin{pmatrix} \Phi^{-1}(\phi_i) \\ \Phi^{-1}(\rho_i) \\ \Phi^{-1}(\delta_i) \\ \log(\lambda_i) \end{pmatrix} = \theta_i \sim N(\mu, \text{diag\_matrix}(\tau)\Omega\text{diag\_matrix}(\tau))$$

where μ is a vector of transformed population means, τ is a vector of standard deviations, and Ω is a correlation matrix. The inverse normal cdf transform $\Phi^{-1}(\cdot)$ ensures that parameters ϕ_i, ρ_i, and δ_i are between zero and one, and the log transform ensures that λ_i is positive.

Apart from the differences due to my Bayesian implementation and the MLE implementation in Cason, Friedman, and Hopkins (2010), there are few important differences in this specification:

1. I permit correlation between the individual-level parameters through the correlation matrix Ω. In Table 3 of Cason, Friedman, and Hopkins (2010) these parameters are assumed to be uncorrelated
2. I assume that all the individual-level parameters are heterogeneous. Note that in the middle panel of Table 3 of Cason, Friedman, and Hopkins (2010), there is not a measure of spread for parameter δ_i, whereas there is one for the other three parameters. I take this to mean that δ_i is assumed to be constant across participants in their models.

10.4.1 Prior calibration

Here I use the normal-half Cauchy-LKJ type of prior for the population-level parameters μ τ and Ω discussed in the Hierarchical Models chapter of this book. For μ, I center the priors on each parameter's prior median from the representative agent models. For the probit-normal parameters ϕ_i, ρ_i, and δ_i, note that the corresponding element for μ pins down their prior medians. Setting a *standard* normal prior for, say, $\mu_\phi \sim N(0,1)$ would therefore mean that our prior belief was that the median of ϕ_i was uniformly distributed. As the median being very close to 0 or 1 for these parameters seems unlikely, I choose a slightly smaller prior standard deviation for these:

$$\mu_\phi,\ \mu_\rho,\ \mu_\delta\ \sim N(0, 0.25^2)$$

Similarly for λ_i:

$$\mu_\lambda \sim N(1.15, 0.75^2)$$

centers the median of this prior over the prior we used for the representative agents model, but reduces some of its variance: our prior uncertainty about a parameter's median should be lower than our prior uncertainty about the parameter itself.

For τ, I select:

$$\tau_\phi,\ \tau_\rho,\ \tau_\delta,\ \tau_\lambda \sim \text{Cauchy}^+(0, 0.05)$$

This makes the 95th percentile of the prior distribution for each of these about 0.6. For perspective, for the log-normal distribution of λ_i, $\tau_\lambda = 0.6$ implies that λ_i will be within about 31% and 320% of its population median with probability 95%. For the other, probit-normally distributed parameters, this ensures that their distributions are single-peaked.

Finally, for the prior for Ω I choose:

$$\Omega \sim \text{LKJ}(5)$$

10.4.2 The *Stan* model

Here is how I modified my representative agent model to a hierarchical model. Hopefully you can see that a lot of it is exactly the same as the representative agent model. The hardest part for me came from converting all of the individual-level parameters from `reals` to `vectors`, as this meant I had to change some of the lines that calculated the attractions so that matrix sizes agreed again, and so on.[43]

```
data {
  int nPeriods;
  int nParticipants;
  int nGames;
  int nActions;

  // action chosen. 1=Rock, 2=Paper, 3=Scissors, 4=Dumb
  int direction[nPeriods,nParticipants];

  // =1 if low payoffs used, =2 if high payoffs used
  int HiPay[nPeriods,nParticipants];
```

[43] But this was still fairly trivial. The kids were having a Nerf battle nearby while I coded, and I was still able to do it :)

```
  // first element: low payoffs exchange rate
  // second element: high payoffs exchange rate
  real payoff_multipliers[2];

  // =1 if the observation should contribute to the likelihood
  int UseData[nPeriods,nParticipants];


  // Pre-calculated components of the EWA model
  real PI[nPeriods,nParticipants,nActions];
  real I[nPeriods,nParticipants,nActions];

  // priors
  real prior_phi[2]; // Beta
  real prior_delta[2]; // Beta
  real prior_rho[2]; // Beta
  real prior_lambda[2]; // log-normal

  real max_payoff;

  vector[2] prior_MU[4];
  vector[4] prior_TAU;
  real prior_OMEGA;

  int nSimPars;
}

transformed data {
  matrix[nParticipants,nActions] DIRECTION[nPeriods];
  matrix[4,nSimPars] zsim;

  for (pp in 1:4) {
    for (ss in 1:nSimPars) {
      zsim[pp,ss] = std_normal_rng();
    }
  }


  for (tt in 1:nPeriods) {
    DIRECTION[tt] = rep_matrix(0.0,nParticipants,nActions);
    for (ii in 1:nParticipants) {
      for (aa in 1:nActions) {
        if (direction[tt,ii]==aa) {
          DIRECTION[tt][ii,aa] = 1.0;
        }
      }
    }
  }
}

parameters {

  vector[4] MU;
```

```
  vector<lower=0>[4] TAU;

  cholesky_factor_corr[4] L_OMEGA;

  matrix[4,nParticipants] z;



}

transformed parameters {
  vector[nParticipants] phi;
  vector[nParticipants] rho;
  vector[nParticipants] delta;
  vector[nParticipants] lambda;

  {
    matrix[4,nParticipants] theta = MU*rep_row_vector(1.0,nParticipants)+ diag_pre_multiply(TAU, L_OMEGA

    phi = Phi_approx(to_vector(theta[1,]));
    rho = Phi_approx(to_vector(theta[2,]));
    delta = Phi_approx(to_vector(theta[3,]));
    lambda = Phi_approx(to_vector(theta[4,]));

  }
}

model {

  // hierarchical structure
  to_vector(z) ~ std_normal();
  L_OMEGA ~ lkj_corr_cholesky(prior_OMEGA);
  TAU ~ cauchy(0.0,prior_TAU);

  for (pp in 1:4) {
    MU[pp] ~ normal(prior_MU[pp][1],prior_MU[pp][2]);
  }




  // initial conditions. See CFH201 footnote 8
  matrix[nParticipants,nActions] A = rep_matrix(1.0,nParticipants,nActions);
  vector[nParticipants] N = rep_vector(0.0,nParticipants);



  for (tt in 1:nPeriods) {

    // increment the likelihood
    matrix[nParticipants,nActions] lA = lambda*rep_row_vector(1.0,nActions) .*(to_vector(payoff_multipli
    vector[nParticipants] lsum_lA = log(exp(lA)*rep_vector(1.0,nActions));
```

```
    target+=to_vector(UseData[tt,]) .* (((lA-lsum_lA*rep_row_vector(1.0,nActions)) .* DIRECTION[tt])*rep

    // Update A and N

    matrix[nParticipants,nActions] I_t = to_matrix(I[tt,,]);
    matrix[nParticipants,nActions] pi_t = to_matrix(PI[tt,,]);

    A = ((phi*rep_row_vector(1.0,nActions)) .*(N*rep_row_vector(1.0,nActions)).*A+(delta*rep_row_vector
          ./ ((rho*rep_row_vector(1.0,nActions)) .* (N * rep_row_vector(1.0,nActions))+1.0);

    N = rho .* N+1.0;

  }

  // priors

  phi ~ beta(prior_phi[1],prior_phi[2]);
  rho ~ beta(prior_rho[1],prior_rho[2]);
  delta ~ beta(prior_delta[1],prior_delta[2]);
  lambda ~ lognormal(prior_lambda[1],prior_lambda[2]);

}

generated quantities {

  real mean_phi;
  real mean_rho;
  real mean_delta;
  real mean_lambda = exp(MU[4]+0.5*TAU[4]^2);

  real median_phi = Phi_approx(MU[1]);
  real median_rho = Phi_approx(MU[2]);
  real median_delta = Phi_approx(MU[3]);
  real median_lambda = exp(MU[4]);

  real sd_phi;
  real sd_rho;
  real sd_delta;
  real sd_lambda = sqrt((exp(TAU[4]^2)-1.0)*exp(2*MU[4]+TAU[4]^2));

  real CV_phi;
  real CV_rho;
  real CV_delta;
  real CV_lambda;


  matrix[4,4] OMEGA;
  OMEGA = L_OMEGA*L_OMEGA';
```

```
  {
    matrix[4,nSimPars] theta = MU*rep_row_vector(1.0,nSimPars)+ diag_pre_multiply(TAU, L_OMEGA) * zsim;

    vector[nSimPars] phi_sim = Phi_approx(to_vector(theta[1,]));
    vector[nSimPars] rho_sim = Phi_approx(to_vector(theta[2,]));
    vector[nSimPars] delta_sim = Phi_approx(to_vector(theta[3,]));
    vector[nSimPars] lambda_sim = Phi_approx(to_vector(theta[4,]));

    mean_phi = mean(phi_sim);
    mean_rho = mean(rho_sim);
    mean_delta = mean(delta_sim);

    sd_phi = sd(phi_sim);
    sd_rho = sd(rho_sim);
    sd_delta = sd(delta_sim);
  }

  CV_phi = sd_phi/mean_phi;
  CV_rho = sd_rho/mean_rho;
  CV_delta = sd_delta/mean_delta;
  CV_lambda = sd_lambda/mean_lambda;
}
```

If you look at the *R* code below implementing this estimator, I got the `bulk ESS low` error when running this with *Stan*'s default `iter = 2000`, where ESS stands for "Effective Sample Size". This means that the 4,000 posterior draws[44] were dependent enough that they did not provide sufficient information to accurately approximate posterior moments. The simple fix to this, which worked for me, was just to run the chains for longer. Doubling the number of iterations to `iter = 4000` fixed this.[45]

As mentioned above if left unchecked, this *Stan* fit object will take up quite a bit of space if we save all of the variables. As I don't want to report anything on the standardized individual-level parameters z, I use the `pars = c("z"), include=FALSE,` options in `sampling`, which drops these variables. What is left are the the individual-level parameters $\begin{pmatrix}\phi_i & \rho_i & \delta_i & \lambda_i\end{pmatrix}$, and the population-level draws for μ, τ, and Ω. Even so, this produces a *Stan* fit object that takes up about 140MB, most of which is storing the individual-level parameters.[46]

10.4.3 Results

For the hierarchical model, I just estimate one model using all of the data from all four treatments. This is because I was having trouble getting *Stan* to estimate the treatment-specific hierarchical models without returning a `divergent transitions` error. I will work on fixing these errors in a future iteration of this chapter.

A summary of this model in shown in Table 20. One of the things I really like about the way Cason, Friedman, and Hopkins (2010) display the estimates from their hierarchical models in Table 3 is that instead of reporting the models' fundamental population-level parameters (i.e. μ, τ, and Ω for me), they instead focus on moments of the implied individual-level parameters. That is, they report the mean, median, and coefficient of variation (standard deviation divided by mean) for ϕ_i, ρ_i, and λ_i. This makes their marginal distributions much easier

[44] By default, *Stan* runs 4 chains with 2,000 iterations, but uses 1,000 of these for warm-up. So we are left with 1,000 draws per chain.

[45] The model using all of the data took a little bit over four hours to run on my laptop with this configuration, so while there is plenty of time to solve a shrine or two while it runs, Hyrule will have to wait a little while longer for its salvation.

[46] In contrast though, my job market paper (Bland 2019a) estimates a hierarchical model with a similar number of individual-level parameters and participants, but took up something closer to 1GB on my hard drive. This is because the Metropolis-Hastings within Gibbs sampler I implemented for that paper requires many more posterior draws to get the same effective sample size. Hamiltonian Monte Carlo, which is what *Stan* uses, generally is much more efficient with its draws.

to interpret. I follow suit in doing this, as well as reporting the standard deviations. As you can see in the `generated quantities` block of my *Stan* program above, these are fairly easy to compute using explicit formulas for the median and for all moments of λ_i, which is log-normally distributed. The other parameters are probit-normally distributed, and so explicit expressions of the mean and standard deviations do not exist. I use Monte Carlo integration to evaluate these.

```
rounding<-3
pars<-c("phi","rho","delta","lambda")
parUnicode<-c("$\\psi$","$\\rho$","$\\delta$","$\\lambda$")
summaries<-c("mean","median","sd","CV")

param_list<-c()
param_labels<-c()
param_summaries<-c()
for (pp in 1:length(pars)) {
  for (ss in summaries) {
    param_list<-c(param_list,paste0(ss,"_",pars[pp]))
    param_labels<-c(param_labels,paste0(parUnicode[pp]))
    param_summaries<-c(param_summaries,ss)
  }
}
FitALL<-readRDS("Code/CFH2010TASP/Estimates_Hierarchical_Pooled.rds")

print(FitALL |> check_hmc_diagnostics())
```

```
##
## Divergences:

## 0 of 8000 iterations ended with a divergence.

##
## Tree depth:

## 0 of 8000 iterations saturated the maximum tree depth of 10.

##
## Energy:

## E-BFMI indicated no pathological behavior.

## NULL
```

```
  estimates<-summary(FitALL)$summary|>
  data.frame() |>
  mutate(msd = paste0(mean |> round(rounding)," (",sd |> round(rounding),")"))

   TAB<-tibble(estimates = estimates[param_list,"msd"],
               param_labels = param_labels,
               `Population property` = param_summaries) |>
     pivot_wider(id_cols = `Population property`,names_from = param_labels,values_from = estimates)




TAB |>
  kbl(caption="Properties of the marginal distributions of parameters. Posterior means with standard dev
  add_header_above(c(" ", "Parameter" = 4)) |>
```

Table 20: Properties of the marginal distributions of parameters. Posterior means with standard deviations in parentheses.

	Parameter			
Population property	ψ	ρ	δ	λ
mean	0.861 (0.013)	0.369 (0.051)	0.072 (0.016)	0.706 (0.073)
median	0.888 (0.012)	0.347 (0.057)	0.069 (0.014)	0.602 (0.056)
sd	0.112 (0.013)	0.195 (0.041)	0.016 (0.023)	0.431 (0.094)
CV	0.13 (0.017)	0.533 (0.114)	0.203 (0.241)	0.607 (0.089)

Table 21: Correlation matrix from the hierarchical model using all data. Posterior means with standard deviations in parentheses. Correlations are for transformed parameters, all of which are marginally normal.

	ψ	ρ	δ	λ
ψ	1	0.593 (0.17)	0.006 (0.262)	-0.676 (0.101)
ρ	0.593 (0.17)	1	0.042 (0.269)	-0.418 (0.185)
δ	0.006 (0.262)	0.042 (0.269)	1	-0.07 (0.264)
λ	-0.676 (0.101)	-0.418 (0.185)	-0.07 (0.264)	1

```
  kable_classic(full_width=F)
```

While I don't have a direct comparison to the middle panel in Table 3 of Cason, Friedman, and Hopkins (2010), note that my posterior mean estimates of ϕ and δ are in the same ballpark, but I get much smaller estimates for ρ (their estimates are much closer to one). Again, I get larger estimates for λ, but I suspect this is due to using different exchange rates. The conclusion that estimated behavior is much closer to reinforcement learning than stochastic fictitious play (i.e. that δ is far away from one) is supported both by my hierarchical model, and the original hierarchical models in Cason, Friedman, and Hopkins (2010).

What about the correlation matrix Ω? This is the one big piece of information that is hidden from us in Table 20. I show this correlation matrix in Table 21. Note that this shows posterior moments of Ω, which is the correlation matrix of the *transformed* individual-level parameters. If we wanted to estimate the correlation matrix of the actual parameters, we could do this using Monte Carlo integration, similarly to how I calculated some of the poseterior means for Table 20. As all transforms are monotonic, we can at lest interpret positive numbers in this table as positive correlations between the real individual-level parameters. Here we can see that there appears to be a substantial positive correlation between ϕ and ρ, and substantial negative correlations between ϕ and λ, and ρ and λ. Note that in Table 3 of Cason, Friedman, and Hopkins (2010), they assume that there is no correlation between the individual-level parameters, and so their correlation matrix would just be the identity matrix.

```
rounding<-3
CORR <- rstan::extract(FitALL)$OMEGA

CORRmean<-CORR |> apply(MARGIN = c(2,3),FUN = mean) |> round(rounding)
CORRsd<-CORR |> apply(MARGIN = c(2,3),FUN = sd) |> round(rounding)

CORRmsd<-paste0(CORRmean," (",CORRsd,")") |> matrix(nrow=4)

rownames(CORRmsd)<-parUnicode
colnames(CORRmsd)<-parUnicode

CORRmsd[CORRmsd=="1 (0)"]<-"1"

CORRmsd |> kbl(caption = "Correlation matrix from the hierarchical model using all data. Posterior means
```

One of the cool things you get out of a hierarchical model estimated using data augmentation, that you don't get from one using Monte Carlo integration, is "shrinkage" estimates of the individual-level parameters.[47] Figure 36 shows histograms of the posterior means of these.

```
phi<-extract(FitALL)$phi
rho<-extract(FitALL)$rho
delta<-extract(FitALL)$delta
lambda<-extract(FitALL)$lambda

IndParams<-tibble()

for (ii in 1:dim(phi)[2]) {
  IndParams<-rbind(
    IndParams,
    tibble(
      id = ii,
      phi = phi[,ii],
      rho = rho[,ii],
      delta = delta[,ii],
      lambda = lambda[,ii]
    )
  )
}

IndParamsLong<-IndParams |>
  pivot_longer(cols = c(phi,rho,delta,lambda),names_to="parameter")
```

```
(
  ggplot(
    IndParamsLong |>
      group_by(id,parameter) |>
      summarize(mean = mean(value)),
    aes(x=mean)
  )
  +geom_histogram()
  +theme_bw()
  +facet_wrap(~parameter,scales="free")
  +xlab("posterior mean estimate")
)
```

One way I like to visualize the individual-level parameters is to show the empirical cumulative density function (ecdf) of the posterior mean overlaid with a Bayesian credible region. This gives you a sense of both scale and precision for these parameters. I do this in Figure 37, which overlays the ecdf with 90% Bayesian credible regions. Here we can see that there is very little heterogeneity in δ, evidenced by the near vertical ecdf and small horizontal scale on this panel. Coupled with the narrow credible regions only covering values close to zero, we can be fairly certain for most participants that reinforcement learning ($\delta = 0$) is a *much* better approximation of behavior than is weighted fictitious play ($\delta = 1$). ρ appears to be the parameter that we learn the least about, as many of its credible regions cover a wide range of the unit interval. On the other hand, ϕ is estimated reasonably accurately.

```
(
  ggplot(
    IndParamsLong |>
```

[47] These *can* be computed post-estimation from models that use Monte Carlo integration, but they are computed by default when when we use data augmentation.

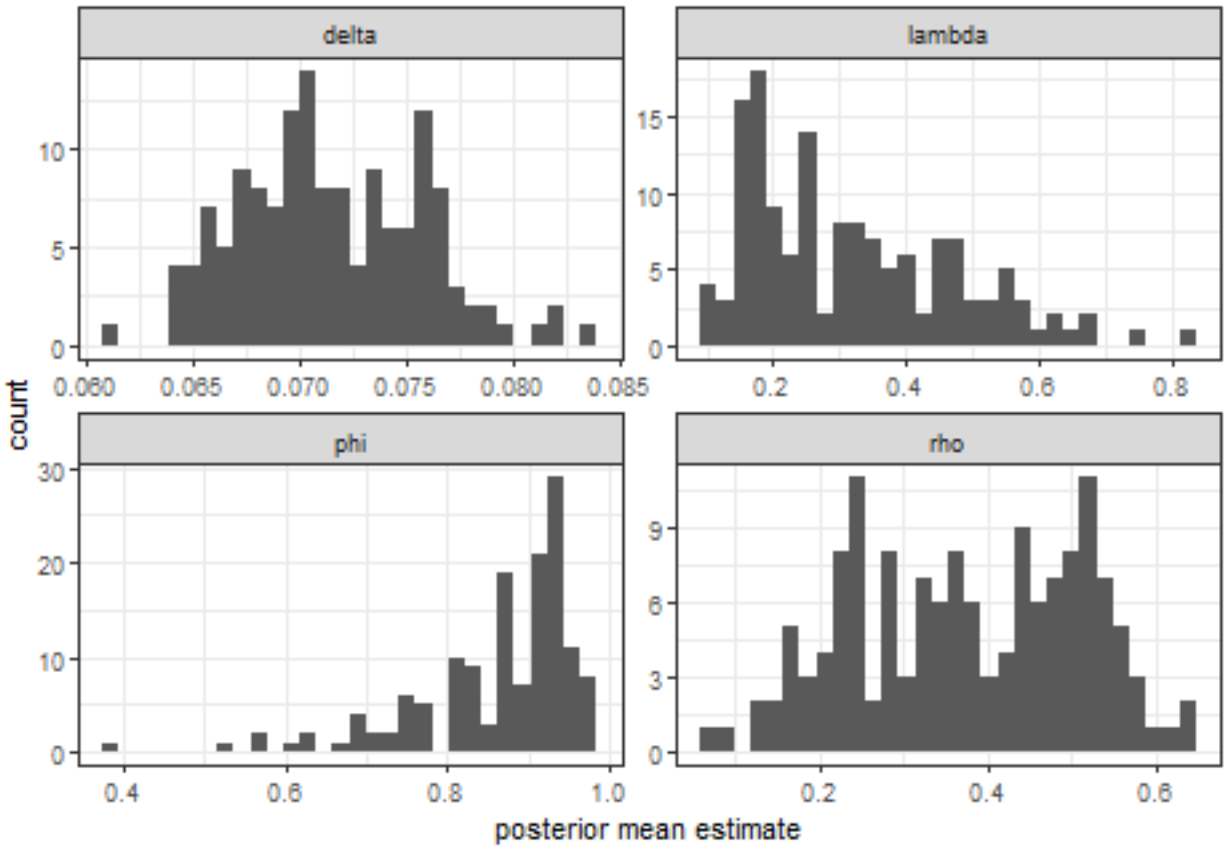


Figure 36: Posterior mean estimates of individual-level parameters.

```
      group_by(id,parameter) |>
      summarize(mean = mean(value),
                p05 = quantile(value,probs = 0.05),
                p95 = quantile(value,probs = 0.95)
                ) |>
      ungroup() |>
      group_by(parameter) |>
      arrange(mean) |>
      mutate(ecdf = (1:n())/n())

    ,
    aes(x=mean,y=ecdf,xmin=p05,xmax=p95)
  )
  #+geom_point(size = 0.2)
  +stat_ecdf()
  +geom_errorbar(alpha = 0.1)
  +theme_bw()
  +facet_wrap(~parameter,scales="free")
  +xlab("posterior mean estimate")
  +ylab("empirical cumulative density")
)
```

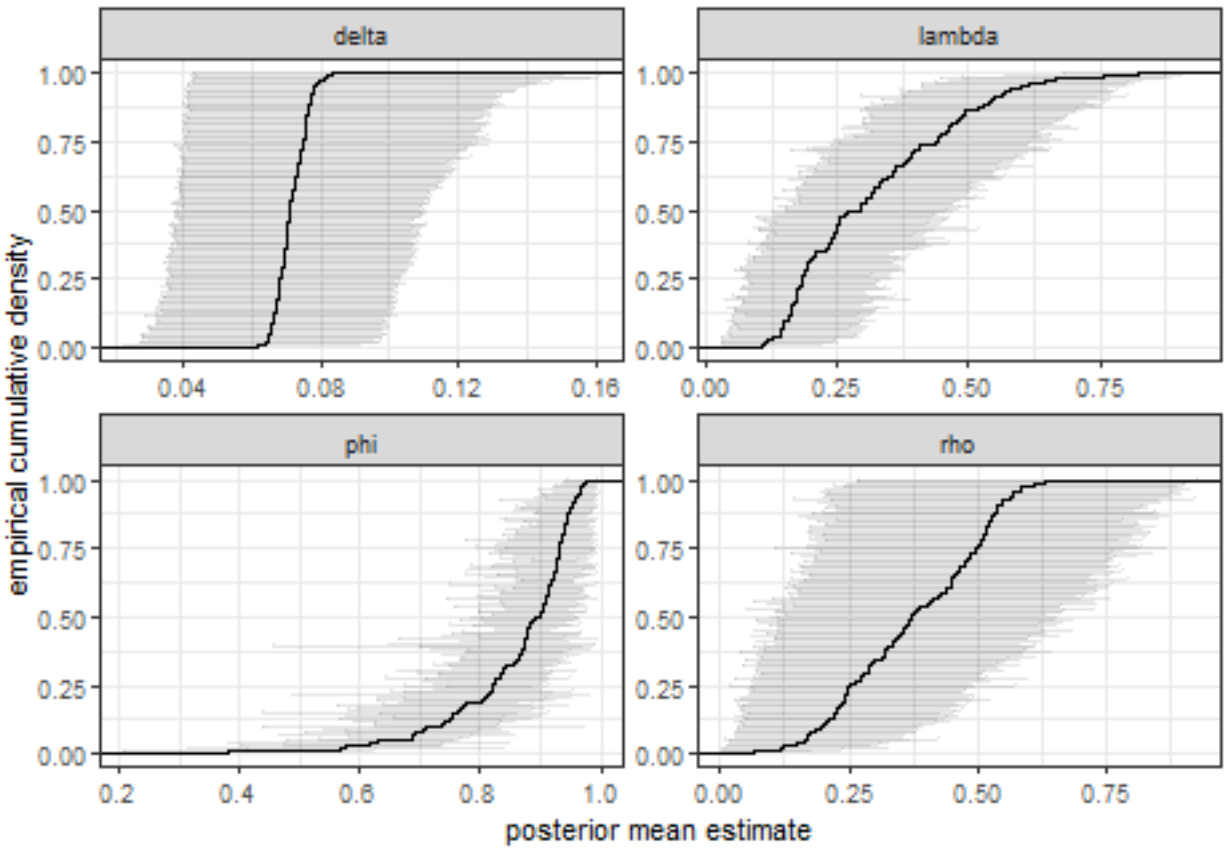


Figure 37: Posterior estimates of individual-level parameters. Empirical cumulative density of posterior means overlayed with 90% Bayesian credible regions.

10.5 Some code used to estimate the models

Here is the *R* code used to load the data and estimate the models.

10.5.1 Loading the data

```
library(tidyverse)
library(rstan)
rstan_options(auto_write = TRUE)
options(mc.cores = parallel::detectCores())

D<- "Data/CFH2010TASP/TASP-12Sess.csv" |>
  read.csv() |>
  select(session,period,subject,stableID,hi_payID,group,direction) |>
  mutate(id = paste(session,subject) |> as.factor() |> as.numeric()) |>
  # direction is coded as 0,1,2,3. If I code it as 1,2,3,4 then I can reference
  # the payoff matrix
  mutate(direction = direction+1) |>
  arrange(id,period) |>
  group_by(period,session,group) |>
  mutate(direction_other = sum(direction)-direction) |>
  ungroup() |>
  # Link this to the payoff matrices below
  mutate(gameCode = stableID+1,
         hi_payID = hi_payID+1
         )


#----------------------------------------
# Wrangle data into a Stan-friendly format
#----------------------------------------

# Here I am going to make everything TxN arrays, or in some cases TxNx4

nPeriods<-length(D$period |> unique())
nParticipants<-length(D$id |> unique())

# PROBLEM: There was a tornado during session 4, and so these participants
# only played 70 rounds, not 80. To see this, note this
D |> group_by(session) |> summarize(maxPeriod=max(period))

ID <-array(0,dim = c(nPeriods,nParticipants))
PERIOD <-array(0,dim = c(nPeriods,nParticipants))
DIRECTION<-array(1,dim = c(nPeriods,nParticipants))
DIRECTION_OTHER<-array(1,dim = c(nPeriods,nParticipants))
GAME_CODE<-array(1,dim = c(nPeriods,nParticipants))
HIPAY<-array(1,dim = c(nPeriods,nParticipants))

for (ii in 1:length(unique(D$id))) {
  d<-D |> filter(id==ii)

  ID[1:dim(d)[1],ii]<-d$id
  PERIOD[1:dim(d)[1],ii]<-d$period
  DIRECTION[1:dim(d)[1],ii]<-d$direction
```

```
  DIRECTION_OTHER[1:dim(d)[1],ii]<-d$direction_other
  GAME_CODE[,ii]<-d$gameCode[1]
  HIPAY[1:dim(d)[1],ii]<-d$hi_payID
}

USEDATA<-1*(ID!=0)









payoffs<-list(
  #Unstable
  rbind(
    c(90,0,120,20),
    c(120,90,0,20),
    c(0,120,90,20),
    c(90,90,90,0)
  ),
  #Stable
  rbind(
    c(60,0,150,20),
    c(150,60,0,20),
    c(0,150,60,20),
    c(90,90,90,0)
  )
)


nActions<-dim(payoffs[[1]])[1]

PAYOFFS<-array(NA,dim=c(2,4,4))

PAYOFFS[1,,]<-payoffs[[1]]
PAYOFFS[2,,]<-payoffs[[2]]




gameCode<-(D |> filter(period==1))$gameCode


PI<-array(0,dim=c(nPeriods,nParticipants,nActions))
I<-array(0,dim=c(nPeriods,nParticipants,nActions))
for (ii in 1:nParticipants) {
  U<-payoffs[[gameCode[ii]]]
  for (aa in 1:nActions) {
    PI[,ii,aa]<-U[aa,DIRECTION_OTHER[,ii]]
    I[,ii,aa]<-1*(DIRECTION[,ii]==aa)
  }
}
```

```
dStan<-list(
 nPeriods = nPeriods,
 nParticipants = nParticipants,
 nGames = length(payoffs),
 nActions = nActions,

 direction = DIRECTION,

 UseData = USEDATA,
 GameCode = GAME_CODE,

 HiPay = HIPAY,
 payoff_multipliers = c(0.035,0.035),

 payoffs = PAYOFFS,

 PI = PI,
 I  = I,

 prior_phi = c(1,1),
 prior_delta = c(1,1),
 prior_rho = c(1,1),
 prior_lambda = c(1.15,1.76),

 max_payoff = 150*0.035

)

dStan |> saveRDS("Data/CFH2010TASP/CFH2010TASP_dStan.rds")
```

10.5.2 Estimating the representative agent models

```
library(tidyverse)
library(rstan)

dStan <- readRDS("Data/CFH2010TASP/CFH2010TASP_dStan.rds")



RAmodel<-stan_model("Code/CFH2010TASP/EWA.stan")

# Model pooling all data
file<-"Code/CFH2010TASP/Estimates_RA_Pooled.rds"
if (!file.exists(file)) {
  print(paste("estimating",file))
  Fit<-sampling(RAmodel,data=dStan,seed=42)
  summary(Fit)$summary |> saveRDS(file=file)
}

# Unstable low payoffs
dd<-dStan$UseData * (dStan$HiPay==1) * (dStan$GameCode==1)
d<-dStan
```

```
d$UseData<-dd
file<-"Code/CFH2010TASP/Estimates_RA_UnstableLow.rds"
if (!file.exists(file)) {
  print(paste("estimating",file))
  Fit<-sampling(RAmodel,data=d,seed=42,iter=2000)
  summary(Fit)$summary |> saveRDS(file=file)
}

# Unstable high payoffs
dd<-dStan$UseData * (dStan$HiPay==2) * (dStan$GameCode==1)
d<-dStan
d$UseData<-dd
file<-"Code/CFH2010TASP/Estimates_RA_UnstableHi.rds"
if (!file.exists(file)) {
  print(paste("estimating",file))
  Fit<-sampling(RAmodel,data=d,seed=42)
  summary(Fit)$summary |> saveRDS(file=file)
}

# Stable low payoffs
dd<-dStan$UseData * (dStan$HiPay==1) * (dStan$GameCode==2)
d<-dStan
d$UseData<-dd
file<-"Code/CFH2010TASP/Estimates_RA_StableLow.rds"
if (!file.exists(file)) {
  print(paste("estimating",file))
  Fit<-sampling(RAmodel,data=d,seed=42)
  summary(Fit)$summary |> saveRDS(file=file)
}

# Stable high payoffs
dd<-dStan$UseData * (dStan$HiPay==2) * (dStan$GameCode==2)
d<-dStan
d$UseData<-dd
file<-"Code/CFH2010TASP/Estimates_RA_StableHi.rds"
if (!file.exists(file)) {
  print(paste("estimating",file))
  Fit<-sampling(RAmodel,data=d,seed=42)
  summary(Fit)$summary |> saveRDS(file=file)
}
```

10.5.3 Estimating the hierarchical model

```
library(tidyverse)
library(rstan)
  options(mc.cores = parallel::detectCores())
  rstan_options(auto_write = TRUE)


dStan <- readRDS("Data/CFH2010TASP/CFH2010TASP_dStan.rds")
```

```
model<-stan_model("Code/CFH2010TASP/EWAhierarchical.stan")

dStan$prior_MU<-list(c(0,0.25),
                     c(0,0.25),
                     c(0,0.25),
                     c(1.15,0.75))

dStan$prior_TAU<-c(0.05,0.05,0.05,0.05)

dStan$prior_OMEGA<-5

dStan$nSimPars<-1000

file<-"Code/CFH2010TASP/Estimates_Hierarchical_Pooled.rds"
if (!file.exists(file)) {
  Fit<-sampling(model,
                data=dStan,
                pars = c("z"),include=FALSE,
                seed=42,
                # The first time running this with the default iter = 2000
                # yielded the "Bulk ESS low" error. This can be remedied with
                # a larger simulation size. Running with these options
                # produced no error
                iter=4000)
  Fit |> saveRDS(file)
}


# I need to work on fixing the divergences here
# So I will hold off on estimating the treatment-specific
# models
```

11 Application: Strategy Frequency Estimation

One of the econometric models that most sparked my interest as a grad student was the Strategy Frequency Estimation Method (SFEM), which first showed up in Dal Bó and Fréchette (2011) and Fudenberg, Rand, and Dreber (2012). In these applications, and a lot that followed, the SFEM is applied to data from infinitely repeated Prisoner's Dilemma experiments. In these games, strategies can be quite complicated beasts, because they must prescribe an action to take after every possible history of play, and histories in an infinitely repeated game can be, well ... infinite. We therefore cannot *observe* strategies (or the pure-action realizations of mixed strategies) like we would in a single-shot game.[48] However we cannot just ignore strategies in these games, because a lot of the theory motivating why cooperation will or will not occur has to do with strategies that support cooperation. The idea behind the SFEM is to focus on a menu of strategies that we have decided could be empirically relevant. We then estimate the fraction of participants whose behavior in our experiment is best described by each of these strategies. Hence, the SFEM is a mixture model.

A strategy s in an indefinitely repeated game can be defined as a function that maps the history of play $h_{i,t-1}$ into a probability distribution over the actions in player i's choice set in the next period of the game, $y_{i,t} \mid h_{i,t-1}, s$, where $y_{i,t}$ is the action taken by participant i in period t. While these strategies can in principle be mixed, unless they are *fully* mixed they will not assign positive probability to all actions, and so we will run into the zero-likelihood problem. The SFEM's solution to this is to assume that players *tremble*. That is, participants follow the action specified by the strategy with probability $1-\epsilon$, and choose another

[48] Two exceptions to this are Romero and Rosokha (2018) and Dal Bó and Fréchette (2019).

action with probability ϵ. In the most common application of the SFEM, which is indefinitely repeated Prisoner's dilemmas, players' choice sets have just two elements (cooperate and defect), so it makes sense that $\epsilon \in (0, 0.5)$. This ensures that players are more likely to choose the action specified by their strategy than not. For this chapter, I will focus on the SFEM when the choice set is binary[49] I will further focus only on strategies that specify pure actions.

The strategy s, tremble probability ϵ, history $\{h_{i,t-1}\}$, and data $\{y_{i,t}\}$ is sufficient information to write down the probability of an individual decision:

$$p(y_{i,t} \mid h_{i,t-1}, s, \epsilon) = (1-\epsilon)^{I(y_{i,t}=s(h_{i,t-1}))}\epsilon^{I(y_{i,t}\neq s(h_{i,t-1}))}$$

where $I(\cdot)$ is the indicator function. That is, when $y_{i,t} = s(h_{i,t-1})$ the participant has followed the prescribed action of the strategy, which happens with probability $1-\epsilon$, and when $y_{i,t} \neq s(h_{i,t-1})$ the participant has not followed the prescribed action of the strategy, which happens with probability ϵ.

As the goal of the SFEM is to estimate the fraction of *participants* who use each strategy, we combine these decision-level likelihoods and mixing probabilities $\{\rho_s\}_{s=1}^{S}$ into a grand likelihood using a mixture model specification:

$$\log p(y \mid \rho, \epsilon) = \sum_{i=1}^{N} \log \left(\sum_{s=1}^{S} \exp \left(\log \rho_s + \sum_{t=1}^{T_i} \log p(y_{i,t} \mid h_{i,t-1}, s, \epsilon) \right) \right)$$

This is one of these situations where we can take advantage of *Stan*'s `log_sum_exp` function, so I have added the logged mixing probabilities to the exponent, rather than multiplying them in levels. Also, it is important to keep track of the i subscript on T_i, the total number of actions taken by participant i. Indefinitely repeated games is one of the situations where we do not have a balanced panel, as the length of a game must be random.

11.1 Simplifying the individual likelihood functions

For pure strategies, there is a lot we can do to speed up estimation. Specifically, we can simplify the summation over t in the above grand likelihood by pre-summarizing our data: we don't actually need to know for which *specific decisions* a participant followed a strategy or trembled, we just need to know the *total number of times* they followed a strategy, and the total number of times they trembled. This is because we have assumed that actions are independent conditional on $h_{i,t-1}$, s, and ϵ. To see this, note that:

$$\begin{aligned}
\sum_{t=1}^{T_i} \log p(y_{i,t} \mid h_{i,t-1}, s, \epsilon) &= \sum_{t=1}^{T_i} \left[I(y_{i,t} = s(h_{i,t-1})) \log(1-\epsilon) + I(y_{i,t} \neq s(h_{i,t-1})) \log(\epsilon) \right] \\
&= \log(1-\epsilon) \sum_{t=1}^{T_i} I(y_{i,t} = s(h_{i,t-1})) + \log(\epsilon) \sum_{t=1}^{T_i} I(y_{i,t} \neq s(h_{i,t-1})) \\
&= \log(1-\epsilon)\text{follow}_{i,s} + \log(\epsilon)(T_i - \text{follow}_{i,s})
\end{aligned}$$

where $\text{follow}_{i,s}$ is the number of times participant i followed the prescribed action in strategy s. Importantly, $\text{follow}_{i,s}$ can be pre-calculated, so we can speed up our estimation by doing this.

[49] For choice sets with more than two elements we need to be a bit more careful with how we specify the trembles. Since I am focusing attention on indefinitely-repeated Prisoner's dilemma experiments, where the choice set is binary, I will not go into more detail here, but might later in a follow-up chapter.

id	follow1	follow2	follow3	follow4	follow5	follow6	n	Treatment
1	21	4	13	13	8	9	25	1
2	25	0	17	16	1	13	25	1
3	25	0	17	17	0	13	25	1
4	22	3	14	15	3	12	25	1
5	25	0	17	15	2	13	25	1
6	13	12	5	11	11	9	25	1

11.2 Example experiment: Dal Bó and Fréchette (2011)

In their Table 7, Dal Bó and Fréchette (2011) estimate the fraction of $S = 6$ strategies being used in their indefinitely repeated Prisoner's dilemma experiment.[50] These strategies are:

1. Always Defect (AD)
2. Always Cooperate (AC)
3. Grim Trigger (G), defect for ever once one person has defected
4. Tit for Tat (TFT)
5. Win Stay Lose Shit (WSLS)
6. Tit for two tats (T2)

From their footnote 19:

> WSLS is a strategty hat starts cooperating and then condition behavior only on the outcome of the previous round. If either both cooperate or neither cooperate, then WSLS cooperates; otherwise it defects. T2 starts cooperating, and a defection by the other triggers two rounds of defection, after which the strategy goes back to cooperation. These two strategies are cooperative st rategies with punishments of limited length.

Each strategy specifies an action (for an infinitely repeated Prisoner's Dilemma, either "Cooperate" or "Defect") to be played after every possible combination of strategies. The trouble, with a few notable exceptions, is that we do not observe strategies, we only observe the actions that result from these strategies, and so we have data like this:

```
DBFData<-readRDS('Data/SFEM/DBFData.rds')
DBFData |> head() |> kbl() |> kable_classic(full_width=F)
```

Each row here is a participant in the experiment. Follow1 is a count of the number of times this participant played the action consistent with AD, likewise for follow2, and so on. n is the number of actions this participant played (some data from early rounds were excluded).

11.2.1 The SFEM with homogeneous trembles

Here is my implementation of the model in *Stan*. I chose not to use the `log_sum_exp` function here because it would have meant that I could not vectorize the calculation of the `target += log(like_i*mix)` line. In order to make the estimates directly comparable to Table 7 of Dal Bó and Fréchette (2011), I estimate one model for each of their six treatments.[51]

```
// SFEM.stan
functions {



}
data {
  // Tell Stan how to read in the data
```

[50] You can download their data (and *Matlab* code to replicate it!) from here

[51] These estimations run very quickly (a second or two on my laptop each), so it's probably not worth trying to find a korok while it runs.

```
  int N; // Number of subjects
  int S; // Number of strategies
  matrix[N,S] COUNTS; // Number of decisions by each player, here I repeat the vector S times to make t
  matrix[N,S] FOLLOWS; // counts of following each strategy
  vector[S] priorMix; // Dirichlet prior for mixing probabilities
}
transformed data {

}
parameters {
  real<lower=0,upper=0.5> trmb; // tremble probability
  simplex[S] mix; // strategy mixing probabilities


}
transformed parameters {

}
model {
    matrix[N,S] like_i;

    mix ~ dirichlet(priorMix);

  // Individual likelihoods
    like_i = exp(FOLLOWS*log(1-trmb)+(COUNTS-FOLLOWS)*log(trmb));

    // Adding the grand likelihood to the target
    target += log(like_i*mix);




  }
generated quantities {

}
```

The estimates from these models are shown in Table @??tab:SFEMSimpleTable), which is arranged in almost the same way as Table 7 in Dal Bó and Fréchette (2011).

```
TAB<-readRDS("Code/SFEM/SFEMHomogeneousTremblesEstimates.rds")
rownames(TAB)<-c("AD","AC","G","TFT","WSLS","T2","tremble")

TAB |>
  kbl(caption = "SFEM estimates from the six treatments of @DBF2011. Posterior means with standard devia
  kable_classic(full_width=F) |>
  add_header_above(c(" "=1,"R = 32"=1,"R = 40"=1,"R = 48"=1,"R = 32"=1,"R = 40"=1,"R = 48"=1)) |>
  add_header_above(c(" "=1,"$\\delta=\\frac{1}{2}$"=3,"$\\delta=\\frac{3}{4}$"=3))
```

Comparing my estimates in Table 22 to Table 7 of Dal Bó and Fréchette (2011), there are many similarities, but my mixing probabilities are all closer to $\frac{1}{6}$ (i.e. large probabilities are attenuated, small probabilities are inflated). This is because $\frac{1}{6}$ is the prior mean, so we should expect this. Some common themes between these tables are:

- Mostly AD in the $R = 32$ or $R = 40$, $\delta = \frac{1}{2}$ treatments

Table 22: SFEM estimates from the six treatments of @DBF2011. Posterior means with standard deviations in parentheses. Each colum is one of the six treatments. R is the payoff when both payers cooperate, and δ is the continuation probability.

	$\delta=\frac{1}{2}$			$\delta=\frac{3}{4}$		
	R = 32	R = 40	R = 48	R = 32	R = 40	R = 48
AD	0.83 (0.053)	0.716 (0.061)	0.49 (0.07)	0.59 (0.071)	0.118 (0.048)	0.02 (0.021)
AC	0.02 (0.02)	0.083 (0.04)	0.084 (0.04)	0.021 (0.021)	0.27 (0.094)	0.112 (0.072)
G	0.02 (0.019)	0.051 (0.032)	0.035 (0.033)	0.034 (0.029)	0.225 (0.096)	0.145 (0.1)
TFT	0.09 (0.042)	0.107 (0.045)	0.332 (0.071)	0.314 (0.069)	0.297 (0.11)	0.455 (0.117)
WSLS	0.02 (0.019)	0.021 (0.02)	0.037 (0.027)	0.02 (0.019)	0.045 (0.043)	0.057 (0.053)
T2	0.02 (0.019)	0.022 (0.022)	0.022 (0.022)	0.02 (0.02)	0.045 (0.041)	0.211 (0.112)
tremble	0.06 (0.008)	0.137 (0.01)	0.089 (0.008)	0.097 (0.007)	0.092 (0.01)	0.03 (0.005)

- Similar fractions of AD and TFT in the $R = 48$ $\delta = \frac{1}{2}$ and $R = 32$ $\delta = \frac{3}{4}$ treatments
- Similar fractions of TFT in the $\delta = \frac{3}{4}$ $R = 40$ and $R = 48$ treatments, and
- Similar fraction of AC in the $\delta = \frac{3}{4}$ $R = 40$ treatment.

11.2.2 Adding heterogeneous trembles and integrating the likelihood

An extension I did with the SFEM in Bland (2020) was to allow for participant-specific heterogeneous tremble probabilities ϵ_i. This can be done a few different ways. Probably the most obvious way is to assume that $2\epsilon_i$ probit-normally distributed,[52] and use data augmentation accordingly. I use this distribution in Bland (2020), but implement a maximum likelihood estimator that uses Monte Carlo integration to integrate out the ϵ_is.[53] Since the data augmentation approach is going to look like any other 1-parameter hierarchical extension that I might show you in this book, I thought I would show you a different solution, where we can find an analytical solution to integrating the likelihood. This is similar to the process of finding conjugate priors.

To begin with, I will write out the strategy-specific likelihood function for one participant's decisions if we assume a general probability density function for the distribution of ϵ, call it $p(\epsilon \mid \theta)$, where θ are some parameters describing the distribution.

$$\begin{aligned} p(y_i \mid h_{i,t-1}, s, \theta) &= \int_0^{0.5} \prod_{t=1}^{T_i} p(y_{i,t} \mid h_{i,t-1}, s, \epsilon_i) p(\epsilon \mid \theta) \mathrm{d}\epsilon \\ &= \int_0^{0.5} (1-\epsilon)^{F_{i,s}} \epsilon^{T_i - F_{i,s}} p(\epsilon \mid \theta) \mathrm{d}\epsilon \end{aligned}$$

where $F_{i,s}$ is short for the variable follow$_{i,s}$.

Looking at the likelihood part, it almost looks like an incomplete beta function, which is:

$$B(x; a, b) = \int_0^x t^{a-1}(1-t)^{b-1} \mathrm{d}t$$

so if we choose:

$$p(\epsilon \mid \theta) \propto \epsilon^{\theta_1 - 1}(1-\epsilon)^{\theta_2 - 1}$$

our likelihood becomes:

[52] Multiplying by 2 ensures that $\epsilon_i \in (0, 0.5)$.

[53] This was not for my lack of trying. I originally wrote that paper using Bayesian estimators, but the editor wanted me to re-do everything using MLE.

$$
\begin{aligned}
p(y_i \mid h_{i,t-1}, s, \theta) &\propto \int_0^{0.5} (1-\epsilon)^{F_{i,s}} \epsilon^{T_i - F_{i,s}} \epsilon^{\theta_1 - 1} (1-\epsilon)^{\theta_2 - 1} \mathrm{d}\epsilon \\
&= \int_0^{0.5} \epsilon^{\theta_1 + T_i - F_{i,s} - 1} (1-\epsilon)^{\theta_2 + F_{i,s} - 1} \mathrm{d}\epsilon \\
&= B(0.5; \theta_1 + T_i - F_{i,s}, \theta_2 + F_{i,s})
\end{aligned}
$$

So we can evaluate the likelihood, up to a constant of proportionality, using the incomplete Beta function! Now we need to work out the constant of proportionality. Fortunately for us, this is just the constant to make sure that $p(\epsilon \mid \theta)$ integrates to one, so it must be that:

$$
\begin{aligned}
1 &= c \int_0^{0.5} \epsilon^{\theta_1 - 1} (1-\epsilon)^{\theta_2 - 1} \mathrm{d}\epsilon \\
&= cB(0.5; \theta_1, \theta_2) \\
c &= \frac{1}{B(0.5; \theta_1, \theta_2)} \\
\implies p(\epsilon \mid \theta) &= \frac{\epsilon^{\theta_1 - 1} (1-\epsilon)^{\theta_2 - 1}}{B(0.5; \theta_1, \theta_2)}
\end{aligned}
$$

And so we can write the strategy-specific log-likelihood of a participant's choices as:

$$
\log p(y_i \mid h_{i,t-1}, s, \theta) = \log B(0.5; \theta_1 + T_i - F_{i,s}, \theta_2 + F_{i,s}) - \log B(0.5; \theta_1, \theta_2)
$$

This means that we have modeled the tremble probabilities ϵ_i as coming from a *truncated beta* distribution.

Unfortunately, *Stan*'s `inc_beta` function returns the *regularized* incomplete beta function, which is not what we have above. The regularized version divides the incomplete beta function by the (complete) beta function. That is, we can evaluate:

$$
\begin{aligned}
\text{(complete) beta: } B(a,b) &= \int_0^1 t^{a-1} (1-t)^{b-1} \mathrm{d}t \\
\text{regularized incomplete beta: } I_x(a,b) &= \frac{\int_0^x t^{a-1} (1-t)^{b-1} \mathrm{d}t}{B(a,b)}
\end{aligned}
$$

so we will need to add in the beta function to this expression. In summary, we have:

$$
\begin{aligned}
\log p(y_i \mid h_{i,t-1}, s, \theta) = &\log I_{0.5}(\theta_1 + T_i - F_{i,s}, \theta_2 + F_{i,s}) + \log B(\theta_1 + T_i - F_{i,s}, \theta_2 + F_{i,s}) \\
&- \log I_{0.5}(\theta_1, \theta_2) - \log B(\theta_1, \theta_2)
\end{aligned}
$$

In *Stan* `inc_beta` is the regularized incomplete beta function, and `lbeta` is the natural logarithm of the (complete) beta function.

Now that we have two more parameters, θ_1 and θ_2, we need to assign priors for these. When using the (un-truncated) beta distribution, I like to re-parameterize this into a mean parameter and a strength parameter like this:

$$
\begin{aligned}
q &= \frac{\theta_1}{\theta_1 + \theta_2} \in (0,1) \\
k &= \theta_1 + \theta_2 > 0
\end{aligned}
$$

for the un-truncated beta distribution, we can therefore think about it as having a mean q and concentration k. The interpretation is less clear for this truncated case, but I will go ahead with it and use this re-parametrized

version to select priors. Since q is the mean, and can take on any probability between 0 and 1,[54] I choose a uniform prior. For k, I use a fairly spread-out:

$$k \sim \text{Cauchy}^{+}(0, 1)$$

Here is the code I came up with to implement this model in *Stan*. I was somewhat disappointed that *Stan* didn't have a vectorized version of the `inc_beta` function, so I had to drop my vectorized implementation of calculating `like_i` from the homogeneous model. All of this is to say, I am not sure whether this implementation where we analytically solve for the integrated likelihood is actually faster than using data augmentation. This is because with data augmentation I *would* be able to vectorize the operations.

```
// SFEMBeta.stan
functions {


}
data {
  // Tell Stan how to read in the data
  int N; // Number of subjects
  int S; // Number of strategies
  matrix[N,S] COUNTS; // Number of decisions by each player, here I repeat the vector S times to make t
  matrix[N,S] FOLLOWS; // counts of following each strategy
  vector[S] priorMix; // Dirichlet prior for mixing probabilities

  real<lower=0> prior_k;
}
transformed data {

}
parameters {

  simplex[S] mix; // strategy mixing probabilities


  real<lower=0,upper=1> q; // mode of tremble distribution
  real<lower=0> k; // concentration of tremble distribution

}
transformed parameters {

  vector[2] theta;

  theta[1] = k*q;
  theta[2] = k*(1-q);

}
model {
    matrix[N,S] like_i;

    mix ~ dirichlet(priorMix);

  // Individual likelihoods
```

[54] For the truncated model, if $q < 0.5$ then q is the *mode* of the distribution.

```
  /* As far as I can tell, there is not a vectorized version of
  inc_beta. Hence the double for loop below here.

  Another thing that tripped me up was that inc_beta takes arguments
  inc_beta(a,b,x), not inc_beta(x,a,b). Always read the manual, peeps!
  */
  for (ii in 1:N) {
    for (ss in 1:S) {
      like_i[ii,ss] = exp(
                  log(inc_beta(theta[1]+COUNTS[ii,ss]-FOLLOWS[ii,ss],theta[2]+FOLLOWS[ii,ss],0.5))
                  +lbeta(theta[1]+COUNTS[ii,ss]-FOLLOWS[ii,ss],theta[2]+FOLLOWS[ii,ss])
                  -log(inc_beta(theta[1],theta[2],0.5))
                  -lbeta(theta[1],theta[2])
                );
    }
  }

    // Adding the grand likelihood to the target
    target += log(like_i*mix);

    // prior for k
    k~ cauchy(0.0,prior_k);

  }
generated quantities {

}
```

Similarly to the previous section, I estimate one model for each treatment in Dal Bó and Fréchette (2011). The estimates are summarized in Table 23. These estimates are very similar to those in Table 22, and therefore we would draw similar conclusions about strategy frequencies with these heterogeneous-trembles models. In this case the heterogeneity does not seem to be changing the results that we are most interested in much.

```
TAB<-readRDS("Code/SFEM/SFEMHeterogeneousTremblesEstimates.rds")
rownames(TAB)<-c("AD","AC","G","TFT","WSLS","T2","q","k")

TAB |>
  kbl(caption = "SFEM estimates from the six treatments of @DBF2011, assuming a heterogeneous distributi
  kable_classic(full_width=F) |>
  add_header_above(c(" "=1,"R = 32"=1,"R = 40"=1,"R = 48"=1,"R = 32"=1,"R = 40"=1,"R = 48"=1)) |>
  add_header_above(c(" "=1,"$\\delta=\\frac{1}{2}$"=3,"$\\delta=\\frac{3}{4}$"=3))
```

11.3 *R* code to do these estimations

```
library(tidyverse)
library(rstan)
options(mc.cores = parallel::detectCores())
rstan_options(auto_write = TRUE)

rounding<-3

#####################################################
```

Table 23: SFEM estimates from the six treatments of @DBF2011, assuming a heterogeneous distribution of tremble probability ϵ. Posterior means with standard deviations in parentheses. Each colum is one of the six treatments. R is the payoff when both payers cooperate, and δ is the continuation probability.

	$\delta=\frac{1}{2}$			$\delta=\frac{3}{4}$		
	R = 32	R = 40	R = 48	R = 32	R = 40	R = 48
AD	0.851 (0.051)	0.749 (0.063)	0.481 (0.072)	0.587 (0.071)	0.121 (0.049)	0.02 (0.02)
AC	0.021 (0.02)	0.047 (0.032)	0.086 (0.041)	0.021 (0.02)	0.254 (0.091)	0.11 (0.074)
G	0.021 (0.021)	0.058 (0.034)	0.039 (0.036)	0.047 (0.033)	0.247 (0.094)	0.13 (0.1)
TFT	0.065 (0.037)	0.104 (0.046)	0.337 (0.074)	0.304 (0.068)	0.285 (0.109)	0.471 (0.12)
WSLS	0.021 (0.02)	0.021 (0.02)	0.033 (0.027)	0.02 (0.02)	0.047 (0.044)	0.062 (0.057)
T2	0.021 (0.02)	0.021 (0.021)	0.024 (0.023)	0.021 (0.02)	0.046 (0.044)	0.207 (0.113)
q	0.354 (0.254)	0.547 (0.236)	0.451 (0.245)	0.4 (0.239)	0.389 (0.243)	0.226 (0.234)
k	0.728 (0.814)	0.831 (0.589)	0.676 (0.57)	1.278 (1.146)	0.673 (0.672)	1.461 (1.826)

```
# Loading the data from the files available on the
# journal website
#########################################################


DBFData<-data.frame()
for (ff in 1:6) {
  fname<-paste("Data/SFEM/dfformatlab_strg_",ff,"_special.txt",sep="")
  d<-data.frame(read.delim(fname, header = FALSE, sep = "\t", dec = "."))
  colnames(d)<-c("match","round","treatment","coop","id","ocoop_all","strg1","strg2","strg3","strg4","s
  DBFData<-rbind(DBFData,d)
}

for (ff in 1:6) {
  str<-paste("DBFData$follow",ff,"<- as.integer(DBFData$coop == DBFData$strg",ff,")",sep="")
  eval(parse(text=str))
}
AggData<-aggregate(DBFData[,c("follow1","follow2","follow3","follow4","follow5","follow6")], by=list(DBI
colnames(AggData)[1]<-"id"
AggData$n<-aggregate(DBFData$id,by=list(DBFData$id),FUN=length)$x
AggData$Treatment<-aggregate(DBFData$treatment,by=list(DBFData$id),FUN=mean)$x
saveRDS(AggData,"Data/SFEM/DBFData.rds")

priorMix<-c(1,1,1,1,1,1)

##############################
# Homogeneous trembles models
##############################

model<-stan_model("Code/SFEM/SFEM.stan")

Fits<-c()

for (tt in 1:6) {
  print(paste("estimating homogeneous trembles model",tt))
  Dt<- AggData %>% filter(Treatment==tt)
  d = list(
```

```
    FOLLOWS=Dt[,2:7],
    N=dim(Dt)[1],
    S=6,
    COUNTS=kronecker(Dt[,"n"],matrix(1,1,6)),
    priorMix=priorMix)
  Fit<-sampling(model,data=d,seed=42,control = list(adapt_delta = 0.8))
  S<-summary(Fit,pars=c("mix","trmb"))$summary[,c("mean","sd")] |>
    data.frame() |>
    mutate(msd = paste0(mean |> round(rounding)," (",sd |> round(rounding),")"))
  Fits<-cbind(Fits,S$msd)
}

Fits |> saveRDS("Code/SFEM/SFEMHomogeneousTremblesEstimates.rds")

##############################
# Heterogeneous trembles models
##############################


model<-stan_model("Code/SFEM/SFEMBeta.stan")

Fits<-c()

for (tt in 1:6) {
  print(paste("estimating heterogeneous trembles model",tt))
  Dt<- AggData %>% filter(Treatment==tt)
  d = list(
    FOLLOWS=Dt[,2:7],
    N=dim(Dt)[1],
    S=6,
    COUNTS=kronecker(Dt[,"n"],matrix(1,1,6)),
    priorMix=priorMix,
    prior_k = 1)
  Fit<-sampling(model,data=d,seed=42,control = list(adapt_delta = 0.8))
  S<-summary(Fit,pars=c("mix","q","k"))$summary[,c("mean","sd")] |>
    data.frame() |>
    mutate(msd = paste0(mean |> round(rounding)," (",sd |> round(rounding),")"))
  Fits<-cbind(Fits,S$msd)
}

Fits |> saveRDS("Code/SFEM/SFEMHeterogeneousTremblesEstimates.rds")
```

12 Application: Strategy frequency estimation with a mixed strategy

In the previous chapter, we learned about the strategy frequency estimation method (SFEM). One omission in this chapter was the possibility for including mixed or behavior strategies in the estimation. That is, the previous chapter only coverered strategies that were *pure* strategies. While the extension to mixing is not difficult, it is worth some attention. In the example below, we are interested in the importance of an equilibrium mixed strategy, compared to some other plausible candidate strategies that players could be using.

Table 24: The four strategies considered in @Anwar2024. $\gamma = 0.528$. c_0 (c_1, c_2) indicates the information set where 0 (1, 2) previous players have chosen to invest.

	Information set					
	Position 1	Position 2		Position 3 & 4		
Strategy	c_0	c_0	c_1	c_0	c_1	c_2
G&M, T1	1	0	1	0	0	1
G&M, T2	1	γ	1	γ	1	
G&M, T3	0	0	0	0	0	0
Free rider	0	0	0	0	0	0
Altruist	1	1	1	1	1	1
Conditional co-operator, T1 & T3	1	0	1	0	1	1
Conditional co-operator, T2	1	0	1	0	1	

12.1 Example dataset and strategies

Anwar and Georgalos (2024) study a sequential 4-player public goods game with poisition uncertainty. That is, participants in this game played sequentially, but did not know exactly the position out of four that they would take their turn. Each player could invest or not invest a fixed amount in a public good (i.e. investment was a binary decision, not continuous). Players receive partial information about the investments of players before them. Specifically,

1. In Treatment 1 players received information about the investment choices of the two players who moved before them. Players did not know their position in the game in this treatment, but could infer whether they were in Position 1 or 2 if they receivec feedback about fewer than 2 opponents.
2. In Treatment 2 players received information about the investment choice of the one player before them. Players did not know their position in the game, but could infer that they were in Position 1 because they would not have received any feedback about opponents' cholices, and
3. Treatment 3 was the same as Treatment 1, except that players knew their position in the game.

In their SFEM, Anwar and Georgalos (2024) consider four strategies shown in Table 24. The "Free rider" and "Altruist" strategeis do not depend on the treatment, and just involve either always not cooperating or cooperating, respectively. The two other strategies, "G&M" and "Conditional cooperator" depend on the treatment. Here the "G&M" strategy is the equilibrium strategy characterized by Gallice and Monzón (2019) for this game. Importantly for this application, in Treatment 2 the strategy invests with probability $\gamma = 0.528$ after some information sets. It is this strategy that will add the novel part to this SFEM compared to the previous chapter.

```
TAB<-rbind(
  c("G&M, T1", 1,0,1,0,0,1),
  c("G&M, T2", 1,"$\\gamma$",1 ,"$\\gamma$",1," "),
  c("G&M, T3", 0, 0, 0, 0, 0, 0),
  c("Free rider",0,0,0,0,0,0),
  c("Altruist",1,1,1,1,1,1),
  c("Conditional co-operator, T1 & T3",1,0,1,0,1,1),
  c("Conditional co-operator, T2",1, 0, 1, 0, 1, " ")
)

TAB |>
  kbl(caption = "The four strategies considered in @Anwar2024. $\\gamma=0.528$. $c_0$ ($c_1$, $c_2$) in
  kable_classic(full_width=FALSE) |>
  add_header_above(c("Strategy","$c_0$","$c_0$","$c_1$","$c_0$","$c_1$","$c_2$")) |>
  add_header_above(c(" ","Position 1","Position 2"=2,"Position 3 & 4"=3)) |>
  add_header_above(c(" ","Information set" = 6))
```

12.2 The likelihood function

Note that we can write each strategy as a 6-vector, where each element represents the probability of playing "invest" in each information set. That is, the G&M strategy for Treatment 1 can be represented by:

$$\sigma_s = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}^\top$$

and the G&M strategy for Treatment 2 can be represented by:

$$\sigma_s = \begin{pmatrix} 1 & 0.528 & 1 & 0.528 & 1 & 0.5 \end{pmatrix}$$

Here since the "Position 3 & 4, c_2" information set is never reached, we can in principle put anything into this element of the vector. However since we will be taking the log of these vectors to calculate the log-likelihood, 0.5 will mean we don't end up taking the log of a negative number.

Since there are still some (nearly all) pure strategies on the list, we still need the tremble probability $\epsilon \in (0, 0.5)$ for these pure strategies to make sure that the likelihood function is not zero everywhere. Therefore we can define the strategy inclusive of trembles as the probability a player chooses "invest" after each of the six information sets:

$$\tilde{\sigma}_s = (1 - \epsilon)\sigma_s + \epsilon(1 - \sigma_s)$$

The only bits we need to worry about is for the second treatment, where the G&M strategy is mixed. Here we just replace the relevant elements of $\tilde{\sigma}_s$ with γ.[55] That is, the G&M strategy in Treatment 2 becomes:

$$\tilde{\sigma}_{\text{GM, T2}} = \begin{pmatrix} 1-\epsilon & 0.528 & 1-\epsilon & 0.528 & 1-\epsilon & 0.5 \end{pmatrix}^\top$$

Now we can construct the log-likelihood of following a particular strategy σ_s for a participant i:

$$\log p(y_i, n_i \mid \sigma_s, \epsilon) = y_i^\top \log(\tilde{\sigma}_s) + (n_i - y_i)^\top \log(1 - \tilde{\sigma}_s)$$

Where y_i is a count of the number of times participant i chose "invest" after each information set, and n_i is a count of the number of times participant i reached each information set.

The rest proceeds exactly as the SFEM would in the previous chapter. We next take the mixing probabilities $\psi \in \Delta^4$ and integrate out the types:

$$\log p(y_i, n_i \mid \psi, \epsilon) = \log \left(\sum_{s=1}^{4} \exp \left(\log \psi_s + \log p(y_i, n_i \mid \sigma_s, \epsilon) \right) \right)$$

Note, as we did in the previous chapter, that we are writing this in a form that means we can use *Stan*'s `log_sum_exp()` function.

12.3 Implementation in *Stan*

The parameters in the model are the strategy frequencies $\psi \in \Delta^4$, and the tremble prbability $\epsilon \in (0, 0.5)$. I assign the following priors to these parameters:

$$\begin{aligned} \psi &\sim \text{Dirichlet}(1, 1, 1, 1) \\ \epsilon &\sim \text{TruncatedBeta}(1, 1, (0, 0.5)) \end{aligned}$$

[55] This is the same as the original implimentation in Anwar and Georgalos (2024). An alternative implementation could be to keep $\tilde{\sigma}_s$ as it is described above. In this case, the participant mixes *and* trembles.

I estimate five models that make different assumptions about how the strategies and trembles relate between treatments. Because all of these programs can use the same `data` and `transformed data` blocks, I wrote a single script called `SFEM_INCLUDE_data.stan` to include at the top of alll the *Stan* programs. Here is `SFEM_INCLUDE_data.stan`:

```
data {

  //
  real<lower=0,upper=1> gamma;

  int n_t1;
  matrix[n_t1,6] coop_t1;
  matrix[n_t1,6] count_t1;
  int n_t2;
  matrix[n_t2,6] coop_t2;
  matrix[n_t2,6] count_t2;
  int n_t3;
  matrix[n_t3,6] coop_t3;
  matrix[n_t3,6] count_t3;


  vector[4] prior_mix;
  vector[2] prior_eps;

}

transformed data {

  // description of strategies. See Table 3 in the paper

  row_vector[6] strg_t1[4];
  row_vector[6] strg_t2[4];
  row_vector[6] strg_t3[4];

  // G&M
  strg_t1[1] = [1,0,1,0,0,1];
  strg_t2[1] = [1,gamma,1,gamma,1,0.5];
  strg_t3[1] = rep_row_vector(0,6);

  // free rider
  strg_t1[2] = rep_row_vector(0,6);
  strg_t2[2] = rep_row_vector(0,6);
  strg_t3[2] = rep_row_vector(0,6);

  // altruist
  strg_t1[3] = rep_row_vector(1,6);
  strg_t2[3] = rep_row_vector(1,6);
  strg_t3[3] = rep_row_vector(1,6);

  // conditional cooperator
  strg_t1[4] = [1,0,1,0,1,1];
  strg_t2[4] = [1,0,1,0,1,0.5];
  strg_t3[4] = [1,0,1,0,1,1];
```

```
}
```

First, I estimate a model that allows the strategy frequencies ψ and the tremble probability ϵ to vary by treatment. This is probably the best comparison to the SFEM estimated in Anwar and Georgalos (2024) (see their Table 4):

```
#include SFEM_INCLUDE_data.stan

parameters {

  simplex[4] mix[3];

  vector<lower=0,upper=0.5>[3] eps;

}

model {

  // get a matrix of cooperation probabilities for each strategy in each treatment

  vector[6] pr_coop_t1[4];
  vector[6] pr_coop_t2[4];
  vector[6] pr_coop_t3[4];
  for (ss in 1:4) {



    pr_coop_t1[ss] = strg_t1[ss]'*(1-eps[1])+(1-strg_t1[ss])'*eps[1];
    pr_coop_t2[ss] = strg_t2[ss]'*(1-eps[2])+(1-strg_t2[ss])'*eps[2];
    pr_coop_t3[ss] = strg_t3[ss]'*(1-eps[3])+(1-strg_t3[ss])'*eps[3];

    // fix the strategies that are actually mixed.
    for (tt in 1:6) {

      pr_coop_t1[ss][tt] = abs(strg_t1[ss][tt]-0.5)<0.4 ? gamma : pr_coop_t1[ss][tt];
      pr_coop_t2[ss][tt] = abs(strg_t2[ss][tt]-0.5)<0.4 ? gamma : pr_coop_t2[ss][tt];
      pr_coop_t3[ss][tt] = abs(strg_t3[ss][tt]-0.5)<0.4 ? gamma : pr_coop_t3[ss][tt];
    }

  }

  matrix[n_t1,4] like_t1 = rep_matrix(log(mix[1]'),n_t1);
  matrix[n_t2,4] like_t2 = rep_matrix(log(mix[2]'),n_t2);
  matrix[n_t3,4] like_t3 = rep_matrix(log(mix[3]'),n_t3);

  for (ss in 1:4) {

    like_t1[,ss]+=coop_t1*log(pr_coop_t1[ss])+(count_t1-coop_t1)*log(1-pr_coop_t1[ss]);
    like_t2[,ss]+=coop_t2*log(pr_coop_t2[ss])+(count_t2-coop_t2)*log(1-pr_coop_t2[ss]);
    like_t3[,ss]+=coop_t3*log(pr_coop_t3[ss])+(count_t3-coop_t3)*log(1-pr_coop_t3[ss]);

  }

  for (ii in 1:n_t1) {
```

```
    target += log_sum_exp(like_t1[ii,]);
  }
  for (ii in 1:n_t2) {
    target += log_sum_exp(like_t2[ii,]);
  }
  for (ii in 1:n_t3) {
    target += log_sum_exp(like_t3[ii,]);
  }

  for (tt in 1:3) {

    target += dirichlet_lpdf(mix[tt]|prior_mix);
    target += beta_lpdf(eps[tt]| prior_eps[1],prior_eps[2]);

  }

}
```

However what I was interested in (and this is not necessarily what Anwar and Georgalos (2024) were interested in) was how stable the strategy frequencies were across treatments, so I estimated a model that assumed that the strategy frequencies were *the same* across the three treatments, but allowed the tremble probability to vary across treatments:

```
#include SFEM_INCLUDE_data.stan

parameters {

  simplex[4] mix;

  vector<lower=0,upper=0.5>[3] eps;

}

model {

  // get a matrix of cooperation probabilities for each strategy in each treatment

  vector[6] pr_coop_t1[4];
  vector[6] pr_coop_t2[4];
  vector[6] pr_coop_t3[4];
  for (ss in 1:4) {

    pr_coop_t1[ss] = strg_t1[ss]'*(1-eps[1])+(1-strg_t1[ss])'*eps[1];
    pr_coop_t2[ss] = strg_t2[ss]'*(1-eps[2])+(1-strg_t2[ss])'*eps[2];
    pr_coop_t3[ss] = strg_t3[ss]'*(1-eps[3])+(1-strg_t3[ss])'*eps[3];

    // fix the strategies that are actually mixed.
    for (tt in 1:6) {

      pr_coop_t1[ss][tt] = abs(strg_t1[ss][tt]-0.5)<0.4 ? gamma : pr_coop_t1[ss][tt];
      pr_coop_t2[ss][tt] = abs(strg_t2[ss][tt]-0.5)<0.4 ? gamma : pr_coop_t2[ss][tt];
      pr_coop_t3[ss][tt] = abs(strg_t3[ss][tt]-0.5)<0.4 ? gamma : pr_coop_t3[ss][tt];
    }
```

```
  }

  matrix[n_t1,4] like_t1 = rep_matrix(log(mix'),n_t1);
  matrix[n_t2,4] like_t2 = rep_matrix(log(mix'),n_t2);
  matrix[n_t3,4] like_t3 = rep_matrix(log(mix'),n_t3);

  for (ss in 1:4) {

    like_t1[,ss]+=coop_t1*log(pr_coop_t1[ss])+(count_t1-coop_t1)*log(1-pr_coop_t1[ss]);
    like_t2[,ss]+=coop_t2*log(pr_coop_t2[ss])+(count_t2-coop_t2)*log(1-pr_coop_t2[ss]);
    like_t3[,ss]+=coop_t3*log(pr_coop_t3[ss])+(count_t3-coop_t3)*log(1-pr_coop_t3[ss]);

  }

  for (ii in 1:n_t1) {
    target += log_sum_exp(like_t1[ii,]);
  }
  for (ii in 1:n_t2) {
    target += log_sum_exp(like_t2[ii,]);
  }
  for (ii in 1:n_t3) {
    target += log_sum_exp(like_t3[ii,]);
  }

  for (tt in 1:3) {


    target += beta_lpdf(eps[tt]| prior_eps[1],prior_eps[2]);

  }

  target += dirichlet_lpdf(mix|prior_mix);

}
```

For good measure, I also estimated a model that imposed that the tremble probabilities were constant across the treatments:

```
#include SFEM_INCLUDE_data.stan

parameters {

  simplex[4] mix;

  vector<lower=0,upper=0.5>[3] eps;

}

model {

  // get a matrix of cooperation probabilities for each strategy in each treatment

  vector[6] pr_coop_t1[4];
  vector[6] pr_coop_t2[4];
```

```
  vector[6] pr_coop_t3[4];
  for (ss in 1:4) {

    pr_coop_t1[ss] = strg_t1[ss]'*(1-eps[1])+(1-strg_t1[ss])'*eps[1];
    pr_coop_t2[ss] = strg_t2[ss]'*(1-eps[2])+(1-strg_t2[ss])'*eps[2];
    pr_coop_t3[ss] = strg_t3[ss]'*(1-eps[3])+(1-strg_t3[ss])'*eps[3];

    // fix the strategies that are actually mixed.
    for (tt in 1:6) {

      pr_coop_t1[ss][tt] = abs(strg_t1[ss][tt]-0.5)<0.4 ? gamma : pr_coop_t1[ss][tt];
      pr_coop_t2[ss][tt] = abs(strg_t2[ss][tt]-0.5)<0.4 ? gamma : pr_coop_t2[ss][tt];
      pr_coop_t3[ss][tt] = abs(strg_t3[ss][tt]-0.5)<0.4 ? gamma : pr_coop_t3[ss][tt];
    }

  }

  matrix[n_t1,4] like_t1 = rep_matrix(log(mix'),n_t1);
  matrix[n_t2,4] like_t2 = rep_matrix(log(mix'),n_t2);
  matrix[n_t3,4] like_t3 = rep_matrix(log(mix'),n_t3);

  for (ss in 1:4) {

    like_t1[,ss]+=coop_t1*log(pr_coop_t1[ss])+(count_t1-coop_t1)*log(1-pr_coop_t1[ss]);
    like_t2[,ss]+=coop_t2*log(pr_coop_t2[ss])+(count_t2-coop_t2)*log(1-pr_coop_t2[ss]);
    like_t3[,ss]+=coop_t3*log(pr_coop_t3[ss])+(count_t3-coop_t3)*log(1-pr_coop_t3[ss]);

  }

  for (ii in 1:n_t1) {
    target += log_sum_exp(like_t1[ii,]);
  }
  for (ii in 1:n_t2) {
    target += log_sum_exp(like_t2[ii,]);
  }
  for (ii in 1:n_t3) {
    target += log_sum_exp(like_t3[ii,]);
  }

  for (tt in 1:3) {


    target += beta_lpdf(eps[tt]| prior_eps[1],prior_eps[2]);

  }

  target += dirichlet_lpdf(mix|prior_mix);

}
```

To test whether *all* players were using the G&M strategy, I estimated a SFEM with just the G&M strategy. Comparing this to the others would give us an idea of how important the free rider, altruist, and conditional co-operator stratgies were:

```
#include SFEM_INCLUDE_data.stan

parameters {

  vector<lower=0,upper=0.5>[3] eps;

}

model {

  // get a matrix of cooperation probabilities for each strategy in each treatment

  vector[6] pr_coop_t1[4];
  vector[6] pr_coop_t2[4];
  vector[6] pr_coop_t3[4];
  for (ss in 1:4) {

    pr_coop_t1[ss] = strg_t1[ss]'*(1-eps[1])+(1-strg_t1[ss])'*eps[1];
    pr_coop_t2[ss] = strg_t2[ss]'*(1-eps[2])+(1-strg_t2[ss])'*eps[2];
    pr_coop_t3[ss] = strg_t3[ss]'*(1-eps[3])+(1-strg_t3[ss])'*eps[3];

    // fix the strategies that are actually mixed.
    for (tt in 1:6) {

      pr_coop_t1[ss][tt] = abs(strg_t1[ss][tt]-0.5)<0.4 ? gamma : pr_coop_t1[ss][tt];
      pr_coop_t2[ss][tt] = abs(strg_t2[ss][tt]-0.5)<0.4 ? gamma : pr_coop_t2[ss][tt];
      pr_coop_t3[ss][tt] = abs(strg_t3[ss][tt]-0.5)<0.4 ? gamma : pr_coop_t3[ss][tt];
    }

  }

  matrix[n_t1,1] like_t1 = rep_matrix(0,n_t1,1);
  matrix[n_t2,1] like_t2 = rep_matrix(0,n_t2,1);
  matrix[n_t3,1] like_t3 = rep_matrix(0,n_t3,1);

  for (ss in 1:1) {

    like_t1[,ss]+=coop_t1*log(pr_coop_t1[ss])+(count_t1-coop_t1)*log(1-pr_coop_t1[ss]);
    like_t2[,ss]+=coop_t2*log(pr_coop_t2[ss])+(count_t2-coop_t2)*log(1-pr_coop_t2[ss]);
    like_t3[,ss]+=coop_t3*log(pr_coop_t3[ss])+(count_t3-coop_t3)*log(1-pr_coop_t3[ss]);

  }

  for (ii in 1:n_t1) {
    target += like_t1[ii,1];
  }
  for (ii in 1:n_t2) {
    target += like_t2[ii,1];
  }
  for (ii in 1:n_t3) {
    target += like_t3[ii,1];
  }
```

```
  for (tt in 1:3) {


    target += beta_lpdf(eps[tt]| prior_eps[1],prior_eps[2]);

  }


}
```

Finally, to test the importance of the G&M strategy, I estimated a model without this strategy. That is, if this strategy is *not* important, then eliminating it from the list should not matter too much:

```
#include SFEM_INCLUDE_data.stan

parameters {

  simplex[3] mix[3];

  vector<lower=0,upper=0.5>[3] eps;

}

model {

  // get a matrix of cooperation probabilities for each strategy in each treatment

  vector[6] pr_coop_t1[3];
  vector[6] pr_coop_t2[3];
  vector[6] pr_coop_t3[3];
  for (ss in 1:3) {

    pr_coop_t1[ss] = strg_t1[ss+1]'*(1-eps[1])+(1-strg_t1[ss+1])'*eps[1];
    pr_coop_t2[ss] = strg_t2[ss+1]'*(1-eps[2])+(1-strg_t2[ss+1])'*eps[2];
    pr_coop_t3[ss] = strg_t3[ss+1]'*(1-eps[3])+(1-strg_t3[ss+1])'*eps[3];

    // fix the strategies that are actually mixed.
    for (tt in 1:6) {

      pr_coop_t1[ss][tt] = abs(strg_t1[ss][tt]-0.5)<0.4 ? gamma : pr_coop_t1[ss][tt];
      pr_coop_t2[ss][tt] = abs(strg_t2[ss][tt]-0.5)<0.4 ? gamma : pr_coop_t2[ss][tt];
      pr_coop_t3[ss][tt] = abs(strg_t3[ss][tt]-0.5)<0.4 ? gamma : pr_coop_t3[ss][tt];
    }

  }

  matrix[n_t1,3] like_t1 = rep_matrix(log(mix[1]'),n_t1);
  matrix[n_t2,3] like_t2 = rep_matrix(log(mix[2]'),n_t2);
  matrix[n_t3,3] like_t3 = rep_matrix(log(mix[3]'),n_t3);

  for (ss in 1:3) {

    like_t1[,ss]+=coop_t1*log(pr_coop_t1[ss])+(count_t1-coop_t1)*log(1-pr_coop_t1[ss]);
    like_t2[,ss]+=coop_t2*log(pr_coop_t2[ss])+(count_t2-coop_t2)*log(1-pr_coop_t2[ss]);
```

```
    like_t3[,ss]+=coop_t3*log(pr_coop_t3[ss])+(count_t3-coop_t3)*log(1-pr_coop_t3[ss]);



  }

  for (ii in 1:n_t1) {
    target += log_sum_exp(like_t1[ii,]);
  }
  for (ii in 1:n_t2) {
    target += log_sum_exp(like_t2[ii,]);
  }
  for (ii in 1:n_t3) {
    target += log_sum_exp(like_t3[ii,]);
  }

  for (tt in 1:3) {

    target += dirichlet_lpdf(mix[tt]|prior_mix[1:3]);
    target += beta_lpdf(eps[tt]| prior_eps[1],prior_eps[2]);

  }

}
```

12.4 Results

Table 25 shows the posterior estimates from the most general model, where strategy frequencies and tremble probabilities can vary by treatment. Here we can see that the G&M strategy accounts for a substantial fraction of decisions in all three treatments. For perspective with the Dirichlet prior, the prior stanard deviation for the mixing probabilities is about 0.19, which is quite a bit larget than the posterior standard deviations: we have learned substantially from the data.

```
fmt<-"%.3f"

strgList<-c("G&M","Free rider","Altruist","Conditional cooperator")

Fit<-summary("Code/Anwar2024/Fit_SFEM.rds" |> readRDS())$summary |>
  data.frame() |>
  rownames_to_column(var = "par") |>
  filter(par!="lp__") |>
  mutate(
    parameter = ifelse(grepl("mix",par),"mix","epsilon")
  ) |>
  mutate(
    treatment = ifelse(parameter=="mix",
                       str_split_i(par,",",1) |> parse_number(),
                       par |> parse_number()
                       ),
    strg = str_split_i(par,",",2) |> parse_number() ,
    strategy = ifelse(parameter=="mix",strgList[strg],"$\\epsilon$")
  )

means<-Fit |>
```

```
  mutate(
    mean = sprintf(fmt,mean)
  ) |>
  pivot_wider(
    id_cols = "strategy",
    names_from = "treatment",
    values_from = "mean"
  )

sds<-Fit |>
  mutate(
    sd = paste0("(",sprintf(fmt,sd),")")
  ) |>
  pivot_wider(
    id_cols = "strategy",
    names_from = "treatment",
    values_from = "sd"
  ) |>
  mutate(
    strategy = ""
  )

TAB<-tibble()

for (rr in 1:dim(means)[1]) {

  TAB<-rbind(TAB,
             means[rr,],
             sds[rr,],
             rep("  ",4)
             )
}

TAB |>
  kbl(caption = "Strategy frequencies and tremble probabilities from the most general model allowing fo
  kable_classic(full_width=FALSE) |>
  add_header_above(c("","Treatment"=3))
```

In Table 26 I show the five models' posterior probabilities assuming equal prior probabilities. Here we can see most starkly that the final two models do *not* organize the data well relative to the others. Specifically (i) the G&M strategy on its own does not do well, and (ii) eliminating the G&M strategy only does not do well. Of the remaining models, the most general does the best, but the support for this unrestricted model is not overwhelming. It seems that assuming that strategy frequencies are common across all thre treatments is not too terrible of a restriction.

```
postprobs<-"Code/Anwar2024/postprobs.rds"|> readRDS()

TAB<-tibble(
  model = c("Unrestricted","Pooled frequencies","Pooled frequencies and trembles","G&M strategy only","
  `posterior probability` = postprobs
)

TAB |>
  kbl(digits=3,caption = "Model posterior probabilities assuming equal prior probabilities. ") |>
```

Table 25: Strategy frequencies and tremble probabilities from the most general model allowing for strategy frequencies and tremble probabilities to vary by treatment.

	Treatment		
strategy	1	2	3
G&M	0.186	0.252	0.142
	(0.069)	(0.091)	(0.093)
Free rider	0.083	0.112	0.144
	(0.045)	(0.051)	(0.092)
Altruist	0.312	0.392	0.097
	(0.079)	(0.091)	(0.056)
Conditional cooperator	0.419	0.244	0.616
	(0.088)	(0.078)	(0.087)
ϵ	0.136	0.109	0.182
	(0.013)	(0.015)	(0.014)

Table 26: Model posterior probabilities assuming equal prior probabilities.

model	posterior probability
Unrestricted	0.548
Pooled frequencies	0.204
Pooled frequencies and trembles	0.247
G&M strategy only	0.000
Excluding G&M strategy	0.000

```
  kable_classic(full_width=FALSE)
```

12.5 *R* code used to estimate the models

```
library(tidyverse)
library(rstan)
options(mc.cores = parallel::detectCores())
rstan_options(auto_write = TRUE)
library(bridgesampling)

#------------------------------------------------------------------------------
# Here I am following "SFEM.R" in the replication files to wrangle the data

data<-"Data/AG2024PositionUncertainty.csv" |>
  read.csv()
dt2=data[data$treatment==2,]
dt3=data[data$treatment==3,]
dt4=data[data$treatment==4,]

t2=c()
#Order the data per treatment, player, round and position.
```

```
for (round in 1:10){
  position=dt2[,which(colnames(dt2)==paste("pg_T2.",round,".player.id_in_group",sep=""))]
  c0p1=dt2[,which(colnames(dt2)==paste("pg_T2.",round,".player.response_0p1",sep=""))]
  c0p2=dt2[,which(colnames(dt2)==paste("pg_T2.",round,".player.response_0p2",sep=""))]
  c0p34=dt2[,which(colnames(dt2)==paste("pg_T2.",round,".player.response_0p34",sep=""))]
  c1p2=dt2[,which(colnames(dt2)==paste("pg_T2.",round,".player.response_1p2",sep=""))]
  c1p34=dt2[,which(colnames(dt2)==paste("pg_T2.",round,".player.response_1p34",sep=""))]
  c2p34=dt2[,which(colnames(dt2)==paste("pg_T2.",round,".player.response_2p34",sep=""))]


  t2=rbind(t2,cbind(sbj=dt2$sbj,treatment=rep(2,32),position,rnd=rep(round,32),c0p1,c0p2,c0p34,c1p2,c1p3
}



t3=c()
#Order the data per treatment, player, round and position.
for (round in 1:10){
  position=dt3[,which(colnames(dt3)==paste("pg_T3.",round,".player.id_in_group",sep=""))]
  c0p1=dt3[,which(colnames(dt3)==paste("pg_T3.",round,".player.response_0p1",sep=""))]
  c0p2=dt3[,which(colnames(dt3)==paste("pg_T3.",round,".player.response_0p2",sep=""))]
  c0p34=dt3[,which(colnames(dt3)==paste("pg_T3.",round,".player.response_0p34",sep=""))]
  c1p2=dt3[,which(colnames(dt3)==paste("pg_T3.",round,".player.response_1p2",sep=""))]
  c1p34=dt3[,which(colnames(dt3)==paste("pg_T3.",round,".player.response_1p34",sep=""))]

  t3=rbind(t3,cbind(sbj=dt3$sbj,treatment=rep(3,32),position,rnd=rep(round,32),c0p1,c0p2,c0p34,c1p2,c1p3
}



t4=c()
#Order the data per treatment, player, round and position.
for (round in 1:10){
  position=dt4[,which(colnames(dt4)==paste("pg_T4.",round,".player.id_in_group",sep=""))]
  c0p1=dt4[,which(colnames(dt4)==paste("pg_T4.",round,".player.response_0p1",sep=""))]
  c0p2=dt4[,which(colnames(dt4)==paste("pg_T4.",round,".player.response_0p2",sep=""))]
  c0p34=dt4[,which(colnames(dt4)==paste("pg_T4.",round,".player.response_0p34",sep=""))]
  c1p2=dt4[,which(colnames(dt4)==paste("pg_T4.",round,".player.response_1p2",sep=""))]
  c1p34=dt4[,which(colnames(dt4)==paste("pg_T4.",round,".player.response_1p34",sep=""))]
  c2p34=dt4[,which(colnames(dt4)==paste("pg_T4.",round,".player.response_2p34",sep=""))]

  t4=rbind(t4,cbind(sbj=dt4$sbj,treatment=rep(4,32),position,rnd=rep(round,32),c0p1,c0p2,c0p34,c1p2,c1p3
}


t2=data.frame(t2)
t3=data.frame(t3)
t4=data.frame(t4)
sbj2=unique(t2$sbj)
sbj3=unique(t3$sbj)
sbj4=unique(t4$sbj)

#-------------------------------------------------------------------------------
```

```
# JB's code starts here
#--------------------------------------------------------------------------------

# Get data into long summarized form

D<-t2 |>
  pivot_longer(cols = c0p1:c2p34) |>
  rbind(
    t3 |>
      pivot_longer(cols = c0p1:c1p34)
  ) |>
  rbind(
    t4 |>
      pivot_longer(cols = c0p1:c2p34)
  ) |>
  mutate(
    coop = value==10
  ) |>
  filter(!is.na(coop)) |>
  select(-value) |>
  mutate(
    pos = str_split_i(name,"p",2),
    history_length = ifelse(pos=="1",0,nchar(pos)),
    history_coop = str_split_i(name,"p",1) |> parse_number()
  ) |>
  group_by(sbj,treatment,history_length,history_coop) |>
  summarize(
    coop_count = sum(coop),
    action_count = n()
  ) |>
  mutate(
    treatment = treatment-1,
    ps = paste0("p",history_length+1,"c",history_coop)
  )

dwide<- D |>
  pivot_wider(
    id_cols = c(sbj,treatment),
    names_from = ps,
    values_from = c(coop_count,action_count),
    values_fill = 0
  ) |>
  ungroup()

t1<-dwide |> filter(treatment==1)
coop_t1<-t1 |> select(contains("coop_count"))
count_t1<-t1 |> select(contains("action_count"))

t2<-dwide |> filter(treatment==2)
coop_t2<-t2 |> select(contains("coop_count"))
count_t2<-t2 |> select(contains("action_count"))

t3<-dwide |> filter(treatment==3)
```

```
coop_t3<-t3 |> select(contains("coop_count"))
count_t3<-t3 |> select(contains("action_count"))




#------------------------------------------------------------------------------

model<-"Code/Anwar2024/SFEM.stan" |>
  stan_model()

dStan<-list(

  gamma = 0.528,

  n_t1 = dim(coop_t1)[1],
  coop_t1 = coop_t1,
  count_t1 = count_t1,

  n_t2 = dim(coop_t2)[1],
  coop_t2 = coop_t2,
  count_t2 = count_t2,

  n_t3 = dim(coop_t3)[1],
  coop_t3 = coop_t3,
  count_t3 = count_t3,

  prior_mix = c(1,1,1,1),
  prior_eps = c(1,1)
)

Fit<-model |>
  sampling(data=dStan,seed=42)



model_pooledfreq<-"Code/Anwar2024/SFEM_pooledfrequencies.stan" |>
  stan_model()

Fit_pooledfreq<-model_pooledfreq |>
  sampling(data=dStan,seed=42)

model_pooledall<-"Code/Anwar2024/SFEM_pooledall.stan" |>
  stan_model()

Fit_pooledall<-model_pooledall |>
  sampling(data=dStan,seed=42)

model_strg1only<-"Code/Anwar2024/SFEM_strg1only.stan" |>
  stan_model()

Fit_strg1only<-model_strg1only |>
  sampling(data=dStan,seed=42)
```

```
model_notGM<-"Code/Anwar2024/SFEM_notGM.stan" |>
  stan_model()

Fit_notGM<-model_notGM |>
  sampling(data=dStan,seed=42)

bs_unrestricted<-Fit |> bridge_sampler()
bs_pooledfreq<-Fit_pooledfreq |> bridge_sampler()
bs_pooledall<-Fit_pooledall |> bridge_sampler()
bs_strg1only<-Fit_strg1only |> bridge_sampler()
bs_notGM<-Fit_notGM |> bridge_sampler()

Fit |>
  saveRDS("Code/Anwar2024/Fit_SFEM.rds")


(postprobs<-post_prob(bs_unrestricted,bs_pooledfreq,bs_pooledall,bs_strg1only,bs_notGM))

postprobs |>
  saveRDS("Code/Anwar2024/postprobs.rds")
```

13 Computing Quantal Response Equilibrium

Quantal response equilibrium (McKelvey and Palfrey 1995) is an extension of Nash equilibrium that replaces the concept of a best response with a *probabilistic* best response. That is, instead of choosing an action that is in their best response correspondence, players assign positive probability to all of their actions in a game, and are more likely to take actions that yield greater utility. This extension offers some big advantages over Nash equilibrium when structurally analyzing data from economic experiments. Firstly, the idea of a *probabilistic* best response is *behaviorally plausible*. Just as we assume a probabilistic choice rule (like softmax) for individual choice experiments, the same reasons for using this in a game apply: people make mistakes, our model is probably mis-specified, and so on. Secondly, also for the same reasons we use them in individual choice experiments, the probabilistic choice rule permits us to use a likelihood. Otherwise one decision that goes against the prescription of the deterministic component of our model will make the likelihood zero everywhere, and so we cannot simulate a posterior. For games, this is especially a problem when players have a strictly dominated strategy, as these should never be played in equilibrium. Finally, Quantal Response Equilibrium has been shown to organize experimental data a lot better than Nash equilibrium. For example, it models well the "own payoff" effect observed in some games, where Nash equilibrium does not.

Compared to typical structural models for *individual* choice experiments, the challenges of estimating a Quantal Response Equilibrium model are somewhat unique. Individual choice experiments are sometimes difficult to estimate because it is difficult for our sampler to explore the posterior distribution implied by our likelihood and prior.[56] While this problem is by no means ruled out for Quantal Response Equilibrium, its typical computational hurdles come from computing the fixed point condition associated with the equilibrium. I will therefore devote the entirety of this chapter to feasible methods for computing quantal response equilibrium, as this can really make the difference between a model being feasible to estimate or not. As such, this chapter will be more of a *computational* chapter and less of an *estimation* chapter. Fear not, I will follow up with a Quantal Response Equilibrium estimation chapter later.

13.1 Overview of quantal response equilibrium

Quantal response equilibrium assumes that players *probabilistically* best respond to their opponents' mixed strategies. We model this with a *probabilistic best response function*, which I will denote as $q(\lambda, u_i(\sigma))$, where

[56] Loosely speaking, this is a similar problem to it being hard to maximize the likelihood in these applications.

$\lambda > 0$ is a choice precision parameter, and $u_{i,a}(\sigma)$ is the expected utility to player i of taking action a, given that they are facing mixed strategy profile σ. Holding λ constant, q_i takes a vector of expected utilities $u_i(\sigma)$, and returns a probability distribution over actions in the player's choice set, which itself is a vector of probabilities the same size as $u_i(\sigma)$. Some popular choices of the functional form of q include the logit or softmax specification:

$$q_{i,a}(\lambda, u_i(\sigma)) = \frac{\exp(\lambda u_{i,a}(\sigma))}{\sum_{b \in A_i} \exp(\lambda u_{i,b}(\sigma))}$$

and the Luce specification:

$$q_{i,a}(\lambda, u_i(\sigma)) = \frac{u_{i,a}(\sigma)^\lambda}{\sum_{b \in A_i} u_{i,b}(\sigma)^\lambda}$$

Note that both of these functions approach best response functions as $\lambda \to \infty$. At the other end of the parameter space, $\lambda = 0$, players randomize uniformly over their action space.

The "equilibrium" part of Quantal Response Equilibrium comes from assuming that players have correct beliefs about their opponents' mixed strategies. Mathematically then, the equilibrium condition is:

$$\sigma_i = q_i(\lambda, u_i(\sigma)) \quad \forall i \in \{1, 2, \ldots, n\}$$

That is, every equilibrium mixed strategy σ_i is a probabilistic best response to the equilibrium mixed strategy profile σ.

13.2 Computing Quantal Response Equilibrium

Finding a solution to (13.1) can be difficult because it is a fixed point condition: σ appears on both the right- and left-hand side of the equation, and we typically cannot find a closed-form solution. Furthermore, we cannot be guaranteed that it is a contraction, so iterating the equation by first guessing a σ, then computing the probabilistic best response profile, and then repeating, will not in general find a solution. A far more reliable way to compute Quantal Response Equilibrium is to use a *predictor-corrector* algorithm. This algorithm in the context of Quantal Response Equilibrium was developed in Turocy (2005) and Turocy (2010), and is discussed in detail in Bland and Turocy (2025). This algorithm is also implemented in the software package *Gambit* (McKelvey, McLennan, and Turocy 2022).

13.2.1 Setting up the problem

While it might be more intuitive to think about solving (13.1) exactly as it is written in probability *levels*, you will likely run into some stability issues if you do. This is because for some games the algorithm will want to jump into "probabilities" that are either less than zero or greater than one. In practice, the problem is much more stable if we first translate (13.1) into *log*-probability *differences*. That is, we re-write this equation as:

$$0 = H_{i,a}(\lambda, \sigma) = \log \sigma_{i,a} - \log \sigma_{i,a+1} - (\log q_{i,a}(\lambda, u_i(\sigma)) - \log q_{i,a+1}(\lambda, u_i(\sigma)))$$
$$\forall i \in \{1, 2, \ldots, n\},\ \forall a \in \{1, 2, \ldots, J_i - 1\}$$

where n is the number of players and J_i is the number of actions in player i's action space.

This representation is particularly useful, because the log-difference of the probabilistic best responses (the term in the parentheses) for the logit and Luce specifications become:

$$\begin{aligned}
\text{logit:} \quad & H_{i,a}(\lambda, \sigma) = \log \sigma_{i,a} - \log \sigma_{i,a+1} - \lambda \left[u_{i,a}(\sigma) - u_{i,a+1}(\sigma)\right] \\
\text{Luce:} \quad & H_{i,a}(\lambda, \sigma) = \log \sigma_{i,a} - \log \sigma_{i,a+1} - \lambda \left[\log u_{i,a}(\sigma) - \log u_{i,a+1}(\sigma)\right]
\end{aligned}$$

respectively. That is, we can cancel out the common denominators.

We then need some adding-up constraints to ensure that the mixed strategies add up to one:

$$0 = H_{i,\Sigma}(\lambda, \sigma) = \sum_{a=1}^{J_i} \sigma_{i,a} - 1, \quad \forall i \in \{1, 2, \ldots, n\}$$

Letting $\rho_{i,a} = \log \sigma_{i,a}$, these constraints become:

$$\begin{aligned}
\text{logit:} \quad & H_{i,a}(\lambda, \rho) = \rho_{i,a} - \rho_{i,a+1} - \lambda \left[u_{i,a}(\exp(\rho)) - u_{i,a+1}(\exp(\rho)) \right] \\
\text{Luce:} \quad & H_{i,a}(\lambda, \rho) = \rho_{i,a} - \rho_{i,a+1} - \lambda \left[\log u_{i,a}(\exp(\rho)) - \log u_{i,a+1}(\exp(\rho)) \right] \\
\text{adding up:} \quad & H_{i,\Sigma}(\lambda, \rho) = \sum_{a=1}^{J_i} \exp(\rho_{i,a}) - 1
\end{aligned}$$

where $\exp(\rho) = \sigma$ is the element-wise exponentiation of ρ.

Stacking all of these together, we have one great big vector of zero conditions defining a quantal response equilibrium:

$$0 = H(\lambda, \rho) = \begin{pmatrix} H_{1,1} \\ H_{1,2} \\ \vdots \\ H_{1,J_1-1} \\ H_{2,1} \\ \vdots \\ H_{2,J_2-1} \\ \vdots \\ H_{n,J_n-1} \\ H_{1,\Sigma} \\ H_{2,\Sigma} \\ \vdots \\ H_{n,\Sigma} \end{pmatrix}$$

13.2.2 A predictor-corrector algorithm

A good way to solve for quantal response equilibrium is to use a predictor corrector algorithm. This algorithm, developed in Turocy (2005) and Turocy (2010), initializes with a known quantal response equilibrium (λ^t, ρ^t), then uses information contained in H find a nearby solution $(\lambda^{t+1}, \rho^{t+1})$.

To begin with, we will re-parameterize the problem using $x \equiv \begin{pmatrix} \rho^\top & \lambda \end{pmatrix}^\top$. This is because the algorithm does not really make a distinction between the parameters ρ and λ.

A quantal response equilibrium (λ, ρ) can therefore be characterized as a solution to:

$$H(x) = 0$$

where $H(x)$ is a $\sum_{i=1}^{n} J_i \times 1$ vector. Furthermore, since the solutions of $H(x) = 0$ form a smooth, differentiable curve in the $\rho - \lambda$ space, we can trace out this curve using information contained in H. Specifically, we will use the *jacobian* of H to make a first-order approximation of the curve. We will then use the zero condition to correct any errors that come from this approximation.

Following the initialization, the algorithm applies two distinct steps:

First, in the *predictor* step, the algorithm makes a first-order approximation of the path continues from point x^t to point x^{t+1}. This approximation is found using the total derivative of H:

$$\frac{\partial H(x)}{\partial x^\top}\Delta x \approx 0$$

There are many solutions to this equation.[57] What we really need from this is a *direction* to move. To do this, we can take a "fat" QR decomposition of the transpose of the jacobian $\frac{\partial H(x)}{\partial x^\top}$. The final column of the Q that comes out of this gives us the direction we want, and has magnitude one. Therefore the predictor step is:

$$x^{t+1} = x^t + h\omega Q_{,-1}(x_t)$$

where:

- $Q_{,-1}(x_t)$ is the final column of the Q component of the "fat" QR decomposition of $\left[\frac{\partial H(x_t)}{\partial x^\top}\right]^\top$. We can compute the fat decomposition of matrix A in R using `qr(A) |> qr.Q(complete=TRUE)`.[58]
- $\omega \in \{-1, 1\}$ ensures that we are moving in the right direction along the curve, and not back-tracking. For example, if we are looking to trace out the curve for increasing λ, then ω will have the same sign as the last element in $Q_{,-1}(x_t)$, which corresponds to λ.
- $h > 0$ is a tuning parameter specifying the step length. Recall that the magnitude of $Q_{,-1}(x_t)$ has a magnitude of 1, so h is the step length.

As noted in Turocy (2005), while it may be feasible to solve for quantal response equilibrium using only predictor steps, which are a form of numerical integration, this ignores some information that we have about quantal response equilibrium: it is characterized by a solution to a set of equations.[59] Therefore, we can also exploit these equations to correct any errors that may have made their way into the system equations because the predictor step is only an approximation of the change in x. Specifically, the following *corrector step* moves x in the direction that gets $H(x^{t+1})$ closer to zero:

$$x^{t+1} = x^t - \left(\frac{\partial H(x^t)}{\partial x^\top}\right)^+ H(x^t)$$

where A^+ is the Moore-Penrose inverse of A. We can compute this inverse in R using `ginv(A)`, which comes in the `MASS` library. We iterate on the corrector step until it has reached our desired level of accuracy.

13.2.3 Initial conditions

So once we have one solution to $H(\lambda, \rho) = 0$, we can trace out many other solutions. Great! But how do we initialize the problem? That is, we are doing this because we want to compute quantal response equilibrium, but we need to know one in order to start the algorithm. Fortunately, there are a couple of natural starting places. The simplest to start with is at the *centroid*: $\lambda = 0$, $\sigma_{i,a} = 1/J_i$ will always be a quantal response equilibrium. That is, when choice precision is zero, they payoffs of the game don't matter, and uniform randomization is the equilibrium strategy profile. The path $\{(\lambda, \rho) : H(\lambda, \rho) = 0, \lambda \geq 0\}$ starting at the centroid and $\lambda = 0$ is called the "principal branch", and is often used to select an equilibrium when multiple are present.

Other places to consider starting might be ones close to a Nash equilibrium. This is because quantal response equilibrium will nest every Nash equilibrium when $\lambda \to \infty$. Therefore starting with $\lambda^0 =$ something very large, and $\rho^0 = \log \sigma^{\text{Nash}}$, then tracing out the branch from there might also be useful. Of course, computing Nash equilibrium can be difficult, too, so this starting point may not be computationally feasible in some games.

[57] For example, if Δx is a solution to this equations, $c\Delta x$ for $c \in \mathbb{R}$ is also a solution.

[58] This was a major time suck for me in converting Ted Turocy's *Python* code into R, because the default options in each language are different. By default, *Python* computes the fat decomposition, but R does not.

[59] That is, the predictor steps are equivalent if we are solving $H(x) = 0$, or $H(x) =$ any other vector of constants. In the predictor step, we explicitly take advantage of this being a zero condition.

13.2.4 Algorithm tuning

The predictor-corrector algorithm described above allows us to choose two tuning parameters in order to (hopefully) improve the algorithm's performance.

1. For the *predictor step*, we can choose the step size h, which pins down how large is $\|x^{t+1} - x^t\|$
2. For the *corrector step*, we can choose the stopping rule, which defines how accurate is "accurate enough" for our solution to $H(0) = 0$.

A good solution to this is to choose a desired level of accuracy for the corrector step, and then use the number of iterations needed in the corrector step to adjust the predictor step's step length h. Broadly speaking, if the corrector step requires a lot of iteration, then the previous predictor step jumped too far, and so we should reduce h for the next step. On the other hand if only a few iterations are needed for the corrector step, then we could have jumped further using the predictor step without too much loss of accuracy, so increasing h in this case is warranted.

13.3 The predictor-corrector algorithm in *R*

Here is my implementation of the predictor-corrector algorithm in *R*. It is unashamedly translated from Ted Turocy's *Python* code, which accompanies our paper Bland and Turocy (2025).

```
# Needed for the Moore-Penrose inverse
library(MASS)

PC<-function(
             # A named list comprising of functions H, Hlambda, and Hrho
             Hlist,
             # Total number of actions in the game
             nActions,
             # range of lambda to compute QRE for. First element is the starting point. Second element
             lambda_span, # c(lambda0,lambda_end)
             # Starting poring for log probabilities
             x0, # rho0
             # first step size
             first_step,
             # Minimum step size
             min_step,
             # Maximum deceleration of step size
             max_decel,
             # Maimum number of iterations for corrector step
             maxiter,
             # Tolerance for corrector step (a step size)
             tol
) {


  # unpack the list of functions Hlist

  H<-Hlist[["H"]]
  Hlambda<-Hlist[["Hlambda"]]
  Hrho<-Hlist[["Hrho"]]

  jac<-function(x) {
    cbind(Hrho(x[nActions+1],x[1:nActions]),Hlambda(x[nActions+1],x[1:nActions]))
  }
  # Maximum distance to curve
```

```
  max_dist <- 0.4
  # Maximum contraction rate in corrector
  max_contr <- 0.6
  # Perturbation to avoid cancellation in calculating contraction rate
  eta <- 0.1

  # Initial conditions
  x<-c(x0,lambda_span[1])
  QRE<-rbind(c(),x)



  # The last column of Q in the QR decomposition is the tangent vector
  t <- (qr(jac(x) |> t())  |> qr.Q(complete=TRUE))[,nActions+1]

  h<-first_step

  # Set orientation of curve, so we are tracing into the interior of `t_span`
  omega<-sign(t[nActions+1])*sign(lambda_span[2]-lambda_span[1])


while ((x[nActions+1]>=min(lambda_span)) & (x[nActions+1]<=max(lambda_span))) {
accept <-TRUE

  if (abs(h)<= min_step) {
    # Stepsize below minimum tolerance; terminate.
    success <-FALSE
    print(paste("Stepsize",abs(h),"less than minimum",min_step))
    break
  }

  # Predictor step
  u<-x + h*omega*t
  q<-qr(jac(u) |> t())  |> qr.Q(complete=TRUE)

  disto <- 0
  decel <- 1/max_decel # deceleration factor

  for (it in 1:maxiter) {
    y<-H(u[nActions+1],u[1:nActions])


    # change in rho proposed by Newton step
    # This is the step we'd use if we kept lambda constant
    #drho <- -solve(Hrho(u[nActions+1],u[1:nActions]),y)
    drho<- - ginv(jac(u)) %*% y
    dist<-(drho^2) |> sqrt() |> sum()


    if (dist >= max_dist) {
      print(paste("Proposed distance",dist,"exceeds max_dist =",max_dist))
      accept<-FALSE
      break
```

```
    }

    decel<-max(c(decel,sqrt(dist/max_dist)*max_decel))

    if (it>1) {
      contr<- dist/(disto +tol*eta)
      if (contr > max_contr) {
        print(paste("Maximum contraction rate exceeded"))
        accept<-FALSE
        break
      }
      decel <-max(c(decel,sqrt(contr/max_contr)*max_decel))
    }
    if (dist < tol) {

      # Success! update and break out of iteration
      break
    }
    disto<-dist

    # if we have got to this point, then:
    #  1. The Newton step has not proposed a too-large change
    #  2. The contraction rate has not been exceeded
    #  3. The proposed Newton step is not so small that we have effectively
    #     found a solution
    # Therefore, we update rho

    u<-u+drho



    if (it==maxiter) {
      # We have run out of iterations. Terminate
      print(paste("Maximum number of iterations",maxiter,"reached"))
    }

  } ## END OF CORRECTOR STEP

    if (!accept) {
      # Step was not accepted; take a smaller step and try again
      h<-h/max_decel
    }

    # Standard steplength adaptation
    h<-abs(h/min(c(decel,max_decel)))

    # Update with outcome of successful PC step
    if (sum(t*q[,nActions+1])<0) {
      # The orientation of the curve as determined by the QR
      # decomposition has changed.
      #
      # The original Allgower-Georg QR decomposition implementation
      # ensures the sign of the tangent is stable; when it is stable
```

Table 27: A generalized matching pennies game

	L	R
U	4, 0	0, 1
D	0, 1	1, 0

```
          # the curve orientation switching is a sign of a bifurcation.
          omega<- -omega
          #print("sign of omega flipped")
        }



        x<-u
        t<-q[,nActions+1]

        #print(c(exp(x[1:nActions]),x[nActions+1]))


         QRE<-rbind(QRE,x |> as.vector())
         #print(c(exp(x[1:nActions]),x[nActions+1]) |> t())



    }
  QRE


}

# Save the function so I can use it in later chapters if needed

PC |> saveRDS("Code/QRE1/PCalgorithm.rds")
```

13.4 Some example games

13.4.1 Generalized matching pennies (Ochs 1995)

To begin with, I will show you the logit quantal response equilibrium for a generalized matching pennies game. Generalized matching pennies games are a good place to start due to their simplicity: there is only one path that we have to follow, which is the principal branch connecting the centroid $(\rho^0, 0)$ to the unique Nash equilibrium. Therefore we can be sure to trace out all quantal response equilibria by starting at the centroid. The game, studied in Ochs (1995), is shows in Table 27. This game is also used as an example in McKelvey and Palfrey (1995).

Here is some working constructing the function H and finding its derivatives:

$$H(\lambda,\rho) = \begin{pmatrix} \rho_U - \rho_D - \lambda(4\exp(\rho_L) - 1\exp(\rho_R)) \\ \rho_L - \rho_R - \lambda(-1\exp(\rho_U) + 1\exp(\rho_D)) \\ \exp(\rho_U) + \exp(\rho_D) - 1 \\ \exp(\rho_L) + \exp(\rho_R) - 1 \end{pmatrix}$$

$$\frac{\partial H(\lambda,\rho)}{\partial \lambda} = \begin{pmatrix} -(4\exp(\rho_L) - 1\exp(\rho_R)) \\ -(-1\exp(\rho_U) + 1\exp(\rho_D)) \\ 0 \\ 0 \end{pmatrix}$$

$$\frac{\partial H(\lambda,\rho)}{\partial \rho^\top} = \begin{bmatrix} 1 & -1 & -4\lambda\exp(\rho_L) & +1\lambda\exp(\rho_R) \\ 1\lambda\exp(\rho_U) & -1\lambda\exp(\rho_D) & 1 & -1 \\ \exp(\rho_U) & \exp(\rho_D) & 0 & 0 \\ 0 & 0 & \exp(\rho_L) & \exp(\rho_R) \end{bmatrix}$$

And here is how we pass this information to the solver function I wrote above:

```
GMP<-list(
  # Function defining QRE conditions
  H = function(lambda,rho) {
    rbind(
      rho[1]-rho[2]-lambda*(4*exp(rho[3])-1*exp(rho[4])),
      rho[3]-rho[4]-lambda*(-1*exp(rho[1])+1*exp(rho[2])),
      exp(rho[1])+exp(rho[2])-1,
      exp(rho[3])+exp(rho[4])-1
    )
  },
  # derivative with respect to lambda
  Hlambda = function(lambda,rho) {
    rbind(
      -(4*exp(rho[3])-1*exp(rho[4])),
      -(-1*exp(rho[1])+1*exp(rho[2])),
      0,
      0
    )
  },
  # derivative with respect to rho
  Hrho = function(lambda,rho) {
    rbind(
      c(1,-1,-4*lambda*exp(rho[3]),1*lambda*exp(rho[4])),
      c(1*lambda*exp(rho[1]),-1*lambda*exp(rho[2]),1,-1),
      c(exp(rho[1]),exp(rho[2]),0,0),
      c(0,0,exp(rho[3]),exp(rho[4]))
    )
  }
)

    QRE<-   PC(GMP,
            4, # total number of actions
            c(0,100), # c(lambda0,lambda_end)
            rep(log(0.5),4), # rho0
            0.1, # first step size
```

```
            1e-4,# minimum step size
            1.1, # max deceleration
            1000, # maximum number of iterations
            1e-4 # tolerance
        )

QRE<-cbind(exp(QRE[,1:4]),QRE[,5])
colnames(QRE)<-c("U","D","L","R","lambda")

QRE<-QRE |> data.frame()

(
  ggplot(QRE,aes(x=U,y=L))
  +geom_path(linewidth=1)
  +theme_bw()
  +coord_fixed(xlim=c(0,1),ylim=c(0,1))
)
```

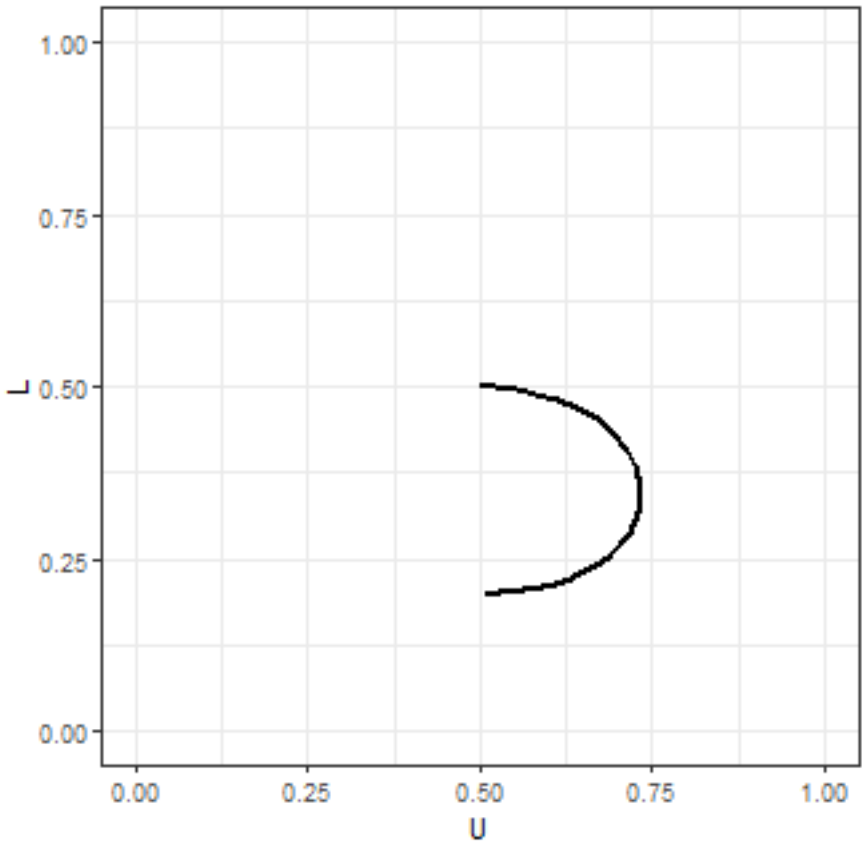


Figure 38: Locus of logit quantal response equilibrium probabilities for a generalized matching pennies game.

Figure @(fig:QRE1GMPlocus) shows the logit quantal response equilibrium *locus*. This is the set of mixed strategy profiles that are a quantal response equilibrium. While this kind of plot obfuscates the choice precision parametrer λ, I really like to show a QRE this way, as it very clearly shows the kind of predictions that it can make.[60]

[60]You can see a lot of plots like this one in Bland and Turocy (2025) and Bland (2023a).

However to visualize the curves that the predictor-corrector algorithm is actually tracing out, you need to include λ. This is shown in Figure 39. Note that I have included λ on a log scale. For this plot this transformation ensures that all of the interesting stuff isn't squished up into the first 10% of horizontal coordinates.

```
(
  ggplot(QRE,aes(x=lambda))
  +geom_path(aes(y=U,linetype="U"),linewidth=1)
  +geom_path(aes(y=L,linetype="L"),linewidth=1)
  +scale_x_continuous(trans="log10")
  +theme_bw()
  +xlab(expression(lambda))
  +ylab("Mixed strategy")
  +scale_linetype_discrete(name="")
)
```

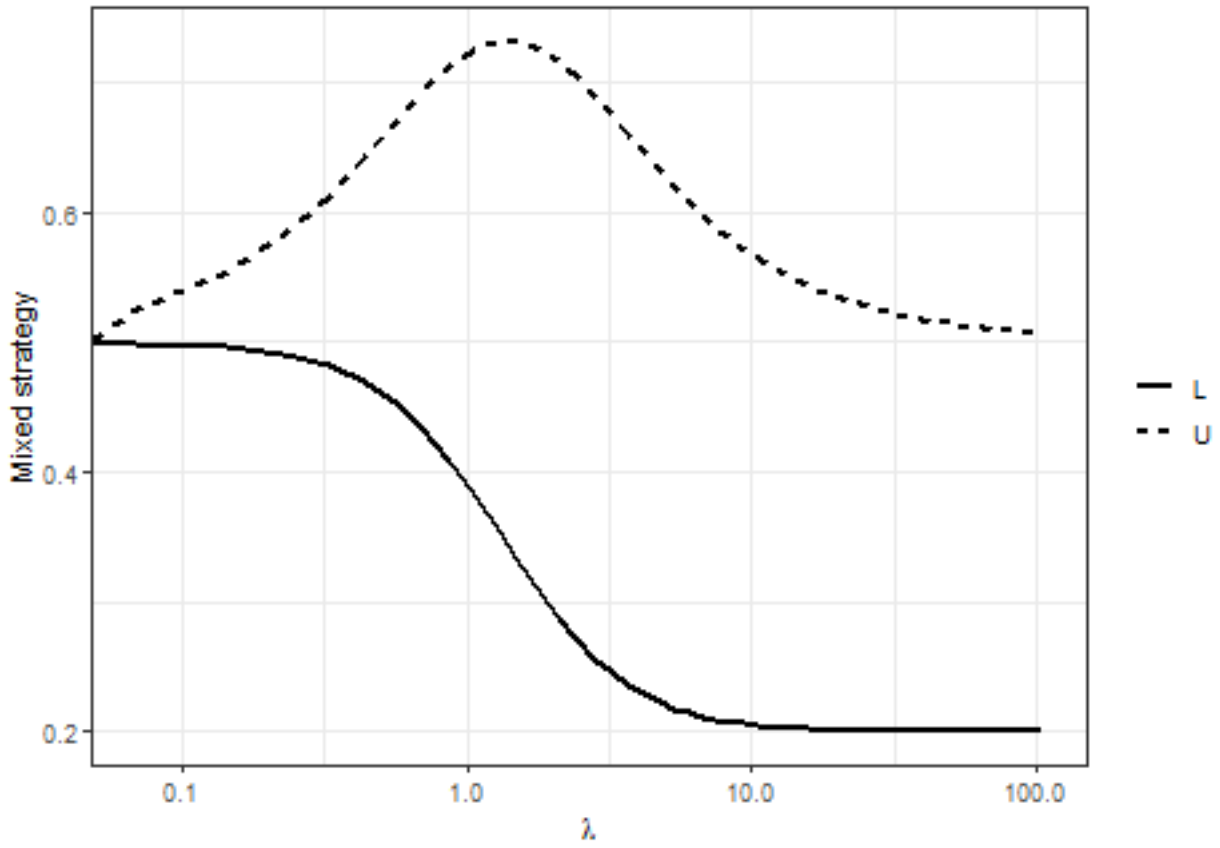


Figure 39: Logit quantal response equilibrium of generalized matching pennies game as a function of choice precision λ

13.4.2 Stag hunt

Table 28 shows a *stag hunt* game, which is a coordination game with three Nash equilibria: (U, L), (D, R), and $\sigma_U = \sigma_L = a/4$. This should get us a bit worried about computing quantal response equilibrium, because we will not find *all* of the paths if we just start at the centroid. To begin with, let's set up the system of equations, and see what happens when we start at the centroid. Here I am going to take advantage of the symmetry of the game, noting that I can set $\sigma_U = \sigma_L$ without loss of generality.

Table 28: A stag hunt game. $0 < a < 4$

	L	R
U	4, 4	0, a
D	a, 0	a, a

$$
\begin{aligned}
H(\lambda,\rho) &= \begin{pmatrix} \rho_U - \rho_D - \lambda(4\exp(\rho_U) - a) \\ \exp(\rho_U) + \exp(\rho_D) - 1 \end{pmatrix} \\
\frac{\partial H(\lambda,\rho)}{\partial \rho^\top} &= \begin{bmatrix} 1 - 4\lambda\exp(\rho_U) & -1 \\ \exp(\rho_U) & \exp(\rho_D) \end{bmatrix} \\
\frac{\partial H(\lambda,\rho)}{\partial \lambda} &= \begin{pmatrix} -(4\exp(\rho_U) - a) \\ 0 \end{pmatrix}
\end{aligned}
$$

```
StagHunt<-function(a) {
  list(
  H = function(lambda,rho) {
    rbind(
      rho[1]-rho[2]-lambda*(4*exp(rho[1])-a),
      exp(rho[1])+exp(rho[2])-1
    )
  }
  ,
  Hrho = function(lambda,rho) {
    rbind(
      c(1-4*lambda*exp(rho[1]),-1),
      c(exp(rho[1]),exp(rho[2]))
    )
  }
  ,
  Hlambda = function(lambda,rho) {
    rbind(
      -(4*exp(rho[1])-a),
      0
    )
  }
)
}

a<-1

QRE1<-   PC(StagHunt(a),
            2, # total number of actions
          c(0,10),
          rep(log(0.5),2), # rho0
            0.03, # first step size
            1e-4,# minimum step size
            1.1, # max deceleration
            1000, # maximum number of iterations
            1e-4 # tolerance
        )
```

```
QRE1[,1:2]<-exp(QRE1[,1:2])

QRE1<-QRE1 |> data.frame()
colnames(QRE1)<-c("U","D","lambda")

(
  ggplot(QRE1,aes(x=lambda,y=U))
  +geom_path(linewidth=1)
  +theme_bw()
  +scale_x_continuous(trans="log10")
  +xlab(expression(lambda))
  +geom_hline(yintercept=c(0,1,a/4),linetype="dashed")
)
```

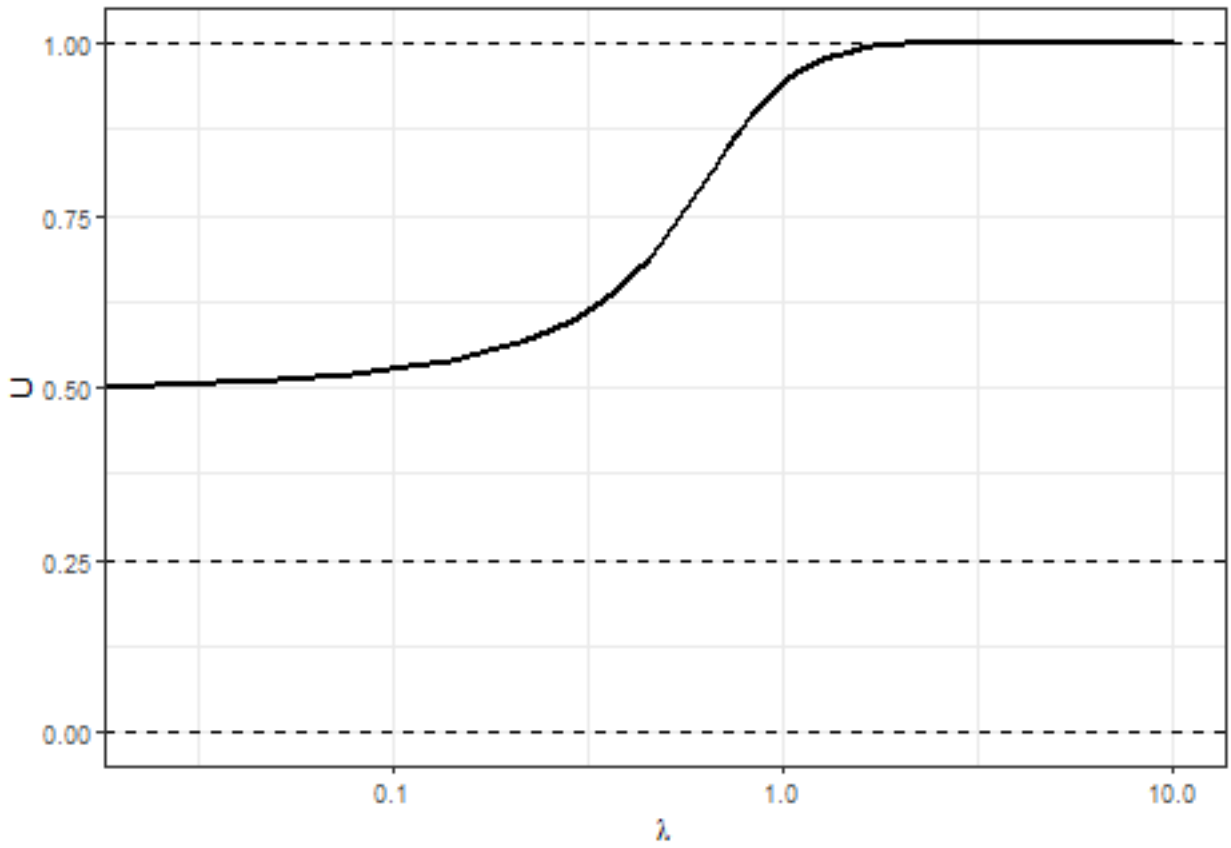


Figure 40: The principal branch of the logit quantal response equilibrium of the stag hunt game with $a = 1$.

Tracing out the principal branch of the logit QRE is done in exactly the same way as we did it for the asymmetric matching pennies games. This branch is shown in Figure 40 for $a = 1$. However note that we have only approach one of the three Nash equilibria, which are shown with horizontal dashed lines. In order to trace out the other parts of the quantal response equilibrium, we need to give the predictor-corrector algorithm different starting points. Let's do this and add them to the principal branch:

```
# Starting close to the mixed strategy Nash equilibrium

QRE2<-   PC(StagHunt(a),
            2, # total number of actions
```

```
            c(20,0),
            c(log(a/4),log(1-a/4)), # rho0
              0.001, # first step size
              1e-4,# minimum step size
              1.01, # max deceleration
              1000, # maximum number of iterations
              1e-4 # tolerance
        )

QRE2[,1:2]<-exp(QRE2[,1:2])

QRE2<-QRE2 |> data.frame()
colnames(QRE2)<-c("U","D","lambda")

## Joining them together

QRE<-rbind(
  QRE1 |> mutate(branch = "Principal"),
  QRE2 |> mutate(branch = "Other")
)

(
  ggplot(QRE,aes(x=lambda,y=U,color=branch))
  +geom_path(linewidth=1)
  +theme_bw()
  +scale_x_continuous(trans="log10")
  +xlab(expression(lambda))
  +geom_hline(yintercept=c(0,1,a/4),linetype="dashed")
)
```

The principal branch is shown alongside the other branch in Figure 41. Note with the above code that I had to tweak some of the tuning parameters: `max_deceleration` is much smaller, which means that the step size will change more slowly. What is also apparent from this Figure is that we can get turning points in λ. This can be seen in the "Other" branch (red curve). We therefore cannot always parameterize these curves as a function of λ.

An interesting knife-edge case for this game is when $a = 2$. Here the principal branch is constant at $\sigma_U = 0.5$, and hence is always equal to the mixed strategy Nash equilibrium. In this case, we get a bifurcation, and the two pure-strategy Nash equilibria are connected by a branch. Therefore, the only way to compute the non-principal branch is to start really close to one of the pure-strategy Nash equilibria. For this case, suppose we start close to the Nash equilibrium where $\sigma_U = 0$. This means that the expected utility of playing U will be approximately zero (for large but finite λ), and the expected utility of playing D will be $a = 2$. Therefore the approximate quantal response equilibrium mixed strategy for large λ will be:

$$\sigma_U(\lambda) \approx (1 + \exp(-\lambda(0 - 2)))^{-1}$$

```
a<-2


QRE1<-   PC(StagHunt(a),
             2, # total number of actions
            c(0,10),
            rep(log(0.5),2), # rho0
              0.03, # first step size
```

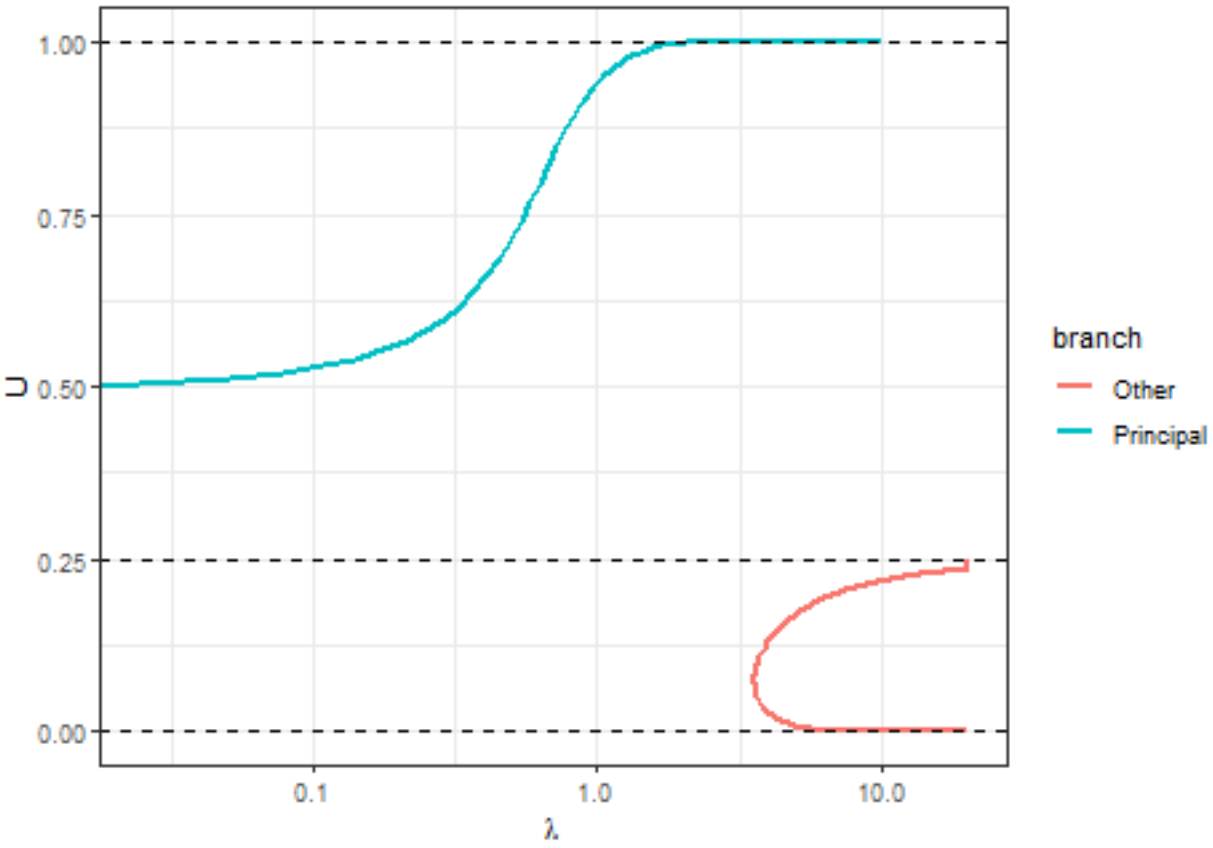


Figure 41: Logit quantal response equilibrium branches for the stag hunt game with $a = 1$.

```
            1e-4,# minimum step size
            1.1, # max deceleration
            1000, # maximum number of iterations
            1e-4 # tolerance
        )

QRE1[,1:2]<-exp(QRE1[,1:2])
QRE1<-QRE1 |> data.frame()
colnames(QRE1)<-c("U","D","lambda")

lambdaStart<-10

QRE2<-   PC(StagHunt(a),
            2, # total number of actions
          c(lambdaStart,0),
          c(-log(1+exp(2*lambdaStart)),-log(1+exp(-2*lambdaStart))), # rho0
            0.03, # first step size
            1e-4,# minimum step size
            1.01, # max deceleration
            1000, # maximum number of iterations
            1e-4 # tolerance
        )

QRE2[,1:2]<-exp(QRE2[,1:2])
QRE2<-QRE2 |> data.frame()
colnames(QRE2)<-c("U","D","lambda")

QRE<-rbind(
  QRE1 |> mutate(branch = "Principal"),
  QRE2 |> mutate(branch = "Other")
)

(
  ggplot(QRE,aes(x=lambda,y=U,color=branch))
  +geom_path(linewidth=1)
  +theme_bw()
  +scale_x_continuous(trans="log10")
  +xlab(expression(lambda))
  +geom_hline(yintercept=c(0,1,a/4),linetype="dashed")
)
```

13.4.3 n-player Volunteer's Dilemma imposing symmetric strategies

In their experimental analysis of the Volunteer's Dilemma (Diekmann 1985), Goeree, Holt, and Smith (2017) analyze the data through the lens of the *symmetric* quantal response equilibrium, which assumes that all players volunteer with the same probability.[61] The Volunteer's dilemma is an $n \geq 2$ person game, and is summarized in Table 29. Each player either volunteers (V) at a private cost of c, or does not volunteer D at no cost. If at least one player volunteers, then all players earn $V > c$, and if no player volunteers, all earn $L < V - c$.

The symmetry assumption here greatly simplifies the equilibrium predictions. For example, in Nash equilibrium, it rules out everything except for the symmetric mixed strategy Nash equilibrium. Similarly for quantal

[61] They then extend the model to investigate several motivations for heterogeneneous volunteering probabilities, which I might get to later.

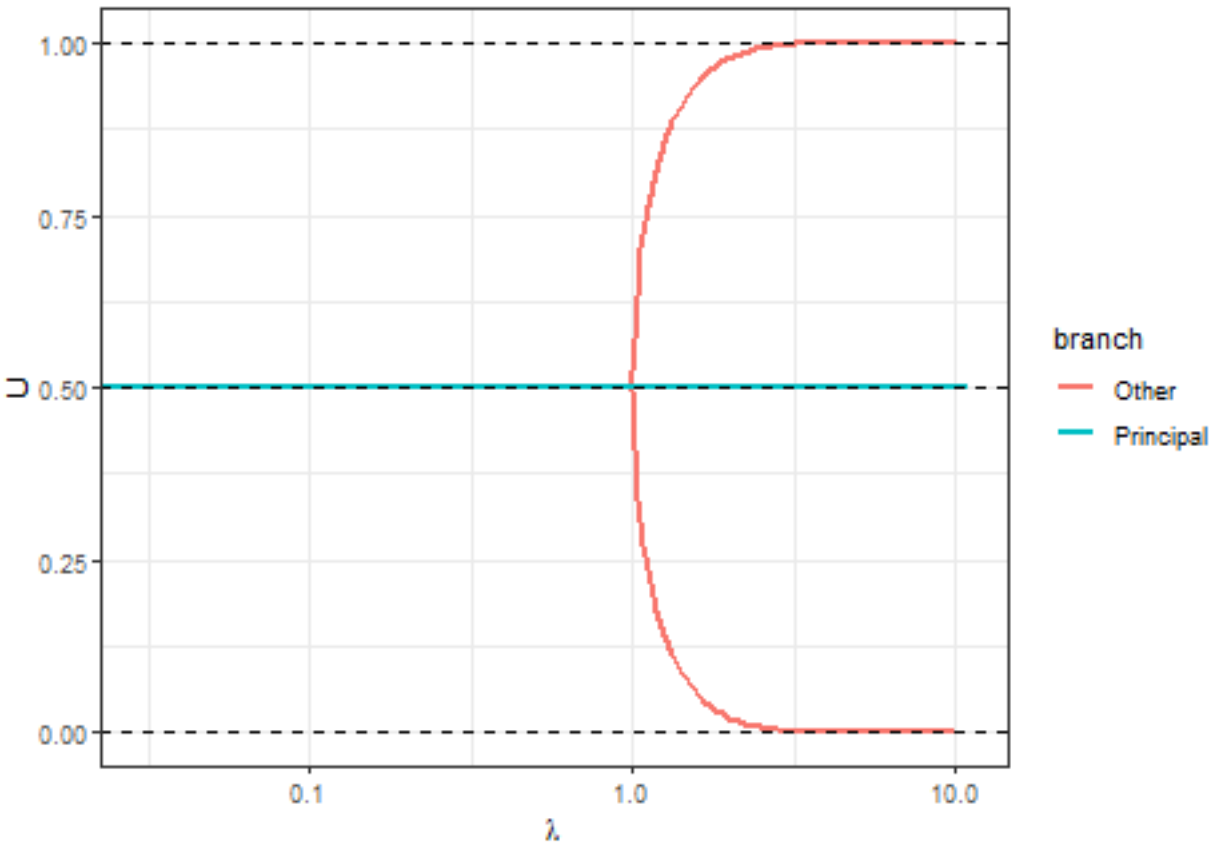


Figure 42: Logit quantal response equilibrium branches for the stag hunt game with $a = 2$.

Table 29: Payoffs for the Volunteer's Dilemma

	At least one other player volunteers	Nobody else volunteers
Volunteer	V-c	V-c
Don't volunteer	V	L

response equilibrium, we are left just with the equilibria where everybody volunteers with probability $\sigma_V(\lambda)$. In working out the payoff differences needed to compute quantal response equilibrium, it is useful to have an expression for the probability that none of the $n-1$ opponents volunteer:

$$\begin{aligned} p(\text{nobody else volunteers}) &= \sigma_N^{n-1} \\ &= \exp(\rho_N)^{n-1} \\ &= \exp((n-1)\rho_N) \\ p(\text{at least one other player volunteers}) &= 1 - \exp((n-1)\rho_N) \end{aligned}$$

These can then be used to construct $H(\rho, \lambda)$ and its derivatives as follows:

$$\begin{aligned} H(\lambda,\rho) &= \begin{pmatrix} \rho_V - \rho_N - \lambda(V - c - V(1-\exp((n-1)\rho_N)) - L\exp((n-1)\rho_N))) \\ \exp(\rho_V) + \exp(\rho_N) - 1 \end{pmatrix} \\ \frac{\partial H(\lambda,\rho)}{\partial \rho^\top} &= \begin{bmatrix} 1 & -1 - \lambda(n-1)\exp((n-1)\rho_N)\,(V-L) \\ 1 & 1 \end{bmatrix} \\ \frac{\partial H(\rho,\lambda)}{\partial \lambda} &= \begin{pmatrix} -(V - c - V(1-\exp((n-1)\rho_N)) - L\exp((n-1)\rho_N))) \\ 0 \end{pmatrix} \end{aligned}$$

Here are these functions coded in *R*:

```
VolunteersDilemma<-function(n,V,c,L) {
  list(
    H = function(lambda,rho) {
      rbind(
        rho[1]-rho[2]-lambda*(
          V-c-V*(1-exp((n-1)*rho[2]))-L*exp((n-1)*rho[2])
        ),
        exp(rho[1])+exp(rho[2])-1
      )
    }
    ,
    Hrho = function(lambda,rho) {
      rbind(c(1, -1-lambda*(n-1)*exp((n-1)*rho[2])*(V-L)),
            c(1, 1)
      )
    }
    ,
    Hlambda = function(lambda,rho) {
      rbind(
        -( V-c-V*(1-exp((n-1)*rho[2]))-L*exp((n-1)*rho[2])),
        0
      )
    }
  )
}
```

And now we can solve this for a few values of the group size n (here I hold $V = 1$, $c = 0.1$, and $L = 0.2$ constant):

```
QRE<-data.frame()

for (nn in 2:6) {
```

Table 30: Payoffs for the Volunteer's Dilemma

	At least one other player volunteers	Nobody else volunteers
Volunteer	V-c	V-c
Don't volunteer	V	L

```
  qre<-PC(VolunteersDilemma(nn,1,0.1,0.2),
              2, # total number of actions
            c(0,1000),
            rep(log(0.5),2), # rho0
              0.03, # first step size
              1e-4,# minimum step size
              1.1, # max deceleration
              1000, # maximum number of iterations
              1e-4 # tolerance
          )

  qre[,1:2]<-exp(qre[,1:2])
  colnames(qre)<-c("V","D","lambda")

  QRE<-rbind(
    QRE,
    qre |>
      data.frame() |>
      mutate(`group size` = nn)
  )

}

(
  ggplot(QRE,aes(x=lambda,y=V,color=as.factor(`group size`)))
  +geom_path(linewidth=1)
  +theme_bw()
  +scale_x_continuous(trans="log10")
  +xlab(expression(lambda))
  +ylab("Probability of volunteering")
  +scale_color_discrete(name="group size")
  +ylim(c(0,1))

)
```

14 Application: Quantal Response Equilibrium and the Volunteer's Dilemma (Goeree, Holt, and Smith 2017)

With what we have learned in the previous chapter about *computing* a quantal response equilibrium, we can now go and *estimate* one! Mechanically, this will just mean translating the *R* code I had in the previous chapter into *Stan*, and then combining the log probabilities that we get out of the predictor-corrector algorithm with choice frequencies to get our likelihood. As always, we will also need to have a good think about priors. This is especially important in this case, where the model is highly non-linear, and it may not be too easy to interpret the choice precision parameter without some carefully thought-out context.

For this chapter, I will show you my Bayesian take on the quantal response equilibrium analysis in Goeree,

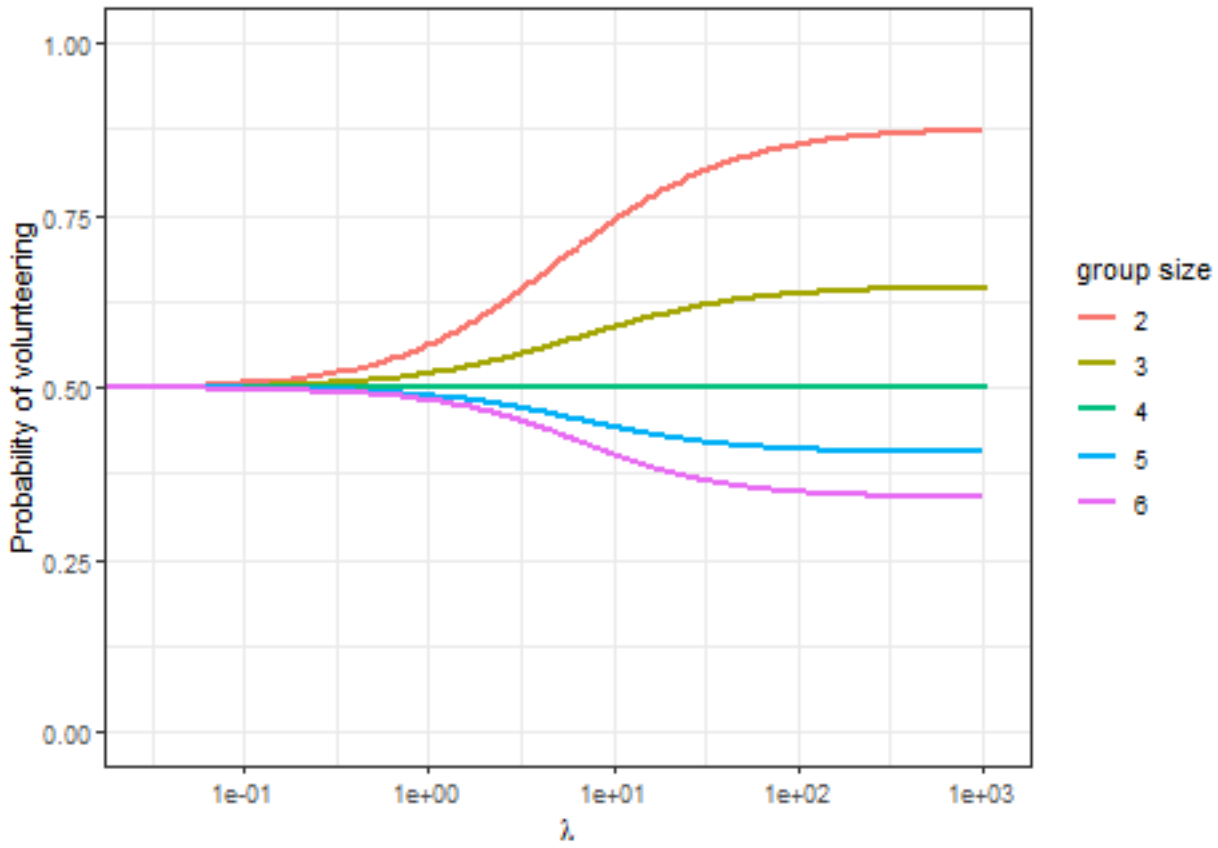


Figure 43: The symmetric quantal response equilubrium paths for the n-player Volunteer's Dilemma with $V = 1$, $c = 0.1$ and $L = 0.2$

Holt, and Smith (2017), which studies the effect of group size in a Volunteer's Dilemma. The game is described in Table 30. Importantly, note that this is *not* a payoff matrix: instead of a second player's actions being the columns of the table, these columns refer to a *summary* of one's opponents' actions. In the Volunteer's dilemma, there is a task that the n players can either volunteer (V) for at cost c, or not volunteer (N) for at no cost. If at least one player volunteers, then *all* players get a large benefit V, and if nobody volunteers all players get a smaller payoff L. While there are many pure strategy Nash equilibria of this game that involve exactly one player volunteering and all other not volunteering, it is natural here to focus on the symmetric mixed-strategy Nash equilibrium, where players all volunteer with probability:

$$\sigma_V^* = 1 - \left(\frac{c}{V-L}\right)^{\frac{1}{n-1}}$$

Goeree, Holt, and Smith (2017) hold constant $V = 1$ and $L = c = 0.2$, and vary the group size n. The data from their main treatments, along with the Nash equilibrium prediction is shown in Figure 44

```
D<-read.csv("Data/GHS2017VD.csv") |>
  mutate(GroupSize = str_count(Other.Id., "ID")+1,
         GroupSize = ifelse(SessionID==10,9,GroupSize),
         GroupSize = ifelse(SessionID==11,12,GroupSize)
         ) |>
  rename(Volunteer = Decision..1.vol..) |>
  mutate(ID = paste0(SessionID,"-",ID) |> as.factor() |> as.numeric() ) |>
  dplyr::select(ID,GroupSize,Volunteer) |>
  # all I need are the counts of actions for each participant:
  group_by(ID,GroupSize) |>
  summarize(Volunteer = sum(Volunteer),count=n())
```

```
V<-1
L<-0.2
c<-0.2

Nash<-tibble(
  GroupSize = 2:12,
) |>
  mutate(Nash = 1-(c/(V-L))^(1/(GroupSize-1)))

(
  ggplot()
  +stat_boxplot(data=D,aes(x=GroupSize,y=Volunteer/20,group=GroupSize))
  +geom_line(data=Nash,aes(GroupSize,y=Nash),color="red")
  +theme_bw()
  +xlab("Group size")
  +ylab("Volunteer rate")
)
```

14.1 Solving logit QRE and estimating the model

From the previous chapter, the system of equations that we are trying to solve is:

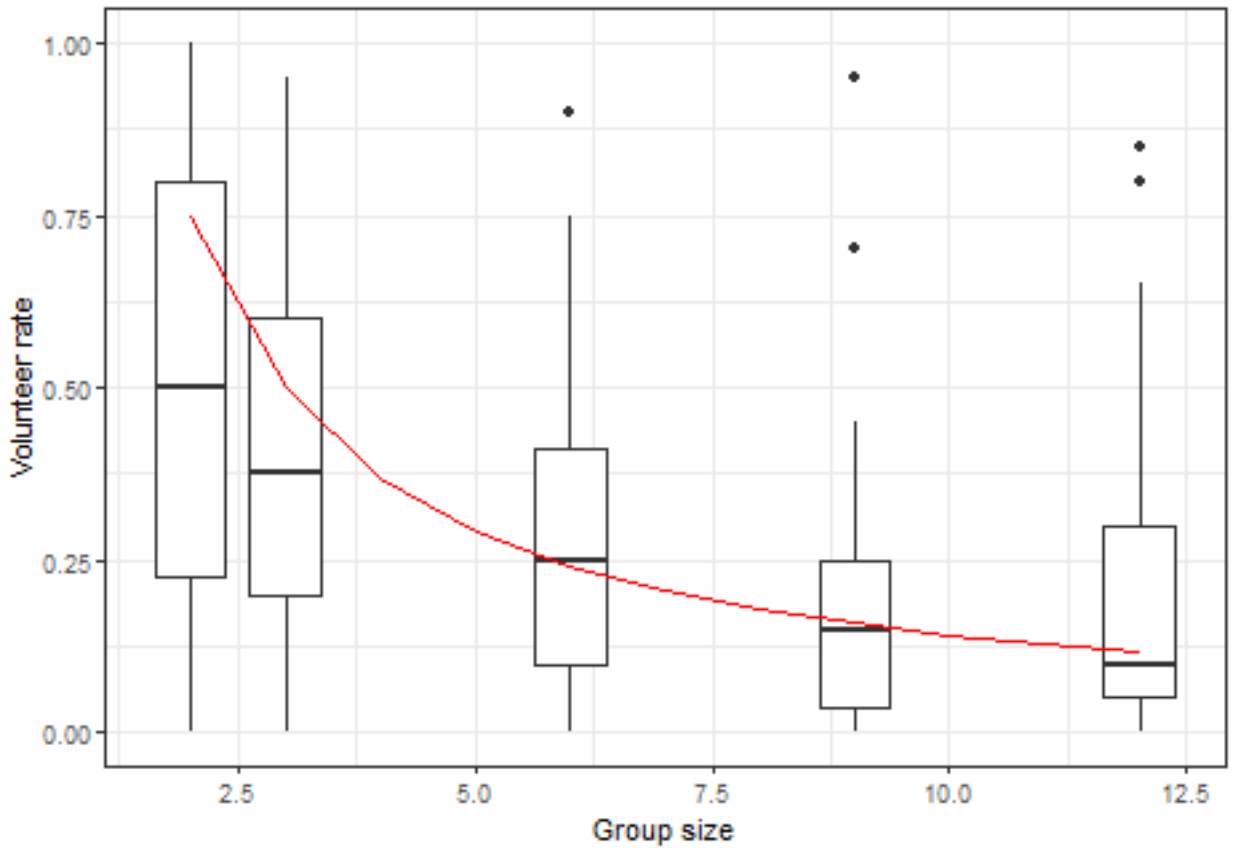


Figure 44: Nash prediction (red line) and participant-specific volunteer rates (black box plots)

$$
\begin{aligned}
H(\lambda, \rho) &= \begin{pmatrix} \rho_V - \rho_N - \lambda(V - c - V(1 - \exp((n-1)\rho_N)) - L\exp((n-1)\rho_N))) \\ \exp(\rho_V) + \exp(\rho_N) - 1 \end{pmatrix} \\
\frac{\partial H(\lambda, \rho)}{\partial \rho^\top} &= \begin{bmatrix} 1 & -1 - \lambda(n-1)\exp((n-1)\rho_N)\,(V - L) \\ 1 & 1 \end{bmatrix} \\
\frac{\partial H(\rho, \lambda)}{\partial \lambda} &= \begin{pmatrix} -(V - c - V(1 - \exp((n-1)\rho_N)) - L\exp((n-1)\rho_N))) \\ 0 \end{pmatrix}
\end{aligned}
$$

Below you can see my *Stan* program that implements the predictor-corrector algorithm in the `functions` block. One issue I came across was that the algorithm as it is written in the previous chapter stops when $\lambda^t > \lambda$, where λ is the choice precision for which we want to compute the equilibrium. This is a problem for estimation, because we need the quantal response equilibrium probabilities at λ *exactly*, not for some $\lambda^t > \lambda$ (but nevertheless somewhat "close" to λ). To fix this, I add in some constant-λ corrector steps at the end of the algorithm to backtrack to the desired probabilities. These steps use Newton's method to find solution ρ to $H(\lambda, \rho) = 0$:[62]

$$
\rho^{t+1} = \rho^t - \left[\frac{\partial H(\lambda, \rho^t)}{\partial \rho^\top}\right]^{-1} H(\lambda, \rho^t)
$$

For quantal response equilibrium models, it is especially important to translate the choice precision parameter λ into something easier to interpret. Following Goeree, Holt, and Smith (2017), I therefore also compute the implied probabilities of volunteering in the `generated quantities` block.

```
/* Homogeneous quantal response equilibrium model for the Volunteer's Dilemma
*/

functions {

  vector Hfun(vector x,
  data real n, data real V, data real c, data real L) {

    return [x[1]-x[2]-x[3]*(V-c-V*(1-exp((n-1.0)*x[2]))-L*exp((n-1.0)*x[2])),
    exp(x[1])+exp(x[2])-1.0]';
  }

  matrix jac(vector x,
  data real n, data real V, data real c, data real L) {


    return [
      [1, -1-x[3]*(n-1.0)*exp((n-1.0)*x[2])*(V-L), -(V-c-V*(1-exp((n-1.0)*x[2]))-L*exp((n-1.0)*x[2]))],
      [1, 1, 0]
    ];
  }


  vector lpQRE(
    real lambda,
    data real n,// Group size
    data real V, data real c, data real L,
```

[62] For some games, these constant-λ corrector steps may be sufficient on their own to solve for the quantal response equilibrium. For example, in Bland (2023a) I find that corrector steps starting at the centroid are just fine for computing equilibrium in some generalized matching pennies games.

```
  data real first_step,
  data real min_step,
  data real max_decel,
  data int maxiter,
  data real tol
) {

  real max_dist = 0.4;
  real max_contr = 0.6;
  real eta = 0.1;

  vector[3] x = [log(0.5),log(0.5),0]';

  vector[3] t = qr_Q(jac(x,n,V,c,L)')[,3];

  real h = first_step;

  real omega = t[3]>=0 ? 1.0 : -1.0;

  // Here we are just going from 0 to lambda

  while (x[3]<=lambda) {

    int accept = 1;

    if (abs(h)<=min_step) {
      // step size below minimum step
      print("Step size",h,"less than minimum");
      break;
    }

    // predictor step
    vector[3] u = x+h*omega*t;
    matrix[3,3] q = qr_Q(jac(u,n,V,c,L)');

    real disto = 0;
    real decel = 1.0/max_decel;

    for (it in 1:maxiter) {

      vector[2] dx = -generalized_inverse(jac(u,n,V,c,L))*Hfun(x,n,V,c,L);
      real dist = sqrt(dx'*dx);

      if (dist >= max_dist) {
        print("Proposed distance ",dist," exceeds max_dist ",max_dist);
        accept = 0;
        break;
      }

      decel = fmax(decel,sqrt(dist/max_dist)*max_decel);

      if (it>1) {
        real contr = dist/(disto+tol*eta);
```

```
      if (contr>max_contr) {
        print("Maximum control rate exceeded");
        accept=0;
        break;
      }
      decel = fmax(decel,sqrt(contr/max_contr)*max_decel);
    }

    if (dist>tol) {
      // success! The corrector step has corrected enough
      break;
    }


    disto=dist;

  /* if we have got to this point, then:
     1. The Newton step has not proposed a too-large change
     2. The contraction rate has not been exceeded
     3. The proposed Newton step is not so small that we have effectively
       found a solution
    Therefore, we update u
   */

   u = u+dx;

   if (it==maxiter) {
     // we have run out of iterations. Terminate
     print("Maximum number of iterations ",maxiter," reached");

   }



  } // END OF CORRECTOR STEP

  if (accept!=1) {
    h = h/max_decel;
  }

  h = abs(h/fmin(decel,max_decel));

  if ((t'*q[,3])<0) {
    omega = -omega;
  }
  x = u;
  t = q[,3];

}


/* Some constant-lambda corrector steps to get the QRE back from
over-shooting a bit
```

```
  */

  for (ii in 1:maxiter) {

    x[3] = lambda;

    vector[2] dx = -jac(x,n,V,c,L)[,1:2]\Hfun(x,n,V,c,L);

    x[1:2] +=dx;

    if (sqrt(dx'*dx)<tol) {
      break;
    }

  }

  return x[1:2];
}



/* Returns the log-QRE probabilities

*/
vector lpQREcorrectorOnly(
  real lambda,
  data int GroupSize,
  data real V, data real c, data real L,
  data real tol
) {

  // initial conditions
  vector[2] x = rep_vector(log(0.5),2);
  real dist = 1e12;

  while (dist>tol) {
  //for (kk in 1:10) {
    vector[2] H = [x[1]-x[2]-lambda*(V-c-V*(1.0-exp((GroupSize-1.0)*x[2]))-L*exp((GroupSize-1.0)*x[2]
    exp(x[1])+exp(x[2])-1.0
    ]';

    matrix[2,2] Hrho =
      [
        [1.0, -1.0-lambda*(GroupSize-1.0)*exp((GroupSize-1.0)*exp(x[2]))*(V-L)],
        [1.0, 1.0]
      ];

    vector[2] dx = -Hrho\H;

    x = x+dx;

    dist = sqrt(dx'*dx);
```

```
    }

    return x;

  }

}

data {
  int<lower=0> N; // Number of participants
  int<lower=0> Volunteer[N];
  int<lower=0> count[N];
  int<lower=2> GroupSize[N];

  // Game parameters
  real V; // Benefit of volunteering
  real c; // Cost of volunteering
  real L; // Benefit if nobody volunteers

  real prior_lambda[2]; // lognormal prior



  real<lower=0> first_step;
  real<lower=0> min_step;
  real<lower=0> max_decel;
  int<lower=0> maxiter;
  real<lower=0> tol; // tolerance for the corrector step

}

parameters {
  // logit choice precision
  real<lower=0> lambda;

}

model {

  for (ii  in 1:N) {
    vector[2] lp = lpQRE(
     lambda,
     GroupSize[ii],// Group size
     V,  c,  L,
     first_step,
     min_step,
     max_decel,
     maxiter,
     tol
  );

  target+=Volunteer[ii]*lp[1]+(count[ii]-Volunteer[ii])*lp[2];
```

```
  }

  lambda ~ lognormal(prior_lambda[1],prior_lambda[2]);

}

generated quantities {
  vector[N] pV;

  for (ii in 1:N) {
    vector[2] lp = lpQRE(
     lambda,
    GroupSize[ii],// Group size
    V, c,  L,
    first_step,
    min_step,
    max_decel,
    maxiter,
    tol
  );

    pV[ii] = exp(lp[1]);
  }
}
```

As you can see in the `model` block, I have chosen to use a log-normal prior for λ. This ensures that it is positive. In this case, the prior I initially used:

$$\log \lambda \sim N(\log(10), 0.25^2)$$

produced results that were almost identical to the maximum likelihood estimates of Goeree, Holt, and Smith (2017). However I was worried about the prior median being 10, which is almost exactly the maximum likelihood estimate for λ. In particular, I wanted to show that a prior *not* centered on the maximum likelihood estimate would also generate similar results. Therefore, I re-estimated the model with:

$$\log \lambda \sim N(\log(5), 1^2)$$

which halves the prior median for λ. As you will see below, this barely changes the results: the posterior distribution must be dominated by the likelihood contribution. For perspective, a 95% prior Bayesian credible region for λ is:

$$[\exp(\log 5 - 1.96),\ \exp(\log 5 + 1.96)] \approx [0.70,\ 35.5]$$

Without further ado, we can now look at the posterior estimates, which are shown in Table 31.

```
FitSum<-summary(readRDS("Code/QRE2/Estimates_VDQREhomogeneous.rds"))$summary |>
  data.frame()

TAB <- FitSum |>
  dplyr::select(mean,sd)

rownames(TAB)<-c("$\\lambda$","n = 2","n = 3","n = 6","n = 9","n = 12","lp__")

TAB |> round(4) |>  kbl(caption="Posterior moments of the homogeneous model. $\\lambda$ is logit choice
```

Table 31: Posterior moments of the homogeneous model. λ is logit choice precision. Rows with n show predictions for the probability of volunteering for each group size

	mean	sd
λ	11.1159	1.1923
n = 2	0.6699	0.0056
n = 3	0.5000	0.0000
n = 6	0.3084	0.0057
n = 9	0.2358	0.0076
n = 12	0.1967	0.0086
lp___	-2393.1565	0.7535

Comparing this Table to Table 3 of Goeree, Holt, and Smith (2017), the results are remarkably similar. I estimate a posterior mean $\lambda \approx 11.1$, whereas their MLE estimate is $\lambda = 11.0$. Furthermore, all predicted probabilities of the Bayesian model are accurate to the two decimal places of their counterparts in Goeree, Holt, and Smith (2017). Probably the most notable difference between my Table 31 and their Table 3 is that I also have an expression of uncertainty for the predicted probabilities. This is something that is done by default in *Stan*. For the maximum likelihood implementation, one would either have to use the delta method or bootstrap to get standard errors on these quantities.[63]

14.2 Adding some heterogeneity

The homogeneous quantal response equilibrium estimated above captures an important feature of the data: volunteer rates are on average pulled toward 50% compared to the mixed-strategy Nash prediction. However Goeree, Holt, and Smith (2017) also note that the sample *variance* of participant-specific volunteer rates is substantially greater than the homogeneous quantal response equilibrium predicts (see their Table 4). This suggests that there could be some participant-specific heterogeneity, and Goeree, Holt, and Smith (2017) explore this with two extensions to the homogeneous model: warm glow volunteering, and duplicate aversion. In this section I will show you how to compute the equilibrium for these models, and then estimate them. This will mean modifying the predictor-corrector algorithm, as well as using some Monte Carlo integration. In both situations this will mean solving for the *average* volunteer rate, which I will denote $\bar{\sigma}_V$, and then augmenting the data appropriately to account for the individual-level heterogeneity.

As the steps I use to compute the equilibrium are almost identical for each of these extensions, I will first outline how this is done in general. I will then introduce the two models.

14.2.1 Computing quantal response equilibrium with heterogeneous parameters

In both of the model extensions, a participant-specific payoff parameter is added to the model. Here I will denote it as θ_i. We then specify a distribution for θ_i. Here I will use:[64]

$$\theta_i \sim \text{Truncated Normal}(\mu, \tau^2, (\mu - \tau, \mu + \tau))$$

This is slightly different to the implementation in Goeree, Holt, and Smith (2017), who assume a *discrete* truncated normal distribution. Specifically, they assume that θ_i is evenly spaced along a grid:[65]

$$\theta_i \in \{\mu - \tau, \mu - 0.8\tau, \mu - 0.6\tau, \ldots, \mu, \ldots, \mu + \tau\}$$

[63] You can see some standard errors on predicted QRE probabilities in Holt, Sahu, and Smith (2022) using the delta method. See their Table 6 and footnote 10.

[64] An earlier attempt at using the un-truncated version of the normal distribution resulted in a host of errors from *Stan*, mainly the `BFMI low` error. I suspect that when using this distribution, the likelihood is having trouble with extreme values of θ_i in the data augmentation part of the model.

[65] This discretization is also used in Goeree, Holt, and Laury (2002).

My departure from the original specification, though, will permit us to use data augmentation to estimate the model.

Since θ_i is participant-specific, we therefore have a participant-specific probabilistic best response function, which I shall denote $\sigma(\lambda, \theta_i, \bar{\sigma}_V)$. $\bar{\sigma}_V$ is the *aggregate* volunteer rate in the population. We can therefore find it by taking the expectation over θ_i:

$$\bar{\sigma}_V = E_\theta[\sigma(\lambda, \theta, \bar{\sigma}_V)]$$

Note that on the left-hand side we have the average equilibrium volunteer rate, and on the left-hand side we have the average probabilistic best response. This is the equilibrium condition, as in equilibrium both of these quantities must be equal.

We have two computational issues that we will need to tackle in order to solve this equation. These are computing the expectation, and then solving the equation. For the former, I approximate the expectation using Monte Carlo integration as follows:

$$\begin{aligned} E_\theta[\sigma(\lambda, \theta, \bar{\sigma}_V)] &\approx \frac{1}{S}\sum_{s=1}^{S} \sigma(\lambda, \theta_s, \bar{\sigma}_V) \\ \theta_s &\sim \text{Truncated Normal}(\mu, \tau^2, (\mu - \tau, \mu + \tau)) \end{aligned}$$

I then need an algorithm that can solve the (approximate) equilibrium condition. For this, I will use constant-λ corrector steps, which are basically just solving the problem using Newton's method. First, in order to solve the problem on the real number line, rather than on the unit interval, I will make the logit transformation $\ell = \log(\bar{\sigma}_V) - \log(1 - \bar{\sigma}_V)$. I can then re-write the equilibrium condition as:

$$H(\ell) = \ell - \log(E_\theta(\sigma(\lambda, \theta, \Lambda(\ell)))) + \log(1 - E_\theta(\sigma(\lambda, \theta, \Lambda(\ell))))$$

where $\Lambda(x) = (1 + \exp(-x))^{-1}$ is the inverse logit function. We can therefore iterate on the following sequence until we reach the desired level of accuracy:

$$\ell^{t+1} = \ell^t - \frac{H(\ell^t)}{H'(\ell^t)}$$

where:

$$\begin{aligned} H'(\ell) &= 1 - \left[\frac{1}{E_\theta(\sigma(\lambda, \theta, \Lambda(\ell)))} - \frac{1}{1 - E_\theta(\sigma(\lambda, \theta, \Lambda(\ell)))}\right] E\left[\frac{\partial \sigma(\lambda, \theta, \Lambda(\ell))}{\partial \Lambda(\ell)}\right] \frac{\partial \Lambda(\ell)}{\partial \ell} \\ &= 1 - \frac{\Lambda(\ell)(1 - \Lambda(\ell))}{E_\theta(\sigma(\lambda, \theta, \Lambda(\ell)))(1 - E_\theta(\sigma(\lambda, \theta, \Lambda(\ell))))} E\left[\frac{\partial \sigma(\lambda, \theta, \Lambda(\ell))}{\partial \Lambda(\ell)}\right] \end{aligned}$$

Note that we will also have to approximate $E\left[\frac{\partial \sigma(\lambda, \theta, \Lambda(\ell))}{\partial \Lambda(\ell)}\right]$ using Monte Carlo integration.

14.2.2 Warm glow volunteering

The first extension is to introduce a participant-specific parameter δ_i that measures their "warm glow" utility from volunteering. That is, instead of receiving payoff $V - c$ when they volunteer, they receive payoff $V - c + \delta_i$. We can therefore write participant i's probabilistic best response to the average volunteer rate $\bar{\sigma}_V$ as:

$$\begin{aligned} \sigma(\lambda, \delta_i, \bar{\sigma}_V) &= \frac{\exp(\lambda(V - c + \delta_i))}{\exp(\lambda(V - c + \delta_i)) + \exp(\lambda(V(1 - (1 - \bar{\sigma}_V)^{n-1}) + L(1 - \bar{\sigma}_V)^{n-1}))} \\ &= \Lambda\left(\lambda\left(V - c + \delta_i - V(1 - (1 - \bar{\sigma}_V)^{n-1}) - L(1 - \bar{\sigma}_V)^{n-1}\right)\right) \end{aligned}$$

The derivative of the probabilistic best response that we need to evaluate is:[66]

$$\frac{\partial \sigma(\lambda, \delta_i, \bar{\sigma}_V)}{\partial \bar{\sigma}_V} = -\lambda(n-1)(1-\bar{\sigma}_V)^{n-2}(V-L)\sigma(\lambda, \delta_i, \bar{\sigma}_V)(1-\sigma(\lambda, \delta_i, \bar{\sigma}_V))$$

Here is the *Stan* program I wrote to estimate the warm glow model:

```
/* Warm glow quantal response equilibrium model for the Volunteer's Dilemma
*/

functions {


  /* Here I am using "PBR" (i.e. probabilistic best response) to
  distinguish between \sigma and \bar\sigma_V
  */

  vector PBR(
    real lambda, vector delta, real sigmaV,
    data real V, data real c, data real L, data real n
  ) {

    return inv_logit(lambda*(
      V-c+delta-V*(1.0-pow(1.0-sigmaV,n-1.0))+L*pow(1.0-sigmaV,n-1.0)
    ));

  }


 // derivative wrt sigmaV
 vector PBR_sigmaV(
    real lambda, vector delta, real sigmaV,
    data real V, data real c, data real L, data real n
  ) {
    return -lambda*(V-L)*(n-1.0).*pow(1.0-sigmaV,n-2.0)
    *PBR(lambda,delta,sigmaV,V,c,L,n) .*(1.0-PBR(lambda,delta,sigmaV,V,c,L,n));
  }


  // Zero condition for equilibrium
  real Hfun(real l,
    real lambda, vector delta,
    data real V, data real c, data real L, data real n
    ) {
      real sigmaV = inv_logit(l);
      real meanPBR = mean(PBR(lambda,delta,sigmaV,V,c,L,n));
      return l - log(meanPBR)+log(1.0-meanPBR);
    }

  // derivative of zero condition
  real Hl(real l,
    real lambda, vector delta,
```

[66] Here I am using the following result for the derivative of the inverse logit function $\Lambda'(x) = \Lambda(x)(1-\Lambda(x))$

```
    data real V, data real c, data real L, data real n
    ) {
      real sigmaV = inv_logit(l);
      real meanPBR = mean(PBR(lambda,delta,sigmaV,V,c,L,n));
      return 1.0-sigmaV*(1.0-sigmaV)/(meanPBR*(1.0-meanPBR))*mean(PBR_sigmaV(lambda,delta,sigmaV,V,c,L,n
    }

  // Compute QRE using constant-lambda corrector steps
  real lpQRE(
    real lambda, vector delta,
    data real V, data real c, data real L, data real n,
    data real tol, data int maxiter
  ) {

    // start at l=0 (sigmaV=0.5)
    real l = 0.0;

    for (it in 1:maxiter) {

      real dl = -Hfun(l,lambda,delta,V,c,L,n)/Hl(l,lambda,delta,V,c,L,n);

      l = l+dl;

      if (abs(dl)<tol) {
        break;
      }

      if (it==maxiter) {
        print("Maximum iterations error, lambda = ",lambda, ", dl = ",abs(dl));
      }

    }

    return inv_logit(l);

  }




}

data {
  int<lower=0> N; // Number of participants
  int<lower=0> Volunteer[N];
  int<lower=0> count[N];

  int<lower=0> nTreatments;
  vector[nTreatments] GroupSize;

  int<lower=0> GameID[N];
```

```
  // Game parameters
  real V; // Benefit of volunteering
  real c; // Cost of volunteering
  real L; // Benefit if nobody volunteers

  real prior_lambda[2]; // lognormal prior
  real prior_mu[2];
  real prior_tau;



  int<lower=0> maxiter;
  real<lower=0> tol; // tolerance for the corrector step

  int<lower=1> nSim; // simulation size for Monte Carlo integration
  vector<lower=-1.0,upper=1.0>[nSim] deltaZ; // standard normals truncated to be between -1 and 1

}

parameters {
  // logit choice precision
  real<lower=0> lambda;

  // Distribution of delta
  real mu;
  real<lower=0> tau;

  vector<lower = mu-tau,upper=mu+tau>[N] delta_i;


}

model {



  vector[nSim] deltaSim = mu+tau*deltaZ;

  vector[nTreatments] sigmaV;

  for (tt  in 1:nTreatments) {
    sigmaV[tt] = lpQRE(
    lambda,deltaSim,
    V,c,L,GroupSize[tt],
    tol,maxiter
  );

  }

  Volunteer ~ binomial_logit(count,lambda*(
      V-c+delta_i-V*(1.0-pow(1.0-sigmaV[GameID],GroupSize[GameID]-1.0))+L*pow(1.0-sigmaV[GameID],GroupS
    ));
```

```
  delta_i ~ normal(mu,tau);

  lambda ~ lognormal(prior_lambda[1],prior_lambda[2]);
  mu ~ normal(prior_mu[1],prior_mu[2]);
  tau ~ cauchy(0,prior_tau);

}

generated quantities {


  vector[nTreatments] sigmaV;
  {
  vector[nSim] deltaSim = mu+tau*deltaZ;
  for (tt  in 1:nTreatments) {
    sigmaV[tt] = lpQRE(
    lambda,deltaSim,
    V,c,L,GroupSize[tt],
    tol,maxiter
  );

  }
  }

}
```

14.2.3 Duplicate aversion

Goeree, Holt, and Smith (2017) also extend the homogeneous quantal response equilibrium model to allow for participant-specific "duplicate aversion". That is, participants suffer disutility γ_i if more than one person volunteers. This modifies the expected payoff of volunteering from $V - c$ to $V - c - \gamma_i(1 - (1 - \bar{\sigma}_V)^{n-1})$, and so the probabilistic best response function for a participant with duplicate aversion γ_i is:

$$\sigma(\lambda, \gamma_i, \bar{\sigma}_V) = \Lambda\left(\lambda\left(V - c - \gamma_i(1 - (1 - \bar{\sigma}_V)^{n-1}) - V(1 - (1 - \bar{\sigma}_V)^{n-1}) - L(1 - \bar{\sigma}_V)^{n-1}\right)\right)$$

and the derivative we need to evaluate is:

$$\frac{\partial \sigma(\lambda, \gamma_i, \bar{\sigma}_V)}{\partial \bar{\sigma}_V} = -\lambda(n-1)(1 - \bar{\sigma}_V)^{n-2}\left(\gamma_i + V - L\right)\sigma(\lambda, \gamma_i, \bar{\sigma}_V)(1 - \sigma(\lambda, \gamma_i, \bar{\sigma}_V))$$

Here is the *Stan* program I wrote to estimate the duplicate aversion model:

```
/* Duplicate aversion quantal response equilibrium model for the Volunteer's Dilemma
*/

functions {


  /* Here I am using "PBR" (i.e. probabilistic best response) to
  distinguish between \sigma and \bar\sigma_V
  */

  vector PBR(
```

```
  real lambda, vector gamma, real sigmaV,
  data real V, data real c, data real L, data real n
) {

  return inv_logit(lambda*(
    V-c-gamma*(1.0-pow(1.0-sigmaV,n-1.0))-V*(1.0-pow(1.0-sigmaV,n-1.0))+L*pow(1.0-sigmaV,n-1.0)
  ));

}


// derivative wrt sigmaV
vector PBR_sigmaV(
  real lambda, vector gamma, real sigmaV,
  data real V, data real c, data real L, data real n
) {
  return -lambda*(n-1.0)*pow(1.0-sigmaV,n-2.0)*(gamma+V-L)
  .*PBR(lambda,gamma,sigmaV,V,c,L,n) .*(1.0-PBR(lambda,gamma,sigmaV,V,c,L,n));
}


// Zero condition for equilibrium
real Hfun(real l,
  real lambda, vector gamma,
  data real V, data real c, data real L, data real n
  ) {
    real sigmaV = inv_logit(l);
    real meanPBR = mean(PBR(lambda,gamma,sigmaV,V,c,L,n));
    return l - log(meanPBR)+log(1.0-meanPBR);
  }

// derivative of zero condition
real Hl(real l,
  real lambda, vector gamma,
  data real V, data real c, data real L, data real n
  ) {
    real sigmaV = inv_logit(l);
    real meanPBR = mean(PBR(lambda,gamma,sigmaV,V,c,L,n));
    return 1.0-sigmaV*(1.0-sigmaV)/(meanPBR*(1.0-meanPBR))*mean(PBR_sigmaV(lambda,gamma,sigmaV,V,c,L,n
  }

// Compute QRE using constant-lambda corrector steps
real lpQRE(
  real lambda, vector gamma,
  data real V, data real c, data real L, data real n,
  data real tol, data int maxiter
) {

  // start at l=0 (sigmaV=0.5)
  real l = 0.0;

  for (it in 1:maxiter) {
```

```
      real dl = -Hfun(l,lambda,gamma,V,c,L,n)/Hl(l,lambda,gamma,V,c,L,n);

      l = l+dl;

      if (abs(dl)<tol) {
        break;
      }

      if (it==maxiter) {
        print("Maximum iterations error, lambda = ",lambda, ", dl = ",abs(dl));
      }

    }

    return inv_logit(l);

  }







}

data {
  int<lower=0> N; // Number of participants
  int<lower=0> Volunteer[N];
  int<lower=0> count[N];

  int<lower=0> nTreatments;
  vector[nTreatments] GroupSize;

  int<lower=0> GameID[N];


  // Game parameters
  real V; // Benefit of volunteering
  real c; // Cost of volunteering
  real L; // Benefit if nobody volunteers

  real prior_lambda[2]; // lognormal prior
  real prior_mu[2];
  real prior_tau;



  int<lower=0> maxiter;
  real<lower=0> tol; // tolerance for the corrector step

  int<lower=1> nSim; // simulation size for Monte Carlo integration
  vector<lower=-1.0,upper=1.0>[nSim] deltaZ; // standard normals truncated to be between -1 and 1
```

```
}

parameters {
  // logit choice precision
  real<lower=0> lambda;

  // Distribution of delta
  real mu;
  real<lower=0> tau;

  vector<lower = mu-tau,upper=mu+tau>[N] gamma_i;


}

model {



  vector[nSim] gammaSim = mu+tau*deltaZ;

  vector[nTreatments] sigmaV;

  for (tt  in 1:nTreatments) {
    sigmaV[tt] = lpQRE(
    lambda,gammaSim,
    V,c,L,GroupSize[tt],
    tol,maxiter
  );

  }

  Volunteer ~ binomial_logit(count,lambda*(
      V-c-gamma_i.*(1.0-pow(1.0-sigmaV[GameID],GroupSize[GameID]-1.0))-V*(1.0-pow(1.0-sigmaV[GameID],Gr
    ));

  gamma_i ~ normal(mu,tau);

  lambda ~ lognormal(prior_lambda[1],prior_lambda[2]);
  mu ~ normal(prior_mu[1],prior_mu[2]);
  tau ~ cauchy(0,prior_tau);

}

generated quantities {


  vector[nTreatments] sigmaV;
  {
  vector[nSim] gammaSim = mu+tau*deltaZ;
  for (tt  in 1:nTreatments) {
    sigmaV[tt] = lpQRE(
    lambda,gammaSim,
```

Table 32: Posterior moments from the heterogeneous models. Rows with n show predictions for the probability of volunteering for each group size

	Warm Glow		Duplicate aversion	
	mean	sd	mean	sd
λ	4.1356	1.1268	6.0312	1.5625
μ	-0.3643	0.0874	0.4723	0.1104
τ	0.5456	0.3621	0.7411	0.1916
n = 2	0.5236	0.0317	0.5619	0.0272
n = 3	0.4098	0.0201	0.4099	0.0199
n = 6	0.2814	0.0162	0.2552	0.0173
n = 9	0.2315	0.0186	0.1987	0.0186
n = 12	0.2048	0.0211	0.1683	0.0201
lp___	-2187.5674	254.7857	-2049.4131	16.4658

```
    V,c,L,GroupSize[tt],
    tol,maxiter
  );

  }
  }

}
```

14.2.4 Results

```
RowNames<-c("$\\lambda$","$\\mu$","$\\tau$","n = 2", "n = 3", "n = 6","n = 9", "n = 12","lp__")

WarmGlow<-summary(readRDS("Code/QRE2/Estimates_VDQREwarmglow.rds"))$summary |>
  data.frame() |>
  round(4) |>
  dplyr::select(mean,sd)

DuplicateAversion<-summary(readRDS("Code/QRE2/Estimates_VDQREduplicateAversion.rds"))$summary |>
  data.frame() |>
  round(4) |>
  dplyr::select(mean,sd)

rownames(WarmGlow)<-RowNames
rownames(DuplicateAversion)<-RowNames

cbind(WarmGlow,DuplicateAversion)  |> kbl(caption = "Posterior moments from the heterogeneous models. R
```

Table 32 shows the posterior estimates of the two heterogeneous-participants quantal response equilibrium models. These can be compared to the middle two panels of Table 5 of Goeree, Holt, and Smith (2017). While the models' fundamental parameters λ, μ, and τ are noticeably different, the predictions for $\bar{\sigma}_V$ for each group size are in agreement to within a couple of percentage points.

14.3 *R* code to run estimations

```
library(kableExtra)
library(tidyverse)
```

```r
library(mnorm)
library(rstan)
  rstan_options(auto_write = TRUE)
  options(mc.cores = parallel::detectCores())
  rstan_options(threads_per_chain = 1)

D<-read.csv("Data/GHS2017VD.csv") |>
    # This dataset does not explicitly store the group size
    # I can infer it from the "Other.Id." variable, but
    # this is not consistent with the experiment description
    # in the "Procedures" section of the paper. Specifically,
    # it looked like there were two GroupSize=7 session, when
    # in reality there should have been one with 9, and one
    # with 12. This fixes things
    mutate(GroupSize = str_count(Other.Id., "ID")+1,
           GroupSize = ifelse(SessionID==10,9,GroupSize),
           GroupSize = ifelse(SessionID==11,12,GroupSize)
    ) |>
    rename(Volunteer = Decision..1.vol..) |>
    mutate(ID = paste0(SessionID,"-",ID) |> as.factor() |> as.numeric() ) |>
    select(ID,GroupSize,Volunteer) |>
    # all I need are the counts of actions for each participant:
    group_by(ID,GroupSize) |>
    summarize(Volunteer = sum(Volunteer),count=n())

##########################################################
# Homogeneous model
##########################################################

file<-"Code/QRE2/Estimates_VDQREhomogeneous.rds"
if (!file.exists(file)) {
  model<-stan_model("Code/QRE2/VDQREhomogeneous.stan")

  d<-D |>
    group_by(GroupSize) |>
    summarize(Volunteer = sum(Volunteer),
              count = sum(count))

  dStan<-list(
    N=dim(d)[1],
    Volunteer = d$Volunteer,
    count = d$count,
    GroupSize = d$GroupSize,
    V = 1.0,
    c = 0.2,
    L = 0.2,
    prior_lambda = c(log(5),1),

    first_step = 0.1,
    min_step = 1e-4,
    max_decel = 1.1,
    maxiter = 1000,
    tol = 1e-4
```

```
  )

  Fit<-sampling(model,data=dStan,seed=42,chains=4,iter=2000
                )
  Fit |> saveRDS(file)
}

##############################################
# Warm glow volunteering heterogeneous model
##############################################


file<-"Code/QRE2/Estimates_VDQREwarmglow.rds"
if (!file.exists(file)) {
  model<-stan_model("Code/QRE2/VDQREwarmglow.stan")
  #stop()

  D<-D |>
    mutate(
      GameID = ifelse(
        GroupSize==2,1,ifelse(
          GroupSize==3,2,ifelse(
            GroupSize==6,3,ifelse(
              GroupSize==9,4,5
            )
          )
        )
      )
    )


  dStan<-list(
    N=dim(D)[1],
    Volunteer = D$Volunteer,
    count = D$count,

    nTreatments = 5,
    GroupSize = c(2,3,6,9,12),

    GameID = D$GameID,

    V = 1.0,
    c = 0.2,
    L = 0.2,
    prior_lambda = c(log(5),1),
    prior_mu = c(0,1),
    prior_tau = 0.05,

    maxiter = 1000,
    tol = 1e-4,

    nSim = 100,
    # This gets me Halton draws, transformed to be from the
```

```
    # standard normal, truncated to lie between -1 and 1
    deltaZ = (pnorm(1)+halton(100)*(1-2*pnorm(1))) |> qnorm() |> as.vector()
  )

  Fit<-sampling(model,data=dStan,seed=42,chains=4,iter=2000,
                control=list(adapt_delta = 0.8,max_treedepth=15),
                pars = "delta_i",include=FALSE
  )
  Fit |> saveRDS(file)
}

##############################################
# Duplicate aversion heterogeneous model
##############################################



file<-"Code/QRE2/Estimates_VDQREduplicateaversion.rds"
if (!file.exists(file)) {
  model<-stan_model("Code/QRE2/VDQREduplicateaversion.stan")
  #stop()

  D<-D |>
    mutate(
      GameID = ifelse(
        GroupSize==2,1,ifelse(
          GroupSize==3,2,ifelse(
            GroupSize==6,3,ifelse(
              GroupSize==9,4,5
            )
          )
        )
      )
    )


  dStan<-list(
    N=dim(D)[1],
    Volunteer = D$Volunteer,
    count = D$count,

    nTreatments = 5,
    GroupSize = c(2,3,6,9,12),

    GameID = D$GameID,

    V = 1.0,
    c = 0.2,
    L = 0.2,
    prior_lambda = c(log(5),1),
    prior_mu = c(0,1),
    prior_tau = 0.05,
```

```
    maxiter = 1000,
    tol = 1e-4,

    nSim = 100,
    # This gets me Halton draws, transformed to be from the
    # standard normal, truncated to lie between -1 and 1
    deltaZ = (pnorm(1)+halton(100)*(1-2*pnorm(1))) |> qnorm() |> as.vector()
  )
  Fit<-sampling(model,data=dStan,seed=42,chains=4,iter=2000,
                control=list(adapt_delta = 0.8,max_treedepth=15),
                pars = "gamma_i",include=FALSE
  )
  Fit |> saveRDS(file)



}
```

15 Application: A Quantal Response Equilibrium with discrete types

In this chapter, I would like to show you how to estimate a quantal response equilibrium model with discrete player types. This is in contrast to the Volunteer's Dilemma chapter, where I implemented the *continuous*-type models studied in Goeree, Holt, and Smith (2017). The distinction between discrete and continuous types can become a bit blurred when there are a lot of types in a model. For example in auctions, we often assume a continuous distribution of values (i.e. types), but when we implement this in the lab we must have a discrete distribution, albeit with a lot of types. Here I am going to show you a model with just two types, so that you can hopefully see the way we solve for the model, then construct the likelihood.

15.1 Example dataset and models

This dataset is particularly special to me, because it is from my job market paper (Bland 2019a)! In it, I was interested in whether choice bracketing was important in games. Choice bracketing is a phenomenon where people who face more than one decision to make partition their decisions into more than one choice set, and then make decisions for each element of these partitions sepatately. To fix ideas, Table 33 shows the two games in the baseline treatment that participants played. These games were played simultaneously and against the same opponenet. Payoffs from both games determined the participants' earnings. However game theory and *broad bracketing* (the standard, "rational" assumption in economics) assumes that players will understand that the payoffs will be added up, and so will consider these two "games" to actually be one big game, shown in Table 34. Note that the actions in this game are the *combinations* of actions that a player could play in the two games in Table 33.

```
u1<- rbind(
  c(10,10),c(100,0)
)
U1<-paste0(u1,", ",u1 |> t()) |> matrix(nrow = 2)
rownames(U1)<-c("A","B")
colnames(U1)<-c("A","B")

u2<- rbind(
  c(56,56),c(160,0)
)
U2<-paste0(u2,", ",u2 |> t()) |> matrix(nrow = 2)
```

Table 33: The two games in the baseline treatment, narrow presentation

	Game 1			Game 2	
	A	B		C	D
A	10, 10	10, 100	C	56, 56	56, 160
B	100, 10	0, 0	D	160, 56	0, 0

Table 34: The 4×4 game when players add up the payoffs of the two 2×2 games

	AC	AD	BC	BD
AC	66, 66	66, 170	66, 156	66, 260
AD	170, 66	10, 10	170, 156	10, 100
BC	156, 66	156, 170	56, 56	56, 160
BD	260, 66	100, 10	160, 56	0, 0

```
U2<-cbind(c("C","D"),U2)
colnames(U2)<-c("","C","D")


cbind(U1,U2) |>
  kbl(caption = "The two games in the baseline treatment, narrow presentation") |>
  kable_classic(full_width=FALSE) |>
  add_header_above(c("","Game 1"=2,"Game 2"=3))
```

```
Ub<-rbind(
  c(66,66,66,66),
  c(170,10,170,10),
  c(156,156,56,56),
  c(260,100,160,0)
)

U<-paste0(Ub,", ",t(Ub)) |> matrix(nrow=4)

colnames(U)<-c("AC","AD","BC","BD")
rownames(U)<-c("AC","AD","BC","BD")

U |>
  kbl(caption="The $4\\times 4$ game when players add up the payoffs of the two $2\\times 2$ games") |>
  kable_classic(full_width=FALSE)
```

My research question was: do people see these as two separate games (i.e. Table 33), or do they integrate the payoffs together (i.e. Table 34)? I did this with a 3-treatment design that varied payoffs in a way that would not affect players' actions under assumptions of broad or narrow bracketing. The way I achieved this was that in a second treatment I added 50 points to all the payoffs in Game 1 compared to the baseleine, and in a third treatment I added 50 points to all the payoffs in Game 2. If players bracketed narrowly then there should be no difference in choice frequencies in Game 1 between the baseline and Treatment 3, and in Game 2 between the baseline and Treatment 2. On the other hand, if people broadly bracketed there should be no difference in choice frequencies between Treatments 2 and 3. Unfortunately, I was able to reject *both* assumptions: the aggregate data didn't look like narrow bracketing, but it didn't look like broad bracketing either!

Fortunately, I wasn't done analyzing me data. One of the structural analyses I did was estimate some Quantal Response Equilibrium models that assumed either broad bracketing, narrow bracketing, or a mixture of

both.[67] That way, I could ask which model *best* organized the data, rather than rejecting both assumptions against a generic alternative assumption of "something else".

All of the quantal response equilibrium models I estimated assume a common choice precision $\lambda > 0$ and common risk-aversion parameter $r > 0$, but differ in their assumptions about how payoffs are calculated. I will start by walking you through the models that assume only broad or narrow bracketing, and then use these to build up to a model where there is a mixture of broad-bracketing and narrow-bracketing participants in the sample.

15.2 A note on replication

What I am about to show you is most certainty *not* an attempt to replicate the quantal response equilibrium analysis of Bland (2019a). Aside from using very different priors for the model's parameters (which I suspect wouldn't have changed things too much), I am using a very differ technique to compute quantal response equilibrium. In particular, in my job market paper, I use the *empirical payoff* technique, which is a computational shortcut that avoids having the solve for the equilibrium at all. However as shown in Bland and Turocy (2025), this technique can produce unreliable results because it can be fitting data to a very different curve than the actual quantal response equilibrium. Instead, I am using the *equilibrium correspondence* approach, which is what I have been teaching you in the previous quantal response equilibrium chapters. Had I known about the issues of using the empirical payoff approach when I wrote my job market paper, I certainly would not have used it, especially since it is fairly easy to compute the equilibrium in this situation. All of this is to say, don't expect what follows to be an exact replication of Bland (2019a).

15.3 Three models that make different assumptions about bracketing

15.3.1 Broad bracketing only

This is probably the most standard Quantal Response Equilibrium model. Here we are going to derive the system of equations that characterize the equilibrium as a function of λ and r, and then propose a method for solving for the equilibrium mixed strategies iteratively.

First, if U^r is the payoff matrix of the row player (in Table 34) raised element-wise to the power of r, and ρ is vector of the logged mixed strategy, then we can write the equilibrium conditions in log differences as follows:

$$0 = H_a(\rho; \lambda, r) \doteq \rho_a - \rho_1 - \lambda[U_a^r - U_1^r] \quad \text{for } a \in \{2, 3, 4\}$$

and then we need one more constraint to ensure that the probabilities add up to one:

$$0 = H_1(\rho; \lambda, r) = 1 - \sum_{a=1}^{4} \exp(\rho_a)$$

We can then write these in matrix notation as:

$$H(\rho; \lambda, r) = A\rho - \lambda A U^r \exp(\rho) - B \exp(\rho) + c$$

where:

$$A = \begin{bmatrix} -1 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ -1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \quad B = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}, \quad c = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

[67] I thank an anonymous referee for requesting some QRE analysis in this paper. At the time, QRE was a passing interest of mine, but I hadn't done any research using it. This was a very helpful nudge!

we can then solve for iterativelythe equilibrium (log) mixed strategy iteratively using Newton's method:

$$\rho^{t+1} = \rho^t - J^{-1}(\rho^t; \lambda, r)H(\rho^t; \lambda, r)$$

Where J is the Jacobian of the system with respect to ρ:

$$J(\rho; \lambda, r) = A - (\lambda A U^r + B)\text{diag\_matrix}(\exp(\rho))$$

For many games this won't be enough to find an equilibrium, and you should then go and use the full predictor-corrector method described earlier in this book and in Bland and Turocy (2025), but in this instance I found that it worked well.

Once we have the log equilibrium mixed strategy ρ, this is exactly the quantity we need to increment the likelihood: just add ρ_a to it every time a participant chooses action a.

Here is the *Stan* program I wrote to estimate this model. One thing to note is that there is a ρ for every treatment (there were three treatments) becuase the payoffs change between treatments.

```
functions {

  vector QRElp(
    matrix U, real lambda,
    data matrix A, data matrix B, data vector c,
    data real tol, data int iter
  ) {

    // returns the log QRE probabilities

    int nactions = dims(U)[1];

    // initialization
    vector[nactions] rho = rep_vector(log(1.0/nactions),nactions);


    for (tt in 1:iter) {


      vector[nactions] H = A*rho-(lambda*A*U+B)*exp(rho)+c;

      matrix[nactions,nactions] J = A-(lambda*A*U+B)*diag_matrix(exp(rho));
      // change in rho
      vector[nactions] drho  = -J\H;

      rho += drho;

      //print(max(abs(H)));
      //print(rho);
      //print("---------------");

      if (max(abs(H))<tol) {
        break;
      }
```

```
      if (tt==iter && max(abs(H))>tol) {

        print("maximum number of iterations reached without reaching tolerance");

        print(max(abs(H)));

      }


    }

    return rho;
  }


}

data {


  // payoff matrices for the 3 treatments
  matrix[4,4] Ubroad[3];

  int N; // number of observations

  int treatment[N];

  // coded as 1=AC, 2=AD, 3=BC, 4=BD
  int action[N];

  vector[2] prior_r;

  vector[2] prior_lambda;


  real<lower=0> tol;
  int<lower=1> iter;

  int<lower=0,upper=1> UseData;

  int<lower=0,upper=1> contextual;



}

transformed data {

  matrix[4,4] A_broad = [
    [-1,1,0,0],
    [-1,0,1,0],
    [-1,0,0,1],
    [0,0,0,0]
```

```
  ];

  matrix[4,4] B_broad = [
    [0,0,0,0],
    [0,0,0,0],
    [0,0,0,0],
    [1,1,1,1]
  ];

  vector[4] c_broad = [0,0,0,1]';


}

parameters {

  // risk aversion
  real<lower=0> r;
  // choice precision
  real<lower=0> lambda;



}

transformed parameters {

  vector[4] rho[3];

  for (tt in 1:3) {

    matrix[4,4] U = pow(Ubroad[tt],r);

    if (contextual==1) {
      // contextual utility normalization
      U = U/(U[4,1]-U[4,4]);
    }

    rho[tt] = QRElp(
      U, lambda,
      A_broad, B_broad, c_broad,
      tol, iter
    );

  }


}

model {


  // likelihood contribution
```

```
  if (UseData==1) {
  for (ii in 1:N) {

    target += rho[treatment[ii]][action[ii]];

  }
  }


  // priors

  target += lognormal_lpdf(r | prior_r[1],prior_r[2]);
  target += lognormal_lpdf(lambda | prior_lambda[1],prior_lambda[2]);

}

generated quantities {

  vector[4] sigma[3];

  for (tt in 1:3) {

    sigma[tt] = exp(rho[tt]);

  }


}
```

15.3.2 Narrow bracketing only

Here we need to compute two equilibrium mixed strategies per treatment. One for Game 1 and one for Game 2. The steps for doing this are similar to that for the broad bracketing assumption, except that we have two log mixed strategies to keep track of, which I call ρ^1 and ρ^2. For each game, we can write the system of eqations describing the equilibrium as:

$$H(\rho^g;\lambda,r) = A\rho^g - \lambda A(U^g)^r \exp(\rho^g) - B\exp(\rho^g) + c, \quad g \in \{1,2\}$$
$$\text{where: } A = \begin{bmatrix} -1 & 1 \\ 0 & 0 \end{bmatrix}, \quad B = \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix}, \quad c = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

So it is basically the same process to solve for the quantal response equilibrium compared to the broad bracketing model.

One additional assumption I had to make here was about dependence between actions in the two games. While broad bracketing implies a probability distribution over all *four* combinations of actions, narrow bracketing only says something about the *marginal* strategy profiles. Here I assumed that they were independent, so the log mixed strategy would be (since multiplying probability levels is the same as adding logged probabilities):

$$\rho = \begin{pmatrix} \rho_A^1 + \rho_C^2 \\ \rho_A^1 + \rho_D^2 \\ \rho_B^1 + \rho_C^2 \\ \rho_B^1 + \rho_D^2 \end{pmatrix}$$

Here is the *Stan* program I wrote to estimate this model:

```
functions {

  vector QRElp(
    matrix U, real lambda,
    data matrix A, data matrix B, data vector c,
    data real tol, data int iter
  ) {

    // returns the log QRE probabilities

    int nactions = dims(U)[1];

    // initialization
    vector[nactions] rho = rep_vector(log(1.0/nactions),nactions);


    for (tt in 1:iter) {


      vector[nactions] H = A*rho-(lambda*A*U+B)*exp(rho)+c;

      matrix[nactions,nactions] J = A-(lambda*A*U+B)*diag_matrix(exp(rho));
      // change in rho
      vector[nactions] drho  = -J\H;

      rho += drho;

      //print(max(abs(H)));
      //print(rho);
      //print("---------------");

      if (max(abs(H))<tol) {
        break;
      }

      if (tt==iter && max(abs(H))>tol) {

        print("maximum number of iterations reached without reaching tolerance");

        print(max(abs(H)));

      }


    }

    return rho;
  }


}
```

```
data {


  // payoff matrices for the 3 treatments
  matrix[2,2] Unarrow1[3];
  matrix[2,2] Unarrow2[3];

  int N; // number of observations

  int treatment[N];

  // coded as 1=AC, 2=AD, 3=BC, 4=BD
  int action[N];

  vector[2] prior_r;

  vector[2] prior_lambda;


  real<lower=0> tol;
  int<lower=1> iter;

  int<lower=0,upper=1> UseData;

  int<lower=0,upper=1> contextual;



}

transformed data {

  matrix[2,2] A_narrow = [
    [-1,1],
    [0,0]
  ];

  matrix[2,2] B_narrow = [
    [0,0],
    [1,1]
  ];

  vector[2] c_narrow = [0,1]';


}

parameters {

  // risk aversion
  real<lower=0> r;
  // choice precision
  real<lower=0> lambda;
```

```
}

transformed parameters {

  vector[2] rho1[3];
  vector[2] rho2[3];

  vector[4] rho[3];

  for (tt in 1:3) {

    matrix[2,2] U1 = pow(Unarrow1[tt],r);
    matrix[2,2] U2 = pow(Unarrow2[tt],r);

    if (contextual==1) {
      // contextual utility normalization
     U1 = U1/(U1[2,1]-U1[2,2]);
     U2 = U2/(U2[2,1]-U2[2,2]);
    }

    rho1[tt] = QRElp(
      U1, lambda,
      A_narrow, B_narrow, c_narrow,
      tol, iter
    );

    rho2[tt] = QRElp(
      U2, lambda,
      A_narrow, B_narrow, c_narrow,
      tol, iter
    );

    rho[tt] = to_vector(log(exp(rho1[tt])*exp(rho2[tt]')));

  }


}

model {


  // likelihood contribution
  if (UseData==1) {
  for (ii in 1:N) {

    target += rho[treatment[ii]][action[ii]];

  }
  }
```

```
  // priors

  target += lognormal_lpdf(r | prior_r[1],prior_r[2]);
  target += lognormal_lpdf(lambda | prior_lambda[1],prior_lambda[2]);

}

generated quantities {

  vector[2] sigma1[3];
  vector[2] sigma2[3];

  vector[4] sigma[3];

  for (tt in 1:3) {
    sigma1[tt] = exp(rho1[tt]);
    sigma2[tt] = exp(rho2[tt]);
    sigma[tt] = exp(rho[tt]);

  }


}
```

15.3.3 A mixture of broad and narrow bracketing

When we have more than one thing that people could be doing, it is possible (and often likely) that all of the things are important. I therefore also estimate a quantal response equilibrium model that assumes fraction $\phi \in (0,1)$ of participants narrowly bracket, and fraction $1-\phi$ broadly bracket. That is, we have two *types* of player in our game. If we weren't doing quantal response equilibrium, we would be looking to solve a *Bayesian* Nash equilibrium. For quantal response equilibrium, while conceptually this is not too hard to implement, some care needs to be taken when writing down the equilibrium conditions. For this, we will need a set of equations for the narrow-bracketing type, a set for the broad-bracketing type, and a description of how the types' mixed strategies relate to the aggregate mixed strategy (when we integrate out the types). By the "aggregate mixed strategy" here I mean the average choice probabilities implied by the mixing probability ϕ and the mixed strategies played by each type. In this kind of game this is the quantity players need to know in order to evaluate their expected utilities (irrespective of whether they are broad or narrow bracketers).

Let ρ be the log probabilities of the broad and narrow actions:

$$\rho = \begin{pmatrix} \rho^B_{AC} & \rho^B_{AD} & \rho^B_{BC} & \rho^B_{BD} & \rho^N_A & \rho^N_B & \rho^N_C & \rho^N_D \end{pmatrix}^\top$$

We can write the aggregate mixed strategy (in levels, rather than logs) from a broad bracketer's perspective as:

$$\sigma^{B*} = \begin{pmatrix} (1-\phi)\exp(\rho^B_{AC}) + \phi\exp(\rho^N_A + \rho^N_C) \\ (1-\phi)\exp(\rho^B_{AD}) + \phi\exp(\rho^N_A + \rho^N_D) \\ (1-\phi)\exp(\rho^B_{BC}) + \phi\exp(\rho^N_B + \rho^N_C) \\ (1-\phi)\exp(\rho^B_{BD}) + \phi\exp(\rho^N_B + \rho^N_D) \end{pmatrix}$$

In order to evaluate the Jacobian, we need the derivative of this, which is:

$$\frac{\partial \sigma^{B*}}{\partial \rho^\top} = \begin{bmatrix} (1-\phi)\exp(\rho^B_{AC}) & 0 & 0 & 0 & \phi\exp(\rho^N_A + \rho^N_C) & 0 & \phi \\ 0 & (1-\phi)\exp(\rho^B_{AD}) & 0 & 0 & \phi\exp(\rho^N_A + \rho^N_D) & 0 & \\ 0 & 0 & (1-\phi)\exp(\rho^B_{BC}) & 0 & 0 & \phi\exp(\rho^N_B + \rho^N_C) & \phi \\ 0 & 0 & 0 & (1-\phi)\exp(\rho^B_{BD}) & 0 & \phi\exp(\rho^N_B + \rho^N_D) & \end{bmatrix}$$

Then we can write the aggregate mixed strategy from a narrow bracketer's perspective as:

$$\sigma^{N*} = \begin{pmatrix} \sigma^{B*}_{AC} + \sigma^{B*}_{AD} \\ \sigma^{B*}_{BC} + \sigma^{B*}_{BD} \\ \sigma^{B*}_{AC} + \sigma^{B*}_{BC} \\ \sigma^{B*}_{AD} + \sigma^{B*}_{BD} \end{pmatrix}$$

Then to get the Jacobian we can write:

$$\frac{\partial \sigma^{N*}}{\partial \rho^\top} = \frac{\partial \sigma^{N*}}{\partial \sigma^{B*\top}} \frac{\partial \sigma^{B*}}{\partial \rho^\top}, \quad \text{where } \frac{\partial \sigma^{N*}}{\partial \sigma^{B*\top}} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

We can then write the system of equations describing the equilibrium as:

$$\begin{aligned} H_{1:4}(\rho; r, \phi, \lambda) &= A^B \rho^B - \lambda A^B (U^B)^r \sigma^{B*} - B^B \exp(\rho^B) + c^B \\ H_{5:6}(\rho; r, \phi, \lambda) &= A^N \rho^N_1 - \lambda A^N (U^N_1)^r \sigma^{N*}_1 - B^N \exp(\rho^N_1) + c^N \\ H_{7:8}(\rho; r, \phi, \lambda) &= A^N \rho^N_2 - \lambda A^N (U^N_2)^r \sigma^{N*}_2 - B^N \exp(\rho^N_2) + c^N \end{aligned}$$

Where A^B is the A matrix defined for the broad-only model, A^N is the A matrix defined for the narrow-only model, and so on.

Now that we can set up and solve the system of equations characterizing the quantal response equilibrium mixed strategies, we need to take these mixed strategies and compute the likelihod. Here, becuase participants made more than one decision in the experiment, we could make (at least) two different assumptions about how the likelihood is calculated. These are (1) the *dichotomous* mixture model assumption, and (2) the *toolbox* mixture assumption. The dichotomous assumption in this model is that participants are either broad *or* narrow bracketers, and *always* make their decisions according to that type. In this case, we compute the log-likelihood contribution for participant i as follows:

$$\log p(y_i \mid \rho^B, \rho^N, \phi) = \log\left(\phi \exp\left(y_i^\top \rho^N\right) + (1-\phi)\exp\left(y_i^\top \rho^B\right)\right)$$

where y_i is a vector of counts of the four combinations of actions.

On the other hand, the toolbox assumption states that participants bracket narrowly for each *decision* with probability ϕ, and so we write the likelihood as follows:

$$\log p(y_i \mid \rho^B, \rho^N, \phi) = y_i^\top \log\left(\phi \exp(\rho^N) + (1-\phi)\exp(\rho^B)\right)$$

Here is the *Stan* program I wrote to estimate the dichotomous model:

```
functions {
```

```
vector QRElp(
  matrix Ub, matrix U1n, matrix U2n,
  real lambda, real mix_n,
  data int iter, data real tol
) {


 matrix[4,4] Ab = [
  [-1,1,0,0],
  [-1,0,1,0],
  [-1,0,0,1],
  [0,0,0,0]
];

matrix[4,4] Bb = [
  [0,0,0,0],
  [0,0,0,0],
  [0,0,0,0],
  [1,1,1,1]
];

vector[4] cb = [0,0,0,1]';

 matrix[2,2] An = [
  [-1,1],
  [0,0]
];

matrix[2,2] Bn = [
  [0,0],
  [1,1]
];



vector[2] cn = [0,1]';


matrix[2,4] Dsigma1n = [
  [1,1,0,0],
  [0,0,1,1]
];
matrix[2,4] Dsigma2n = [
  [1,0,1,0],
  [0,1,0,1]
];

// initialization
vector[8] rho;
    rho[1:4] = rep_vector(log(0.25),4);
    rho[5:8] = rep_vector(log(0.50),4);
```

```
// START LOOP HERE-----------------------------------------------------------

for (tt in 1:iter) {

// aggregate mixed strategy, from the broad bracketers' perspective
vector[4] SIGMAb = (1-mix_n)*exp(rho[1:4]);
   SIGMAb[1] += mix_n*exp(rho[5]+rho[7]);
   SIGMAb[2] += mix_n*exp(rho[5]+rho[8]);
   SIGMAb[3] += mix_n*exp(rho[6]+rho[7]);
   SIGMAb[4] += mix_n*exp(rho[6]+rho[8]);

// aggregate mixed strategy, from the narrow bracketers' perspective
vector[2] SIGMA1n;
   SIGMA1n[1] = SIGMAb[1]+SIGMAb[2];
   SIGMA1n[2] = SIGMAb[3]+SIGMAb[3];
vector[2] SIGMA2n;
   SIGMA2n[1] = SIGMAb[1]+SIGMAb[3];
   SIGMA2n[2] = SIGMAb[2]+SIGMAb[4];




vector[4] rhob = rho[1:4];
vector[2] rho1n = rho[5:6];
vector[2] rho2n = rho[7:8];

vector[8] H;
   H[1:4] = Ab*rhob-lambda*Ab*Ub*SIGMAb-Bb*exp(rhob)+cb;
   H[5:6] = An*rho1n-lambda*An*U1n*SIGMA1n-Bn*exp(rho1n)+cn;
   H[7:8] = An*rho2n-lambda*An*U2n*SIGMA2n-Bn*exp(rho2n)+cn;


matrix[4,8] DSIGMAb = [
  [(1-mix_n)*exp(rhob[1]),0,                    0,                    0,                    mix_n*exp
  [0,                     (1-mix_n)*exp(rho[2]),0,                    0,                    mix_n*exp
  [0,                     0,                    (1-mix_n)*exp(rho[3]),0,                    0 ,
  [0,                     0,                    0,                    (1-mix_n)*exp(rho[4]),0,
];


matrix[8,8] J = rep_matrix(0.0,8,8);
   J[1:4,1:4] += Ab-Bb*diag_matrix(exp(rho[1:4]));
   J[5:6,5:6] += An-Bn*diag_matrix(exp(rho[5:6]));
   J[7:8,7:8] += An-Bn*diag_matrix(exp(rho[7:8]));

   J[1:4,] += -lambda*Ab*Ub*DSIGMAb;
   J[5:6,] += -lambda*An*U1n*Dsigma1n*DSIGMAb;
   J[7:8,] += -lambda*An*U2n*Dsigma2n*DSIGMAb;


vector[8] drho = -J\H;

rho += drho;
```

```
   if (max(abs(H))<tol) {

     break;
   }

   if (tt==iter && max(abs(H))>tol) {

     print("Maximum iterations reached without achieveing tolerance");

   }

   }

   return rho;

  }









}

data {


  // payoff matrices for the 3 treatments
  matrix[4,4] Ubroad[3];
  matrix[2,2] Unarrow1[3];
  matrix[2,2] Unarrow2[3];

  int N; // number of observations

  int id[N];

  int nparticipants;

  int treatment[N];

  // coded as 1=AC, 2=AD, 3=BC, 4=BD
  int action_broad[N];

  vector[2] prior_r;

  vector[2] prior_lambda;

  vector[2] prior_mix;
```

```
  real<lower=0> tol;
  int<lower=1> iter;

  int<lower=0,upper=1> UseData;

  int<lower=0,upper=1> contextual;



}

transformed data {




  vector[4] rho0 = rep_vector(log(1.0/4.0),4);


  int action_narrow1[N];
  int action_narrow2[N];

  for (ii in 1:N) {

    action_narrow1[ii] = action_broad[ii]==1 || action_broad[ii]==2 ? 1 : 2;
    action_narrow2[ii] = action_broad[ii]==1 || action_broad[ii]==3 ? 1 : 2;

  }


}

parameters {

  // risk aversion
  real<lower=0> r;
  // choice precision
  real<lower=0> lambda;

  real<lower=0,upper=1> mix_narrow;



}

transformed parameters {

  vector[8] rho[3];

  vector[4] sigma_broad[3];
  vector[2] sigma_narrow1[3];
  vector[2] sigma_narrow2[3];
```

```
  for (tt in 1:3) {

    matrix[4,4] Ub = pow(Ubroad[tt],r);
    matrix[2,2] U1n = pow(Unarrow1[tt],r);
    matrix[2,2] U2n = pow(Unarrow1[tt],r);

    if (contextual==1) {
      // contextual utility normalization
      Ub = Ub/(Ub[4,1]-Ub[4,4]);
      U1n = U1n/(U1n[2,1]-U1n[2,2]);
      U2n = U2n/(U2n[2,1]-U2n[2,2]);
    }

    rho[tt] = QREIp(
      Ub,U1n, U2n,
      lambda,mix_narrow,
      iter,tol
  );



    sigma_broad[tt] = exp(rho[tt][1:4]);

    sigma_narrow1[tt] = exp(rho[tt][5:6]);
    sigma_narrow2[tt] = exp(rho[tt][7:8]);

  }


}

model {




  // likelihood contribution
  if (UseData==1) {
    vector[nparticipants] ll_narrow = rep_vector(log(mix_narrow),nparticipants);
    vector[nparticipants] ll_broad = rep_vector(log(1-mix_narrow),nparticipants);

    for (ii in 1:N) {

      ll_broad[id[ii]] += log(sigma_broad[treatment[ii]])[action_broad[ii]];

      ll_narrow[id[ii]] += log(sigma_narrow1[treatment[ii]])[action_narrow1[ii]];
      ll_narrow[id[ii]] += log(sigma_narrow2[treatment[ii]])[action_narrow2[ii]];

    }

    for (ii in 1:nparticipants) {
      target += log_sum_exp(ll_broad[ii],ll_narrow[ii]);
```

```
    }
  }


  // priors

  target += lognormal_lpdf(r | prior_r[1],prior_r[2]);
  target += lognormal_lpdf(lambda | prior_lambda[1],prior_lambda[2]);
  target += beta_lpdf(mix_narrow | prior_mix[1],prior_mix[2]);

}

generated quantities {

  vector[4] sigma[3];

  for (tt in 1:3) {

    sigma[tt] = exp(rho[tt]);

  }


}
```

And is the *Stan* program I wrote to estimate the toolbox model:

```
functions {


  vector QRElp(
    matrix Ub, matrix U1n, matrix U2n,
    real lambda, real mix_n,
    data int iter, data real tol
  ) {


   matrix[4,4] Ab = [
    [-1,1,0,0],
    [-1,0,1,0],
    [-1,0,0,1],
    [0,0,0,0]
  ];

  matrix[4,4] Bb = [
    [0,0,0,0],
    [0,0,0,0],
    [0,0,0,0],
    [1,1,1,1]
  ];

  vector[4] cb = [0,0,0,1]';
```

```
 matrix[2,2] An = [
   [-1,1],
   [0,0]
];

matrix[2,2] Bn = [
   [0,0],
   [1,1]
];



vector[2] cn = [0,1]';


matrix[2,4] Dsigma1n = [
   [1,1,0,0],
   [0,0,1,1]
];
matrix[2,4] Dsigma2n = [
   [1,0,1,0],
   [0,1,0,1]
];

// initialization
vector[8] rho;
    rho[1:4] = rep_vector(log(0.25),4);
    rho[5:8] = rep_vector(log(0.50),4);


 // START LOOP HERE----------------------------------------------------------------

 for (tt in 1:iter) {

 // aggregate mixed strategy, from the broad bracketers' perspective
 vector[4] SIGMAb = (1-mix_n)*exp(rho[1:4]);
    SIGMAb[1] += mix_n*exp(rho[5]+rho[7]);
    SIGMAb[2] += mix_n*exp(rho[5]+rho[8]);
    SIGMAb[3] += mix_n*exp(rho[6]+rho[7]);
    SIGMAb[4] += mix_n*exp(rho[6]+rho[8]);

 // aggregate mixed strategy, from the narrow bracketers' perspective
 vector[2] SIGMA1n;
    SIGMA1n[1] = SIGMAb[1]+SIGMAb[2];
    SIGMA1n[2] = SIGMAb[3]+SIGMAb[3];
 vector[2] SIGMA2n;
    SIGMA2n[1] = SIGMAb[1]+SIGMAb[3];
    SIGMA2n[2] = SIGMAb[2]+SIGMAb[4];



 vector[4] rhob = rho[1:4];
```

```
  vector[2] rho1n = rho[5:6];
  vector[2] rho2n = rho[7:8];

  vector[8] H;
     H[1:4] = Ab*rhob-lambda*Ab*Ub*SIGMAb-Bb*exp(rhob)+cb;
     H[5:6] = An*rho1n-lambda*An*U1n*SIGMA1n-Bn*exp(rho1n)+cn;
     H[7:8] = An*rho2n-lambda*An*U2n*SIGMA2n-Bn*exp(rho2n)+cn;


  matrix[4,8] DSIGMAb = [
    [(1-mix_n)*exp(rhob[1]),0,                    0,                    0,                    mix_n*exp
    [0,                     (1-mix_n)*exp(rho[2]),0,                    0,                    mix_n*exp
    [0,                     0,                    (1-mix_n)*exp(rho[3]),0,                    0 ,
    [0,                     0,                    0,                    (1-mix_n)*exp(rho[4]),0,
  ];


  matrix[8,8] J = rep_matrix(0.0,8,8);
     J[1:4,1:4] += Ab-Bb*diag_matrix(exp(rho[1:4]));
     J[5:6,5:6] += An-Bn*diag_matrix(exp(rho[5:6]));
     J[7:8,7:8] += An-Bn*diag_matrix(exp(rho[7:8]));

     J[1:4,] += -lambda*Ab*Ub*DSIGMAb;
     J[5:6,] += -lambda*An*U1n*Dsigma1n*DSIGMAb;
     J[7:8,] += -lambda*An*U2n*Dsigma2n*DSIGMAb;


  vector[8] drho = -J\H;

  rho += drho;

  if (max(abs(H))<tol) {

    break;
  }

  if (tt==iter && max(abs(H))>tol) {

    print("Maximum iterations reached without achieveing tolerance");

  }

  }

  return rho;

}
```

```
}

data {


  // payoff matrices for the 3 treatments
  matrix[4,4] Ubroad[3];
  matrix[2,2] Unarrow1[3];
  matrix[2,2] Unarrow2[3];

  int N; // number of observations

  int id[N];

  int nparticipants;

  int treatment[N];

  // coded as 1=AC, 2=AD, 3=BC, 4=BD
  int action_broad[N];

  vector[2] prior_r;

  vector[2] prior_lambda;

  vector[2] prior_mix;


  real<lower=0> tol;
  int<lower=1> iter;

  int<lower=0,upper=1> UseData;

  int<lower=0,upper=1> contextual;



}

transformed data {




  vector[4] rho0 = rep_vector(log(1.0/4.0),4);


  int action_narrow1[N];
  int action_narrow2[N];
```

```
  for (ii in 1:N) {

    action_narrow1[ii] = action_broad[ii]==1 || action_broad[ii]==2 ? 1 : 2;
    action_narrow2[ii] = action_broad[ii]==1 || action_broad[ii]==3 ? 1 : 2;

  }


}

parameters {

  // risk aversion
  real<lower=0> r;
  // choice precision
  real<lower=0> lambda;

  real<lower=0,upper=1> mix_narrow;



}

transformed parameters {

  vector[8] rho[3];

  vector[4] sigma_broad[3];
  vector[2] sigma_narrow1[3];
  vector[2] sigma_narrow2[3];

  for (tt in 1:3) {

    matrix[4,4] Ub = pow(Ubroad[tt],r);
    matrix[2,2] U1n = pow(Unarrow1[tt],r);
    matrix[2,2] U2n = pow(Unarrow1[tt],r);

    if (contextual==1) {
      // contextual utility normalization
      Ub = Ub/(Ub[4,1]-Ub[4,4]);
      U1n = U1n/(U1n[2,1]-U1n[2,2]);
      U2n = U2n/(U2n[2,1]-U2n[2,2]);
    }

    rho[tt] = QREllp(
      Ub,U1n, U2n,
      lambda,mix_narrow,
      iter,tol
  );



    sigma_broad[tt] = exp(rho[tt][1:4]);
```

```
    sigma_narrow1[tt] = exp(rho[tt][5:6]);
    sigma_narrow2[tt] = exp(rho[tt][7:8]);

  }


}

model {




  // likelihood contribution
  if (UseData==1) {


    for (ii in 1:N) {

      target += log(
        mix_narrow*sigma_narrow1[treatment[ii]][action_narrow1[ii]]*sigma_narrow2[treatment[ii]][action
        +(1.0-mix_narrow)*sigma_broad[treatment[ii]][action_broad[ii]]
      );


    }

  }


  // priors

  target += lognormal_lpdf(r | prior_r[1],prior_r[2]);
  target += lognormal_lpdf(lambda | prior_lambda[1],prior_lambda[2]);
  target += beta_lpdf(mix_narrow | prior_mix[1],prior_mix[2]);

}

generated quantities {

  vector[4] sigma[3];

  for (tt in 1:3) {

    sigma[tt] = exp(rho[tt]);

  }


}
```

Note that they are exactly the same except for how the likelihood is clacluated.

15.4 Results

I estimate these three models with the following priors:

$$\log r \sim N(\log(0.5), 1)$$
$$\log \lambda \sim N(\log(5), 1)$$
$$\phi \sim \text{Beta}(1, 1)$$

I used data from the final 10 rounds of play in order to hopefully mitigate any learning effects.

Table 35 shows the parameter estimates (first three rows) and implied action frequencies (remaining rows) for the four models estimated. The rightmost column of this Table also shows the empirical choice frequencies from the data used to estimate these models. Comparing the "Data" column to the estimated choice frequencies reveals that none the models perform particularly well at organizing these choice frequencies. However model posterior probabilities (assigning equal prior probabilities to all four models) overwhelmingly (one to at least five decimal places) select the dichotomous mixture model. Here we estimate that approximately 46% of participants narrowly bracket the games. If I were forced to select one of the one-type models, the narrow assumption does overwhelmingly better than the broad assumption based on model posterior probability.

```
fmt<-"%.3f"

ESTIMATES <-rbind(
  summary("Code/HowManyGames/Fit_QRE_broad.rds" |> readRDS())$summary |>
    data.frame() |>
    rownames_to_column(var = "parameter") |>
    mutate(
      model = "Broad"
    ),
   summary("Code/HowManyGames/Fit_QRE_narrow.rds" |> readRDS())$summary |>
    data.frame() |>
    rownames_to_column(var = "parameter") |>
    mutate(
      model = "Narrow"
    ),
   summary("Code/HowManyGames/Fit_QRE_mix.rds" |> readRDS())$summary |>
    data.frame() |>
    rownames_to_column(var = "parameter") |>
    mutate(
      model = "Dichotomous"
    ),
   summary("Code/HowManyGames/Fit_QRE_toolbox.rds" |> readRDS())$summary |>
    data.frame() |>
    rownames_to_column(var = "parameter") |>
    mutate(
      model = "Toolbox"
    )
) |>
  mutate(
    treatment = parameter |> str_split_i(",",1) |> parse_number(),
    action =  c("{AC}","{AD}","{BC}","{BD}")[(parameter |> str_split_i(",",2) |> parse_number())]
  ) |>
  filter(parameter!="lp__" & !grepl("rho",parameter) & !grepl("sigma_broad",parameter) & !grepl("sigma_
  mutate(
    parameter = ifelse(is.na(action),parameter,
                       paste0("$\\sigma^",treatment,"_",action,"$")
```

```
                        )
  ) |>
  mutate(
    msd = paste0(sprintf(fmt,mean)," (",sprintf(fmt,sd),")")
  )

TAB<-ESTIMATES |>
  pivot_wider(
    id_cols = parameter,
    names_from = model,
    values_from = msd
  )


D<-"Data/Bland2019HowManyGames.csv" |>
  read.csv() |>
  arrange(uid,Period) |>
  filter(Period>10) |>
  mutate(
    uid = paste("uid =",uid) |> as.factor() |> as.integer()
  ) |>
  mutate(
    BroadActComb = 2*(1-ActionAB)+(1-ActionCD)+1,
    treatment = ifelse(
      d1==0 & d2==0,1,ifelse(
        d1!=0, 2, 3
      )
    )
  ) |>
  group_by(treatment) |>
  mutate(
    count = n()
  ) |>
  group_by(treatment,BroadActComb) |>
  summarize(
    Data = sprintf(fmt,n()/mean(count))
  ) |>
  mutate(
    parameter = paste0("$\\sigma^",treatment,"_",c("{AC}","{AD}","{BC}","{BD}")[BroadActComb],"$")
  ) |>
  ungroup() |>
  dplyr::select(Data,parameter,-treatment)


TAB<-TAB |>
  full_join(D,by="parameter")

TAB[is.na(TAB)]<-""

TAB<-TAB |>
  mutate(
    order = ifelse(
      parameter == "r",1,
```

Table 35: Estimates from the three models. Posterior means with standard deviations in parentheses. The 'Data' column shows the empirical choice frequencies from the data used to estimate the models.

	One type		Two types		
parameter	Broad	Narrow	Dichotomous	Toolbox	Data
r	0.474 (1.476)	0.085 (0.026)	0.022 (0.010)	0.021 (0.010)	
lambda	0.149 (0.075)	2.281 (0.410)	3.562 (0.330)	0.935 (0.151)	
mix_narrow			0.457 (0.059)	0.927 (0.063)	
σ^1_{AC}	0.250 (0.002)	0.397 (0.014)	0.268 (0.003)	0.258 (0.001)	0.462
σ^1_{AD}	0.250 (0.001)	0.249 (0.007)	0.267 (0.003)	0.257 (0.001)	0.208
σ^1_{BC}	0.252 (0.002)	0.217 (0.006)	0.279 (0.004)	0.260 (0.001)	0.142
σ^1_{BD}	0.248 (0.004)	0.137 (0.008)	0.187 (0.004)	0.225 (0.003)	0.188
σ^2_{AC}	0.247 (0.001)	0.245 (0.006)	0.232 (0.003)	0.242 (0.001)	0.422
σ^2_{AD}	0.248 (0.001)	0.402 (0.016)	0.190 (0.005)	0.232 (0.003)	0.320
σ^2_{BC}	0.252 (0.001)	0.134 (0.009)	0.315 (0.007)	0.268 (0.003)	0.050
σ^2_{BD}	0.253 (0.002)	0.220 (0.004)	0.263 (0.005)	0.259 (0.001)	0.207
σ^3_{AC}	0.247 (0.001)	0.309 (0.007)	0.163 (0.009)	0.225 (0.005)	0.406
σ^3_{AD}	0.248 (0.001)	0.194 (0.006)	0.207 (0.003)	0.236 (0.002)	0.294
σ^3_{BC}	0.252 (0.001)	0.305 (0.006)	0.286 (0.002)	0.263 (0.002)	0.147
σ^3_{BD}	0.253 (0.002)	0.192 (0.007)	0.344 (0.012)	0.276 (0.005)	0.153

```
      ifelse(parameter=="lambda",2,
             ifelse(parameter=="mix_narrow",3,4))
    )
  ) |>
  arrange(order,parameter) |>
  dplyr::select(-order)

TAB |>
  kbl(digits=3,caption = "Estimates from the three models. Posterior means with standard deviations in p
  kable_classic(full_width=FALSE) |>
  add_header_above(c("","One type"=2,"Two types"=2,""))
```

Of some concern here is that the estimated posterior means for r for all but the broad-only model are not particularly plausible: they are very close to zerom, which implies extreme risk-aversion. One explanation for this could be that risk-aversion isn't really what is driving behavior in this game, and we might need another model of behavior to better organize the data. Alternatively, it could be that quantal response equilibrium is not a good model for play. In another chapter I will further explore this by taking a *learning* model to the data (instead of an *equilibrium* model). This may better respect the heterogeneity in participants' decision-making processes.

One final thing to note here is that we are evaluating some non-nested models. In particular, broad bracketing cannot generate the same predictions as narrow bracketing and vice versa. On the other hand, both mixture models can generate the same predictions as the broad bracketing model by setting $\phi = 0$, or the narrow bracketing model by setting $\phi = 1$. On a a third hand, if ϕ is strictly in the interior of the unit interval, then the two mixture models will have different implications about the data. The process of evaluating model posterior probabilities, however, is exactly the same whether they are nested or not. This is not the case if we were estimating these models using maximum likelihood, where we might have ended up having to use either AIC or BIC to select a model.

15.5 *R* code used to estimate these models

```
library(tidyverse)
library(rstan)
rstan_options(auto_write = TRUE)
options(mc.cores = parallel::detectCores())
library(bridgesampling)

D<-"Data/Bland2019HowManyGames.csv" |>
  read.csv() |>
  arrange(uid,Period) |>
  filter(Period>10) |>
  mutate(
    uid = paste("uid =",uid) |> as.factor() |> as.integer()
  ) |>
  mutate(
    BroadActComb = 2*(1-ActionAB)+(1-ActionCD)+1,
    treatment = ifelse(
      d1==0 & d2==0,1,ifelse(
        d1!=0, 2, 3
      )
    )
  )

U0broad<-rbind(
  c(66, 66, 66, 66),
  c(170,10,170, 10),
  c(156,156,56,56),
  c(260,100,160,0)
)

U0ABnarrow<-rbind(
  c(10,10),
  c(100,0)
)

U0CDnarrow<-rbind(
  c(56,56),
  c(160,0)
)


dStan<-list(

  Ubroad = list(
    U0broad,
    U0broad + 50,
    U0broad + 50
  ),


  Unarrow1 = list(
    U0ABnarrow,
    U0ABnarrow+50,
```

```
    U0ABnarrow
  ),

  Unarrow2 = list(
    U0CDnarrow,
    U0CDnarrow,
    U0CDnarrow+50
  ),

  N = dim(D)[1],

  id = D$uid,

  nparticipants = D$uid |> unique() |> length(),

  treatment = D$treatment,
  action = D$BroadActComb,
  action_broad = D$BroadActComb,

  prior_r = c(log(0.5),1),
  prior_lambda = c(log(5),1),
  prior_mix = c(1,1),

  tol = 10^-5,
  iter = 1000,

  UseData = 1,

  contextual = 1


)

model_toolbox<-"Code/HowManyGames/QRE_toolbox.stan" |>
  stan_model()



Fit_toolbox <- model_toolbox |>
  sampling(
    #chains=1,iter=100, # debugging settings
    data=dStan,seed=42
  )

bs_toolbox<-Fit_toolbox |> bridge_sampler()

Fit_toolbox |>
  saveRDS("Code/HowManyGames/Fit_QRE_toolbox.rds")



model_mix<-"Code/HowManyGames/QRE_mix.stan" |>
```

```
  stan_model()



Fit_mix <- model_mix |>
  sampling(
    #chains=1,iter=100, # debugging settings
    data=dStan,seed=42
     )

bs_mix<-Fit_mix |> bridge_sampler()

Fit_mix |>
  saveRDS("Code/HowManyGames/Fit_QRE_mix.rds")

model_narrow<-"Code/HowManyGames/QRE_narrow.stan" |>
  stan_model()

Fit_narrow<-model_narrow |>
  sampling(data=dStan,seed=42)

bs_narrow<-Fit_narrow |> bridge_sampler()

Fit_narrow |>
  saveRDS("Code/HowManyGames/Fit_QRE_narrow.rds")

model_broad<-"Code/HowManyGames/QRE_broad.stan" |>
  stan_model()

Fit_broad <- model_broad |>
  sampling(
    #iter = 100,chains=1, # debugging settings
    data=dStan,seed=42,
    control = list(adapt_delta = 0.99)
  )

bs_broad<-Fit_broad |> bridge_sampler()

Fit_broad |>
  saveRDS("Code/HowManyGames/Fit_QRE_broad.rds")

postprob<-post_prob(bs_mix,bs_toolbox,bs_narrow,bs_broad)

postprob |>
  saveRDS("Code/HowManyGames/postprob_QRE.rds")
```

16 Application: QRE in a Bayesian game and cursed equilibrium

Many games we study in the lab are *Bayesian* games. These could include, but are not limited to, auctions, signaling games, and poker. Bayesian games involve heterogeneous players. This adds a lot of realism to the model, but also makes things more complicated, because we need to keep track of player *types*. For quantal

Table 36: Proposal game studied in Bochet and Magnani (2024)

	P	N
P	$\mu(q_2)$, $\mu(q_1)$	$\rho(q_1)$, $\rho(q_2)$
N	$\rho(q_1)$, $\rho(q_2)$	$\rho(q_1)$, $\rho(q_2)$

response equilibrium, this usually means solving for the equilibrium mixed strategy for *every* type.[68] Once the sytem of equations describing the QRE is set up, solving for QRE is done in exactly the same way as a non-Bayesian game.

16.1 Example game and dataset

In this chapter, we will use a game studied experimentally in Bochet and Magnani (2024). In this game, two players are randomly endowed with a quality $q_i \in \{H, M, L\}$, and a signal $s_i \in \{h, m, l\}$ that is informative of their opponent's quality. Each of the three qualities are equally likely to be drawn, and players' qualities are independent draws. After observing their qualities and signals, players play a one-shot game, whose payoffs depend on the players' qualities. This game is shown in Table 36. Here $\mu(\cdot)$ and $\rho(\cdot)$ are funtions of qualities. In particular, note that if player i chooses to not propose (N), then they guarantee a known payoff of $\rho(q_i)$ (since they know their own quality). They also earn this payoff if their opponent chooses N, irrespective of their own action. On the other hand, if both players choose to propose (P), then each player earns a payoff that is a function of their *opponent's* quality. The payoff values for μ and ρ are shown in Table 37. Here there are two treatments, with Game A having a larger reservation payoff for the high type compared to Game B.

```
payoffs<-rbind(
  c("$\\mu(q_2)$, $\\mu(q_1)$","$\\rho(q_1)$, $\\rho(q_2)$"),
  c("$\\rho(q_1)$, $\\rho(q_2)$","$\\rho(q_1)$, $\\rho(q_2)$")
)
colnames(payoffs)<-c("P","N")->rownames(payoffs)
payoffs |>
  kbl(caption = "Proposal game studied in Bochet and Magnani (2024)") |>
  kable_classic(full_width=FALSE)
```

```
TAB<-rbind(c(160,80,40),
           c(100,75,25),
           c(80,75,25)
           )

rownames(TAB)<-c("$\\mu(q)$","$\\rho(q)$ (Game A)","$\\rho(q)$ (Game B)")
colnames(TAB)<-c("$H$","$M$","$L$")

TAB |>
  kbl(caption = "Payoff values in game studied in Bochet and Magnani (2024)
") |>
  kable_classic(full_width=FALSE) |>
  add_header_above(c(" "=1,"$q$"=3))
```

16.2 Solving for QRE

As in Bochet and Magnani (2024), we will here focus on the *symmetric* equilibrium of the game, where all players with the same type choose P with the same probability. Since there are three different qualities that a player could have, and three different signals a player could receive, this means that there are $3 \times 3 = 9$

[68] There are some Bayesian games where we need not do this. These are games where an opponent's type does not affect the player's utility function. See Bland (2023c) for more details on this.

Table 37: Payoff values in game studied in Bochet and Magnani (2024)

	q		
	H	M	L
$\mu(q)$	160	80	40
$\rho(q)$ (Game A)	100	75	25
$\rho(q)$ (Game B)	80	75	25

types in this game. That is, the types indexed by the quality-signal combination are:

$$(q, s) = \tau \in \{Hh, Mh, Lh, Hm, Mm, Lm, Hl, Ml, Ll\}$$

In the code below you will see that there are a lot of 9-dimensional vectors and 9×9 matrices for this reason. I will use this ordering of types as a convention for indexing these vectors and matrices. For example, the fourth element of the vector of mixed strategies will correspond to the proposal probability for players who are quality $q = H$ and receive signal $s = m$.

We will start by deriving equations that describe the quantal response equilibrium of the game assuming that players properly condition their beliefs on whether or not their opponent proposes. This will be our "baseline" model. We will then implement a 1-parameter extension that permits players to not fully take the informational content of the opponent's strategy into account. This is called "cursed equilibrium" (Eyster and Rabin 2005).

16.2.1 Baseline model

Note here that I am not following the notation of the original paper. I found it very helpful to work my way through this with using the joint distribution of qualities and signals, which I denote $\pi(q, s, q', s')$. From here, all expected utilities can be expressed as a conditional distribution $\pi(q', s' \mid q, s)$ which is a player with quality q and signal s's belief about the quality and signal of their opponent.

A player with quality q and signal s proposes with probability $\sigma(q, s)$. Let $\pi(q, s)$ be the joint distribution of quality and signals.

Given signal s, a player's posterior beliefs about their opponent's quality is:

$$\pi(q' \mid s) = \frac{\pi(q', s)}{\pi(s)} = \frac{\pi(q', s)}{\sum_{q''} \pi(q'', s)}$$

Note that the player's quality q is informative of the signal the opponent received. That is:

$$\pi(s' \mid q) = \frac{\pi(q, s')}{\pi(q)} = \frac{\pi(q, s')}{\sum_{s''} \pi(q, s'')}$$

Hence, the player has informed beliefs about their opponent's quality, $\pi(q' \mid s)$, and informed beliefs about their opponent's signal $\pi(s' \mid q)$. We can summarize this as joint beliefs over the opponent's quality and signal as follows:

$$\pi(q', s' \mid q, s) = \frac{\pi(q, s, q', s')}{\sum_{q', s'} \pi(q, s, q', s')}$$

$$\begin{aligned} \pi(q', s', q, s) &= \pi(s, s' \mid q, q')\pi(q, q') \\ &= \pi(s' \mid q)\pi(s \mid q')\pi(q)\pi(q') \end{aligned}$$

where the second line follows because qualities are unconditionally independent, and signals are independent conditional on opponent's quality.[69] Since $\pi(q', s' \mid q, s)$ is only a function of the type-generating process, which is exogenous in the game, we can pre-compute these quantities in *Stan*'s `transformed data` block. You will see these computed there as `PI`(for the joint distribution), and `PI_cond` (for the conditional distribution).

It follows that the expected utility of proposing with quality q and signal s is:

$$\begin{aligned} U^P(q,s) &= E_{q',s'}\left[\mu(q')\sigma(q',s') + (1-\sigma(q',s'))\rho(q) \mid q, s\right] \\ &= \sum_{q',s'}\left[\mu(q')\sigma(q',s') + (1-\sigma(q',s'))\rho(q)\right]\pi(q',s' \mid q,s) \end{aligned}$$

and the utility of not proposing is $U^N(q,s) = \rho(q)$.

We can therefore write the QRE conditions in logit form, with $\ell(q,s) = \text{logit}(\sigma(q,s))$, as:

$$H_{q,s}(\lambda, \sigma) = \ell(q,s) - \lambda\left[\sum_{q',s'}\left[\mu(q')\sigma(q',s') + (1-\sigma(q',s'))\rho(q)\right]\pi(q',s' \mid q,s) - \rho(q)\right]$$

with derivatives:

$$\begin{aligned} \frac{\partial H_{q,s}(\lambda,\sigma)}{\partial \ell(t,v)} &= I((q,s)=(t,v)) - \lambda\left[\mu(t) - \rho(q)\right]\pi(t,v \mid q,s)\frac{\partial \sigma(t,v)}{\partial \ell(t,v)} \\ &= I((q,s)=(t,v)) - \lambda\left[\mu(t) - \rho(q)\right]\pi(t,v \mid q,s)\sigma(t,v)(1-\sigma(t,v)) \end{aligned}$$

and:

$$\frac{\partial H_{q,s}}{\partial \lambda} = -\sum_{q',s'}\left[\mu(q')\sigma(q',s') + (1-\sigma(q',s'))\rho(q)\right]\pi(q',s' \mid q,s) - \rho(q)$$

I then implement a predictor-corrector algorithm as follows:[70]

0. **Initialization:** Start at a known QRE. This is $\lambda = 0$, $\sigma(q,s) = 0.5\ \forall q, s$; or $\lambda = 0$, $\ell(q,s) = 0\ \forall q, s$.
1. **Predictor step:** Use the derivative $\frac{\partial H}{\partial \lambda}$ to predict the direction of ℓ for a small increase in λ. That is: $\ell^{t+1} = \ell^t + \frac{\partial H}{\partial \lambda}(\lambda^{t+1} - \lambda^t)$
2. **Corrector steps:** Correct for any errors made with the linear approximation in (1) using Newton's method. That is, iterate on $\ell^{t+1} = \ell^t - \left(\frac{\partial H(\lambda,\ell)}{\partial \ell^\top}\right)^{-1} H(\lambda, \ell)$ until a desired level of accuracy is achieved.
3. **Repeat:** If λ^t is less than the required λ, go back to step (1).

Here is the *Stan* program I wrote to estimate this baseline QRE model. Note that I am passing an array of integers called `UseData` to *Stan*. This is a list of rows of the data that will contribute to the likelihood. This will be useful for cross-validation, as we will need to estimate the model several times leaving out different groups of observations.

```
functions {

  vector H(real lambda, vector ell,
    data matrix PI_cond,
    data vector mu, data vector rho,
```

[69] Think about how you would simulate qualities and signals here. You would (1) draw each player's quality, then (2) use this to draw their opponent's signal.

[70] This is a variant on the predictor-corrector algorithm discussed earlier in this book. Here, I fix a grid of λs, with the last λ being the value we need to compute QRE, and trace out the QRE over this grid.

```
  data int[] QQ, data int[] SS
) {

  vector[9] sigma = inv_logit(ell);

  vector[9] result = ell;

  for (rr in 1:9) {

    result[rr] -= lambda*(sum(
      (mu[QQ].*sigma+(1.0-sigma)*rho[QQ[rr]]).*PI_cond[,rr]
      )
      -rho[QQ[rr]]
    );

  }


  return result;

}


matrix DH(real lambda, vector ell,
  data matrix PI_cond,
  data vector mu, data vector rho,
  data int[] QQ, data int[] SS
) {
 vector[9] sigma = inv_logit(ell);

 matrix[9,9] result = diag_matrix(rep_vector(1.0,9));

 for (rr in 1:9) {
   for (cc in 1:9) {


     result[rr,cc]+= -lambda*(mu[QQ[cc]]-rho[QQ[rr]])*PI_cond[cc,rr]*sigma[cc]*(1.0-sigma[cc]);

   }
 }
 return result;
}

vector dlH(real lambda, vector ell,
  data matrix PI_cond,
  data vector mu, data vector rho,
  data int[] QQ, data int[] SS
) {

  vector[9] sigma = inv_logit(ell);

  vector[9] result = ell;
```

```
  for (rr in 1:9) {

    result[rr] -= (sum(
      (mu[QQ].*sigma+(1.0-sigma)*rho[QQ[rr]]).*PI_cond[,rr]
      )
      -rho[QQ[rr]]
    );

  }


  return result;

}



vector solve_QRE(real lambda,
  data matrix PI_cond,
  data vector mu, data vector rho,
  data int[] QQ, data int[] SS,
  data int maxiter, data real tol, data vector lxgrid
) {
     vector[9] result = rep_vector(0.0,9);

     int nx = dims(lxgrid)[1];

     for (ll in 2:nx) {

       // predictor step

       real l = lambda*0.5*(lxgrid[ll-1]+lxgrid[ll]);
       vector[9] Hlambda = dlH(l,result,PI_cond,mu,rho,QQ,SS);
       result += Hlambda*(lxgrid[ll]-lxgrid[ll-1]);

       l = lambda*lxgrid[ll];

       // corrector step
       for (ii in 1:maxiter) {

         matrix[9,9] J = DH(l,result,PI_cond,mu,rho,QQ,SS);
         vector[9]   F = H(l,result,PI_cond,mu,rho,QQ,SS);

         vector[9] dx = -J\F;

         result += dx;

         if (max(abs(dx))<tol) {

           break;
         }

         if (ii == maxiter && ll==nx) {
```

```
            print("Corrector step did not converge for lambda =",l);

          }

        }

        //
      }

      return result;
  }

}


data {

  int N;

  int<lower=1,upper=9> type[N];
  int<lower=1,upper=2> game[N];
  int<lower=0,upper=1> action[N];

  int<lower=0,upper=N> nUseData;
  int UseData[nUseData];


  vector[2] prior_lambda;

  int maxiter;
  real tol;
  int nlxgrid;

}

transformed data {

  // grid to trace out QRE

  vector[nlxgrid] lxgrid;
  for (ii in 1:nlxgrid) {
    lxgrid[ii] = (ii-1.0)/(nlxgrid-1.0);
  }



  // game parameters
  vector[3] mu = [160,80,40]';
  vector[3] rho[2];
    rho[1] = [100,75,25]';
    rho[2] = [80,75,25]';
```

```
// data that actually contributes to the likelihood
int type_training[nUseData] = type[UseData];
int game_training[nUseData] = game[UseData];
int action_training[nUseData] = action[UseData];

// data used for testing
int type_test[N-nUseData];
int game_test[N-nUseData];
int action_test[N-nUseData];

int count_test = 1;
int count_train = 1;
for (ii in 1:N) {

  if (nUseData>0) {

    if (UseData[count_train]==ii) {

      if (count_train < nUseData) {
        count_train +=1;
      }

    } else {
      type_test[count_test] = type[ii];
      game_test[count_test] = game[ii];
      action_test[count_test] = action[ii];
      count_test+=1;
    }
  } else {
      type_test[count_test] = type[ii];
      game_test[count_test] = game[ii];
      action_test[count_test] = action[ii];
      count_test+=1;

  }



}


// some indices for identifying types


int QQ[9];
int SS[9];

for (qq in 1:3) {
  for (ss in 1:3) {

    int tt = qq+3*(ss-1);

    QQ[tt] = qq;
```

```
      SS[tt] = ss;

    }
  }

  matrix[3,3] signal_likelihoods = [
    [0.5,0.5,0.0],
    [0.0,1.0,0.0],
    [0.0,0.5,0.5]
    ];

  matrix[9,9] PI;

  for (rr in 1:9) {
    for (cc in 1:9) {

      PI[rr,cc] = (1.0/3.0)*(1.0/3.0)*signal_likelihoods[QQ[rr],SS[cc]]*signal_likelihoods[QQ[cc],SS[rr]


    }
  }

  matrix[9,9] PI_cond = rep_matrix(0.0,9,9);

  for (rr in 1:9) {
    for (cc in 1:9) {

      //if (sum(PI[,cc])>0) {
        PI_cond[rr,cc] += PI[rr,cc]/sum(PI[,cc]);
      //}

    }
  }



}


parameters {

 real<lower=0> lambda;

}

transformed parameters {

  vector[9] ELL[2];

  ELL[1] = solve_QRE(lambda,
    PI_cond,
    mu, rho[1],
    QQ,SS,
```

```
    maxiter,tol,lxgrid
  ) ;
  ELL[2] = solve_QRE(lambda,
    PI_cond,
    mu, rho[2],
    QQ,SS,
    maxiter,tol,lxgrid
  ) ;


}


model {

  // prior
  target += lognormal_lpdf(lambda | prior_lambda[1],prior_lambda[2]);

  // likelihood
  for (ii in 1:nUseData) {
    target += bernoulli_logit_lpmf(action_training[ii] |  ELL[game_training[ii]][type_training[ii]]);
  }

}

generated quantities {

  vector[9] sigma[2];
    sigma[1] = inv_logit(ELL[1]);
    sigma[2] = inv_logit(ELL[2]);

  real logL_test = 0;

  for (ii in 1:(N-nUseData)) {
    logL_test += bernoulli_logit_lpmf(action_test[ii] |  ELL[game_test[ii]][type_test[ii]]);
  }


  //matrix[9,9] PI_cond_test = PI_cond;

  //matrix[9,9] PI_test = PI;
}
```

16.2.2 Cursed equilibrium

In Bayesian games, a popular one-parameter extension is the "Cursed Equilibrium" (Eyster and Rabin 2005). In this model, players maintain correct beliefs about opponents' strategies, but underestimate the information that is contained in these strategies. In this game, this means that a participant might not fully factor in that if an opponent proposes, that tells us something about their quality. In the most extreme "fully cursed" case, the player believes that their payoff if both propose is the *average* $\mu(q)$ (i.e. integrating out q), which I will denote $\bar{\mu}$. In the χ-cursed model, players *partially* incorporate this information, and so their expected utility from proposing is:

$$U_{\chi}^{P}(q,s) = \sum_{q',s'} \left[\underbrace{((1-\chi)\mu(q') + \chi\bar{\mu})}_{\text{the new part}} \sigma(q',s') - (1-\sigma(q',s'))\rho(q) \right] \pi(q',s' \mid q,s)$$

where $\chi \in [0,1]$ measures the degree of "cursedness". Modifying the QRE equations here is relatively simple: whenever you see a $\mu(\cdot)$, just replace it with $(1-\chi)\mu(\cdot) + \chi\bar{\mu}$.

Here is the *Stan* program I wrote to estimate the cursed model:

```
functions {

  vector H(real lambda, vector ell,
    real chi,
    data matrix PI_cond,
    data vector mu, data vector rho,
    data int[] QQ, data int[] SS
  ) {

    vector[9] sigma = inv_logit(ell);

    vector[9] result = ell;

    for (rr in 1:9) {

      result[rr] -= lambda*(sum(
        (((1.0-chi)*mu[QQ]+chi*mean(mu)).*sigma+(1.0-sigma)*rho[QQ[rr]]).*PI_cond[,rr]
        )
        -rho[QQ[rr]]
      );

    }


    return result;

  }


  matrix DH(real lambda, vector ell,
    real chi,
    data matrix PI_cond,
    data vector mu, data vector rho,
    data int[] QQ, data int[] SS
  ) {
   vector[9] sigma = inv_logit(ell);

   matrix[9,9] result = diag_matrix(rep_vector(1.0,9));

   for (rr in 1:9) {
     for (cc in 1:9) {


       result[rr,cc]+= -lambda*((1-chi)*mu[QQ[cc]]+chi*mean(mu)-rho[QQ[rr]])*PI_cond[cc,rr]*sigma[cc]*(
```

```
    }
  }
  return result;
}

vector dlH(real lambda, vector ell,
  real chi,
  data matrix PI_cond,
  data vector mu, data vector rho,
  data int[] QQ, data int[] SS
) {

  vector[9] sigma = inv_logit(ell);

  vector[9] result = ell;

  for (rr in 1:9) {

    result[rr] -= (sum(
      (((1.0-chi)*mu[QQ]+chi*mean(mu)).*sigma+(1.0-sigma)*rho[QQ[rr]]).*PI_cond[,rr]
      )
      -rho[QQ[rr]]
    );

  }


  return result;

}



vector solve_QRE(real lambda, real chi,
  data matrix PI_cond,
  data vector mu, data vector rho,
  data int[] QQ, data int[] SS,
  data int maxiter, data real tol, data vector lxgrid
) {
     vector[9] result = rep_vector(0.0,9);

     int nx = dims(lxgrid)[1];

     for (ll in 2:nx) {

       // predictor step

       real l = lambda*0.5*(lxgrid[ll-1]+lxgrid[ll]);
       vector[9] Hlambda = dlH(l,result,chi,PI_cond,mu,rho,QQ,SS);
       result += Hlambda*(lxgrid[ll]-lxgrid[ll-1]);

       l = lambda*lxgrid[ll];
```

```
          // corrector step
          for (ii in 1:maxiter) {

            matrix[9,9] J = DH(l,result,chi,PI_cond,mu,rho,QQ,SS);
            vector[9]   F = H(l,result,chi,PI_cond,mu,rho,QQ,SS);

            vector[9] dx = -J\F;

            result += dx;

            if (max(abs(dx))<tol) {

              break;
            }

            if (ii == maxiter && ll==nx) {

              print("Corrector step did not converge for lambda =",l);

            }

          }

          //
        }

        return result;
  }

}


data {

  int N;

  int<lower=1,upper=9> type[N];
  int<lower=1,upper=2> game[N];
  int<lower=0,upper=1> action[N];

  int<lower=0,upper=N> nUseData;
  int UseData[nUseData];


  vector[2] prior_lambda;
  vector[2] prior_chi;

  int maxiter;
  real tol;
  int nlxgrid;

}
```

```
transformed data {

  // grid to trace out QRE

  vector[nlxgrid] lxgrid;
  for (ii in 1:nlxgrid) {
    lxgrid[ii] = (ii-1.0)/(nlxgrid-1.0);
  }



  // game parameters
  vector[3] mu = [160,80,40]';
  vector[3] rho[2];
    rho[1] = [100,75,25]';
    rho[2] = [80,75,25]';


  // data that actually contributes to the likelihood
  int type_training[nUseData] = type[UseData];
  int game_training[nUseData] = game[UseData];
  int action_training[nUseData] = action[UseData];

  // data used for testing
  int type_test[N-nUseData];
  int game_test[N-nUseData];
  int action_test[N-nUseData];

  int count_test = 1;
  int count_train = 1;
  for (ii in 1:N) {

    if (nUseData>0) {

      if (UseData[count_train]==ii) {

        if (count_train < nUseData) {
          count_train +=1;
        }
      } else {
        type_test[count_test] = type[ii];
        game_test[count_test] = game[ii];
        action_test[count_test] = action[ii];
        count_test+=1;
      }
    } else {
        type_test[count_test] = type[ii];
        game_test[count_test] = game[ii];
        action_test[count_test] = action[ii];
        count_test+=1;

    }
```

```
  }


  // some indices for identifying types


  int QQ[9];
  int SS[9];

  for (qq in 1:3) {
    for (ss in 1:3) {

      int tt = qq+3*(ss-1);

      QQ[tt] = qq;
      SS[tt] = ss;

    }
  }

  matrix[3,3] signal_likelihoods = [
    [0.5,0.5,0.0],
    [0.0,1.0,0.0],
    [0.0,0.5,0.5]
    ];

  matrix[9,9] PI;

  for (rr in 1:9) {
    for (cc in 1:9) {

      PI[rr,cc] = (1.0/3.0)*(1.0/3.0)*signal_likelihoods[QQ[rr],SS[cc]]*signal_likelihoods[QQ[cc],SS[rr]


    }
  }

  matrix[9,9] PI_cond = rep_matrix(0.0,9,9);

  for (rr in 1:9) {
    for (cc in 1:9) {

      //if (sum(PI[,cc])>0) {
        PI_cond[rr,cc] += PI[rr,cc]/sum(PI[,cc]);
      //}

    }
  }



}
```

```
parameters {

 real<lower=0> lambda;
 real<lower=0,upper=1> chi;

}

transformed parameters {

  vector[9] ELL[2];

  ELL[1] = solve_QRE(lambda,chi,
    PI_cond,
    mu, rho[1],
    QQ,SS,
    maxiter,tol,lxgrid
  ) ;
  ELL[2] = solve_QRE(lambda,chi,
    PI_cond,
    mu, rho[2],
    QQ,SS,
    maxiter,tol,lxgrid
  ) ;


}


model {

  // prior
  target += lognormal_lpdf(lambda | prior_lambda[1],prior_lambda[2]);
  target += beta_lpdf(chi | prior_chi[1],prior_chi[2]);

  // likelihood
  for (ii in 1:nUseData) {
    target += bernoulli_logit_lpmf(action_training[ii] |  ELL[game_training[ii]][type_training[ii]]);
  }

}

generated quantities {

  vector[9] sigma[2];
    sigma[1] = inv_logit(ELL[1]);
    sigma[2] = inv_logit(ELL[2]);

  real logL_test = 0;

  for (ii in 1:(N-nUseData)) {
    logL_test += bernoulli_logit_lpmf(action_test[ii] |  ELL[game_test[ii]][type_test[ii]]);
  }
```

```
}
```

16.3 A quick prior calibration

The cursed model has two parameters: $\lambda > 0$ and $\chi \in (0, 1)$. For χ I am going to assume a uniform prior, meaning that in the prior each level of cursedness is equally likely. I do this because the parameter has natural bounds, and we don't want to rule out or favor any particular values of χ. On the other hand, I found it much harder to think about a prior for λ. Without looking at the model's predictions, it is hard to have any idea what values of λ seem reasonable. So we will do just that: look at the predictions. To do this, I traced out the QRE mixed strategies for a wide range of λ using *Stan*'s `Fixed_param` algorithm. Here are what the predictions look like for the baseline model ($\chi = 0$):

```
QRE<-"Code/BM2024/QRE.csv" |>
  read.csv() |>
  mutate(
    quality = paste("quality =",quality) |> factor(levels = paste("quality =",c("H","M","L")) ),
    signal = paste("signal =",signal) |> factor(levels = paste("signal =",c("h","m","l")) )
  )

 (
    ggplot(QRE,aes(x=lambda,y=sigma,linetype=game))
    +facet_grid(quality~signal)
    +geom_line()
    +scale_x_continuous(trans="log10")
    +theme_bw()
    +xlab(expression(lambda))
    +ylab(expression(sigma~(proposal~rate)))
  )
```

Looking at Figure 45, I noticed that (i) play is effectively coin-flipping for $\lambda < 10^{-3}$, and (ii) play appears to have converged for $\lambda > 1$. I will deem this the "interesting" part of the QRE correspondence, and place 95% prior probability in this region. That is, if we use a log-normal prior $\log \lambda \sim N(m, s^2)$, then:

$$\begin{aligned}
\log(10^{-3}) &= m - 1.96s \\
\log(1) &= m + 1.96s \\
\log(10^{-3}) + \log(1) &= 2m \\
m &= \frac{1}{2}(\log(10^{-3}) + \log(1)) \approx -3.45 \\
\log(1) - \log(10^{-3}) &= 2 \times 1.96s \\
s &= \frac{\log(1) - \log(10^{-3})}{2 \times 1.96} \approx 1.76
\end{aligned}$$

Which, for another sanity check, gets us a prior median for λ of $\exp(-3.45) \approx 0.032$ and prior mean of $\exp(-3.45 + 0.5 \times 1.76^2) \approx 0.149$.

16.4 Model results

Here are *Stan*'s summaries of the models' parameters:

```
Fit_baseline<- summary("Code/BM2024/Fit_baseline_qre.rds" |>
  readRDS())$summary |>
  data.frame() |>
  rownames_to_column(var = "parameter") |>
```

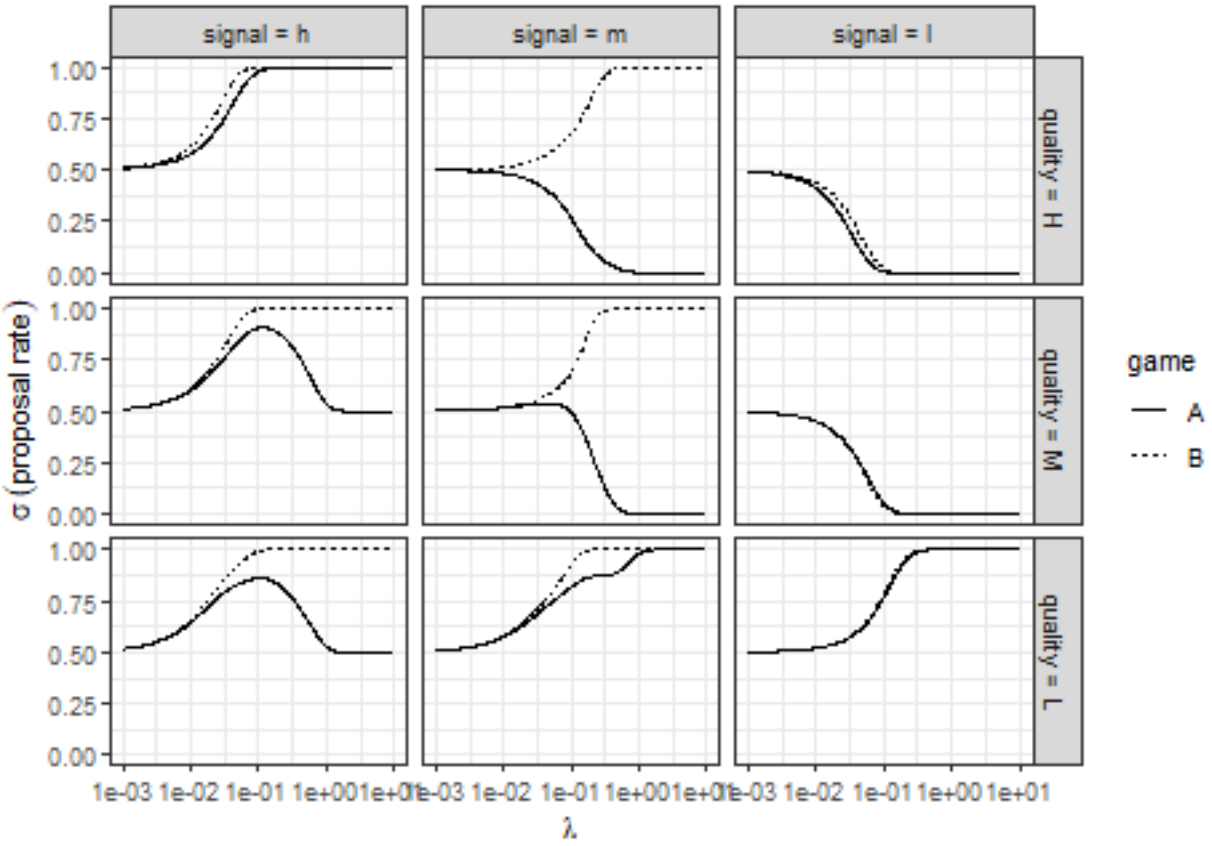


Figure 45: QRE predictions for the baseline ($\chi = 0$) model.

Table 38: Summary of the two models' posterior distributions

model	parameter	mean	se_mean	sd	X2.5.	X25.	X50.	X75.	X97.5.	n_eff	Rhat
baseline	λ	0.091	0.000	0.004	0.083	0.088	0.091	0.094	0.099	1124.232	1.001
cursed	λ	0.097	0.000	0.004	0.090	0.095	0.097	0.100	0.105	3600.401	1.000
cursed	χ	0.350	0.001	0.030	0.289	0.330	0.351	0.371	0.406	3454.873	1.000

```
  filter(parameter=="lambda" | parameter=="chi") |>
  mutate(model = "baseline") |>
  dplyr::select("model","parameter","mean","se_mean","sd","X2.5.","X25.","X50.","X75.","X97.5.","n_eff"

Fit_cursed<-summary("Code/BM2024/Fit_cursed_qre.rds" |>
  readRDS())$summary |>
  data.frame() |>
  rownames_to_column(var = "parameter") |>
  filter(parameter=="lambda" | parameter=="chi") |>
  mutate(model = "cursed") |>
  dplyr::select("model","parameter","mean","se_mean","sd","X2.5.","X25.","X50.","X75.","X97.5.","n_eff"

TAB<-rbind(Fit_baseline,Fit_cursed) |>
  mutate(
    parameter = paste0("$\\",parameter,"$")
  )

TAB |>
  kbl(digits=3,caption = "Summary of the two models' posterior distributions") |>
  kable_classic(full_width=FALSE)
```

So we can be fairly certain that $\chi \in (0.2, 0.4)$ for the cursed model.[71]

Interpreting χ is something we can do. $\chi \approx 0.35$ is closer to the baseline model than full cursedness, but it is still substantially far away from zero. So we should conclude that players are substantially cursed, but not fully cursed.

It is also useful with QRE to look at what they say about choices. To do this, I plot the posterior distributions of the mixed strategy alongside the empirical frequencies of actions in the experiment. These are shown in Figures 46 and 47. Firstly, it looks like the cursed model does slightly better at matching the data than does the baseline model. We will be a bit more rigorous in the next section with a model evaluation. But what I *really* like about these models is that they are organizing the data very well for just one or two parameters! The cursed model only has two parameters, but it is making predictions about eighteen empirical choice frequencies.

```
predictions_cursed<-extract("Code/BM2024/Fit_cursed_qre.rds" |>
  readRDS())$sigma

predictions_baseline<-extract("Code/BM2024/Fit_baseline_qre.rds" |>
  readRDS())$sigma

quality<-rep(c("H","M","L"),3)
signal<-c(rep("h",3),rep("m",3),rep("l",3))

PREDICTIONS<-tibble()
```

[71] I want to make it clear here that I am not trying to exactly replicate Table 9 of Bochet and Magnani (2024), who estimate $\chi \approx 0.9$ for the games. My estimation just uses data from the games, whereas they include data from two other tasks.

```
for (tt in 1:9) {
  for (gg in 1:2) {

    PREDICTIONS<-rbind(
      PREDICTIONS,
      tibble(
        game = c("A","B")[gg],type=tt,model="baseline",
        sigma = predictions_baseline[,gg,tt]
      ),
      tibble(
        game = c("A","B")[gg],type=tt,model="cursed",
        sigma = predictions_cursed[,gg,tt]
      )
      )

  }
}

PREDICTIONS<-PREDICTIONS |>
  mutate(
    quality = quality[type],
    signal = signal[type]
  ) |>
  mutate(
    quality = paste("quality =",quality) |> factor(levels = paste("quality =",c("H","M","L")) ),
    signal = paste("signal =",signal) |> factor(levels = paste("signal =",c("h","m","l")) )
  )

min_round<-21

empirical<-"Data/BochetMagnani2024/data_analysis/data/Main_data_base_game.csv" |>
  read.csv() %>%
  filter(subsession.round_number>=min_round) %>%
  mutate(player.type=factor(player.type)) %>%
  mutate(player.type=fct_recode(player.type,
                                "H"="1",
                                "M"="2",
                                "L"="3")) %>%
  mutate(player.signal=factor(player.signal)) %>%
  mutate(player.signal=fct_recode(player.signal,
                                  "h"="1",
                                  "m"="2",
                                  "l"="3")) %>%
  mutate(subsession.game_name=factor(subsession.game_name)) |>
  # now for JB's but coding up things to pass to Stan --------------------------
  mutate(
    type_code = paste0(player.type,player.signal) |>
      factor(levels = c("Hh","Mh","Lh","Hm","Mm","Lm","Hl","Ml","Ll")) |>
      as.numeric(),

    game_code = ifelse(subsession.game_name=="A",1,2),

    action = player.choice,
```

```
    uid = participant.code |> factor() |> as.numeric(),

    ugroupid = paste(session.code,group.id_in_subsession) |> factor() |> as.numeric()
  ) |>
  ungroup() |>
  mutate(
    rownum = 1:n()
  )|>
  group_by(player.type,player.signal,game_code) |>
  summarize(
    `Proposal rate` = mean(action)
  ) |>
  mutate(
    quality = paste("quality =",player.type) |> factor(levels = paste("quality =",c("H","M","L")) ),
    signal = paste("signal =",player.signal) |> factor(levels = paste("signal =",c("h","m","l")) )
  )


(
  ggplot(PREDICTIONS |> filter(game=="A"),aes(x=sigma,fill=model))
  +geom_histogram(aes(y=after_stat(density)),bins=100)
  +geom_vline(data=empirical |> filter(game_code==1),aes(xintercept=`Proposal rate`))
  +facet_grid(quality~signal)
  +theme_bw()
  +xlab(expression(sigma~(proposal~rate)))
  +ylab("posterior density")
  +scale_fill_manual(values = c("blue","red"))
  +labs(title = "Game A")
)
```

```
(
  ggplot(PREDICTIONS |> filter(game=="B"),aes(x=sigma,fill=model))
  +geom_histogram(aes(y=after_stat(density)),bins=100)
  +geom_vline(data=empirical |> filter(game_code==2),aes(xintercept=`Proposal rate`))
  +facet_grid(quality~signal)
  +theme_bw()
  +xlab(expression(sigma~(proposal~rate)))
  +ylab("posterior density")
  +scale_fill_manual(values = c("blue","red"))
  +labs(title = "Game B")
)
```

16.5 Model evaluation

Since we have taken two models to the data, we should do some kind of model evaluation to see which model we should work with. Here, I divided up the data into the six independent groups of participants (they were randomly re-matched within these groups) and performed 6-fold cross-validation based on these groups. Using ELPD as the measure of out-of-sample goodness-of-fit, it looks like the cursed model performs better here.

```
kfoldCV<-"Code/BM2024/kfoldCV.rds" |>
  readRDS()

kfoldCV$baseline%*%rep(1,6) |> mean()
```

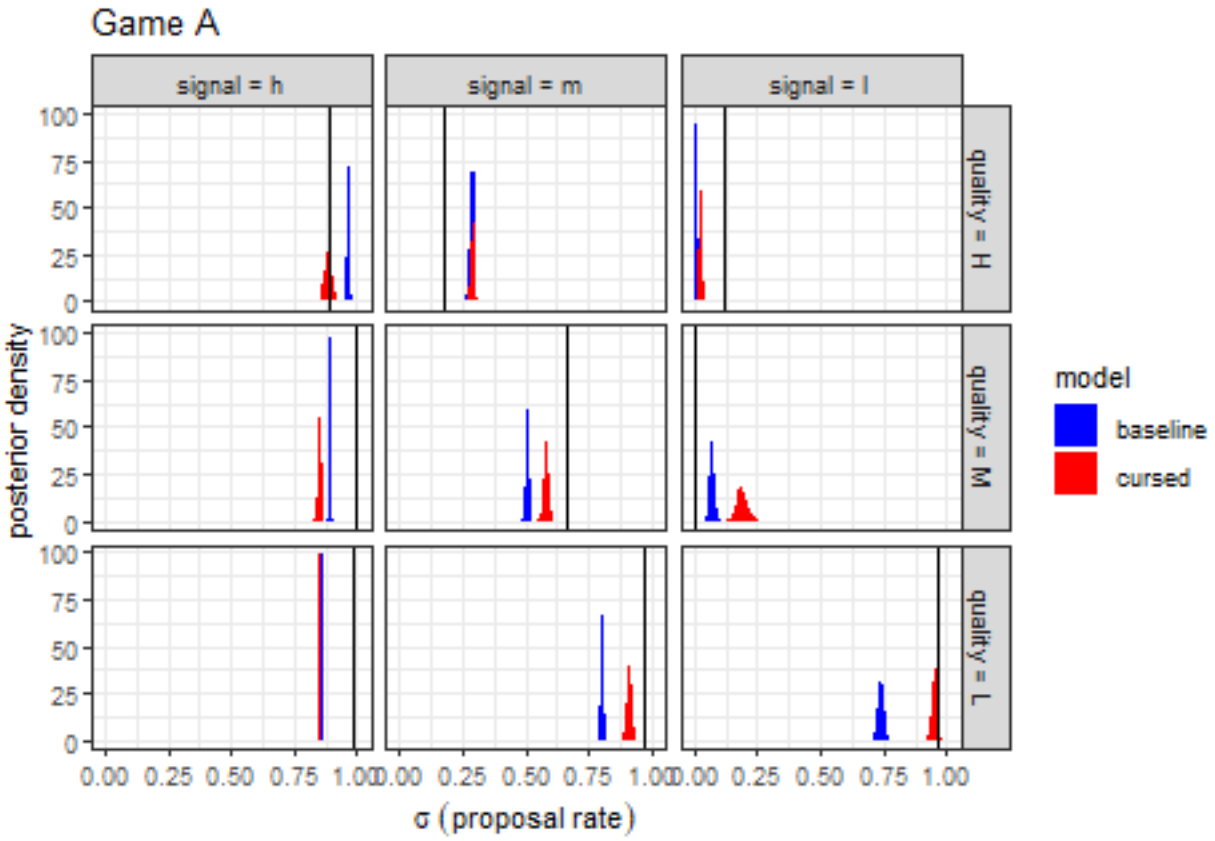


Figure 46: Posterior distribution of mixed strategies (histograms) and empirical choice frequencies (vertical black lines) for Game A

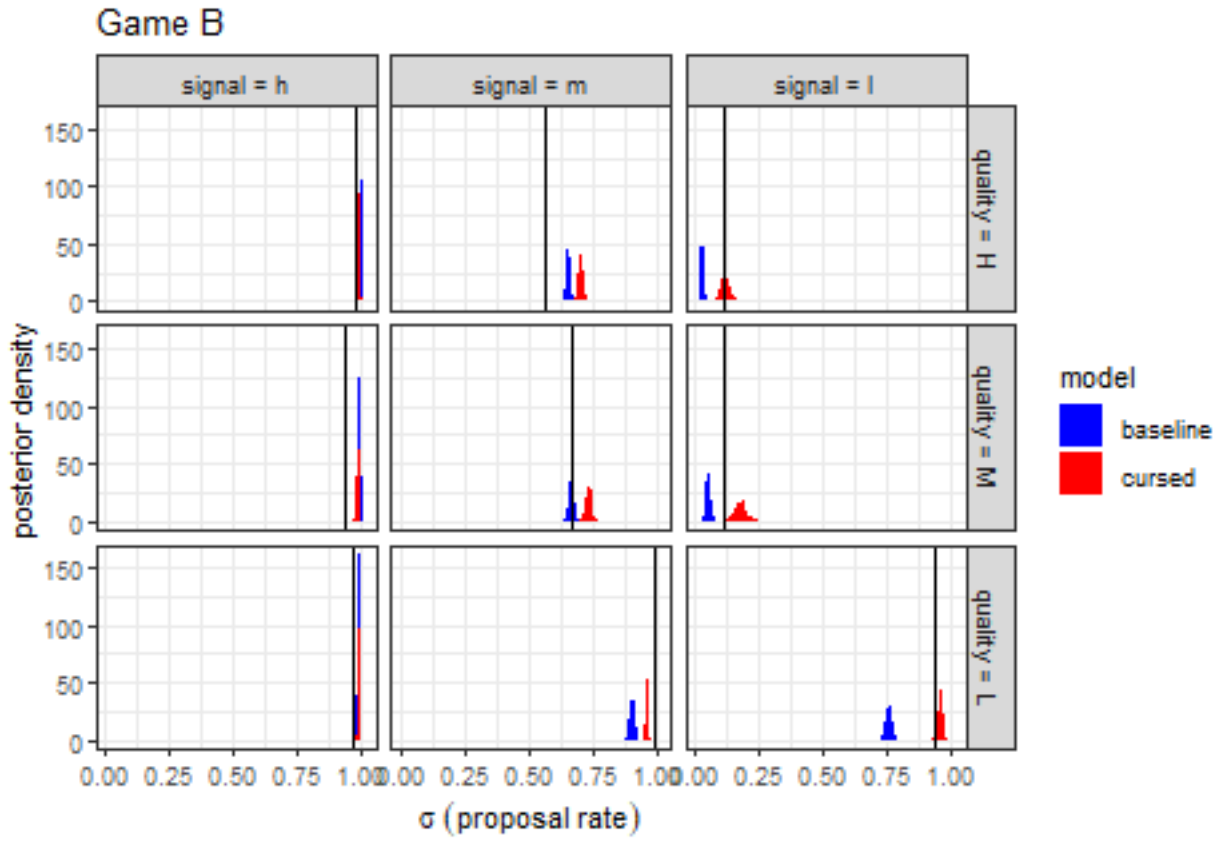


Figure 47: Posterior distribution of mixed strategies (histograms) and empirical choice frequencies (vertical black lines) for Game B

```
## [1] -811.7324
```

```
kfoldCV$cursed%*%rep(1,6) |> mean()
```

```
## [1] -758.2627
```

16.6 *R* code used in this chapter

```
library(tidyverse)
library(rstan)
rstan_options(auto_write = TRUE)
options(mc.cores = parallel::detectCores())

RERUN<-FALSE

# Here I am following the replication package in importing the data -----------

# set min round ----------------------------------------------------
min_round <- 21

D<-"Data/BochetMagnani2024/data_analysis/data/Main_data_base_game.csv" |>
  read.csv() %>%
  filter(subsession.round_number>=min_round) %>%
  mutate(player.type=factor(player.type)) %>%
  mutate(player.type=fct_recode(player.type,
                                "H"="1",
                                "M"="2",
                                "L"="3")) %>%
  mutate(player.signal=factor(player.signal)) %>%
  mutate(player.signal=fct_recode(player.signal,
                                  "h"="1",
                                  "m"="2",
                                  "l"="3")) %>%
  mutate(subsession.game_name=factor(subsession.game_name)) |>
  # now for JB's but coding up things to pass to Stan --------------------------
  mutate(
    type_code = paste0(player.type,player.signal) |>
      factor(levels = c("Hh","Mh","Lh","Hm","Mm","Lm","Hl","Ml","Ll")) |>
      as.numeric(),

    game_code = ifelse(subsession.game_name=="A",1,2),

    action = player.choice,

    uid = participant.code |> factor() |> as.numeric(),

    ugroupid = paste(session.code,group.id_in_subsession) |> factor() |> as.numeric()
  ) |>
  ungroup() |>
  mutate(
    rownum = 1:n()
  )

# My attempt to replicate Figure 5
```

```
dplt<-D |>
  filter(game_code==1) |>
  group_by(player.type,player.signal) |>
  summarize(
    `Proposal rate` = mean(action)
  )

(
  ggplot(dplt,aes(x=player.type,y=`Proposal rate`,fill = player.signal))
  +geom_bar(stat="identity", position=position_dodge())
)

# and Figure 6

dplt<-D |>
  filter(game_code==2) |>
  group_by(player.type,player.signal) |>
  summarize(
    `Proposal rate` = mean(action)
  )

(
  ggplot(dplt,aes(x=player.type,y=`Proposal rate`,fill = player.signal))
  +geom_bar(stat="identity", position=position_dodge())
)


# I'm happy with that

dStan<-list(
  N = dim(D)[1],

  type = D$type_code,
  game = D$game_code,
  action = D$action,

  nUseData = 0,
  UseData = array(dim=0),

  prior_lambda = c(-3.45,1.76),

  maxiter = 100,
  tol = 1e-6,
  nlxgrid = 100
)




model<-"Code/BM2024/baseline_qre.stan" |>
  stan_model()
```

```
cursed_model<-"Code/BM2024/cursed_qre.stan" |>
  stan_model()

# Check that the thing works


Fit<-model |>
  sampling(data=dStan,chains=1,iter=10,
           algorithm = "Fixed_param",init = list(list(lambda = 1))
           )
# trace out a grid of lambda to get the QRE correspondence
file<-"Code/BM2024/QRE.csv"

if (!file.exists(file) | RERUN) {
  QRE<-tibble()

  for (lambda in  (0.001*10^seq(0,4,by=0.01))) {

    Fit<-model |>
      sampling(data=dStan,chains=1,iter=1,
               algorithm = "Fixed_param",init = list(list(lambda = lambda))
      )

    SIGMA<-extract(Fit)$sigma[1,,]

    QRE<-rbind(QRE,
               tibble(
                 lambda = lambda,
                 sigma1=SIGMA[1,],
                 sigma2=SIGMA[2,],
                 logL = extract(Fit)$logL_test
               )|>
                 mutate(type=1:n())
               )

  }

  quality<-rep(c("H","M","L"),3)
  signal<-c(rep("h",3),rep("m",3),rep("l",3))

  QRE<-QRE |>
    mutate(
      quality = quality[type],
      signal = signal[type]
    ) |>
    pivot_longer(
      cols = sigma1:sigma2,
      names_to = "var",
      values_to="sigma"
    ) |>
    mutate(
      game = c("A","B")[var |> parse_number()]
    )
```

```
  QRE |>
    write.csv(file)

  (
    ggplot(QRE,aes(x=lambda,y=sigma,linetype=game))
    +facet_grid(quality~signal)
    +geom_line()
    +scale_x_continuous(trans="log10")
    +theme_bw()
    +xlab(expression(lambda))
    +ylab(expression(sigma))
  )

  plotThis<-QRE |>
    filter(game=="A",quality=="H",signal=="h")

  (
    ggplot(plotThis,aes(x=lambda,y=logL))
    +geom_line()
    +scale_x_continuous(trans="log10")
    +theme_bw()
    +coord_cartesian(ylim = c(-3000,0))
  )

}
# Estimate the baseline model

file<-"Code/BM2024/Fit_baseline_qre.rds"

if (!file.exists(file) | RERUN) {
  print("fitting baseline model")
  dStan<-list(
    N = dim(D)[1],

    type = D$type_code,
    game = D$game_code,
    action = D$action,

    nUseData = dim(D)[1],
    UseData = D$rownum,#UseData = array(dim=0),

    prior_lambda = c(-3.45,1.76),

    maxiter = 100,
    tol = 1e-3,
    nlxgrid = 100
  )

  Fit<-model |>
    sampling(data=dStan,chains=4,iter=2000,
             seed=42
    )
```

```
  Fit |>
    saveRDS(file)

}


# Estimate the baseline model

file<-"Code/BM2024/Fit_cursed_qre.rds"

if (!file.exists(file) | RERUN) {
  print("fitting cursed model")
  print("https://www.youtube.com/watch?v=CI1-74VQgUk")
  dStan<-list(
    N = dim(D)[1],

    type = D$type_code,
    game = D$game_code,
    action = D$action,

    nUseData = dim(D)[1],
    UseData = D$rownum,

    prior_lambda = c(log(0.1),1),
    prior_chi = c(1,1),

    maxiter = 100,
    tol = 1e-3,
    nlxgrid = 100
  )

  Fit<-cursed_model |>
    sampling(data=dStan,chains=4,iter=2000,
             seed=42
    )

  Fit |>
    saveRDS(file)

}

# k-fold CV ------------------------------------------------------------------

file<-"Code/BM2024/kfoldCV.rds"

if (!file.exists(file) | RERUN) {

  ll_baseline<-c()
  ll_cursed<-c()

  for (gg in sort(unique(D$ugroupid))) {

    print(paste("estimating leaving out group",gg))
```

```
    UseData<-D |>
      filter(ugroupid!=gg)

    dStan<-list(
      N = dim(D)[1],

      type = D$type_code,
      game = D$game_code,
      action = D$action,

      nUseData = dim(UseData)[1],
      UseData = UseData$rownum,

      prior_lambda = c(-3.45,1.76),
      prior_chi = c(1,1),

      maxiter = 100,
      tol = 1e-3,
      nlxgrid = 100
    )

    print("     Estimating baseline model")

    Fit_baseline<-model |>
      sampling(data=dStan,chains=4,iter=2000,
               seed=42
      )

    print("     Estimating cursed model")

    Fit_cursed<-cursed_model |>
      sampling(data=dStan,chains=4,iter=2000,
               seed=42
      )


    ll_baseline<-cbind(ll_baseline,extract(Fit_baseline)$logL)
    ll_cursed<-cbind(ll_cursed,extract(Fit_cursed)$logL)
  }

  list(
    baseline = ll_baseline,
    cursed = ll_cursed
  ) |>
    saveRDS(file)

}
```

17 Application: Level-k models

In this chapter, I will show you a few ways of estimating a level-k model (Stahl and Wilson 1994) using data from a guessing game (Costa-Gomes and Crawford 2006). The level-k model is particularly useful for

modeling initial play in games (or play without feedback) because it is a plausible way that participants might reason their way through their decision problem in order to choose an action.

Computationally, the model poses a slight problem for us because the parameter of interest, k, is an integer, and *Stan* can't cope with integer parameters. Fortunately, it is very easy (albeit somewhat slow) to estimate a model assuming a given value of k. From here, we can build up a model-averaged k that has a nice interpretation. But, perhaps more satisfyingly, this really lends itself to a finite mixture model, where we estimate the distribution of k in the population. The final product also has a hierarchical specification for some nuisance parameters, which we don't necessarily care about directly, but nevertheless have to be estimated alongside the quantities that we do care about.

17.1 Data and game

Participants in Costa-Gomes and Crawford (2006) played sixteen 2-player guessing games without feedback. In each game, player $i \in \{1, 2\}$ chose a number $x^i \in [a^i, b^i]$.[72] Their payoff functions rewarded choices that were closer to p^i multiplied by their opponent's choice. Specifically, their payoff function was:

$$\pi^i(x^i, x^j) = \max\{0, 200 - |x^i - p^i x^j|\} + \max\{0, 100 - |x^i - p^i x^j|/10\}$$

Note that this payoff function is single-peaked at $x^i = p^i x^j$, where the player receives a payoff of 300.

When I got my hands on these data, one of the first things I noticed was that there was a *lot* of rounding of participants' choices. Specifically, there were very few choices that were not multiples of 10. The best way for me to visualize this was to plot a histogram of the digit in the ones place of participants' choices, which is shown in Figure 48:

```
D<-"Code/CGC2006/CGC2006cleaned.rds" |>
  readRDS() |>
  mutate(
    intchoice = choice |> round(0),
    onesplace = intchoice-10*floor(intchoice/10)
  )

(
  ggplot(D,aes(x=onesplace))
  +geom_histogram(bins=10,binwidth=1)
  +theme_bw()
  +scale_x_continuous(breaks=0:10)
  +xlab("Digit in ones place of players' choices")
)
```

This is plausible because participants in a typical game could choose between a few hundred numbers: restricting attention to just the multiples of ten probably cut out a lot of the cost of making a decision in this experiment. I therefore decided to deal exclusively with the rounded data. For example, for Game 1 player 1 has to choose a number between 100 and 500 that they think will be closest to 70% of their opponent's choice (which must be between 100 and 900). I divide these bounds and participants' choices by 10, round them to the nearest integer, and treat these as the data. Therefore the payoff function becomes:

$$\pi^i(x^i, x^j) = \max\{0, 200 - 10|x^i - p^i x^j|\} + \max\{0, 100 - |x^i - p^i x^j|\}$$

which means that if Player 2 chooses $x^2 = 50$ (really $x^2 = 500$ before dealing with rounded numbers), Player 1's payoff function looks like this:

[72] Participants in the experiment were in fact allowed to choose outside these bounds, but such choices were adjusted to the closer of the bounds. For simplicity, I describe the game in terms of these "adjusted" choices.

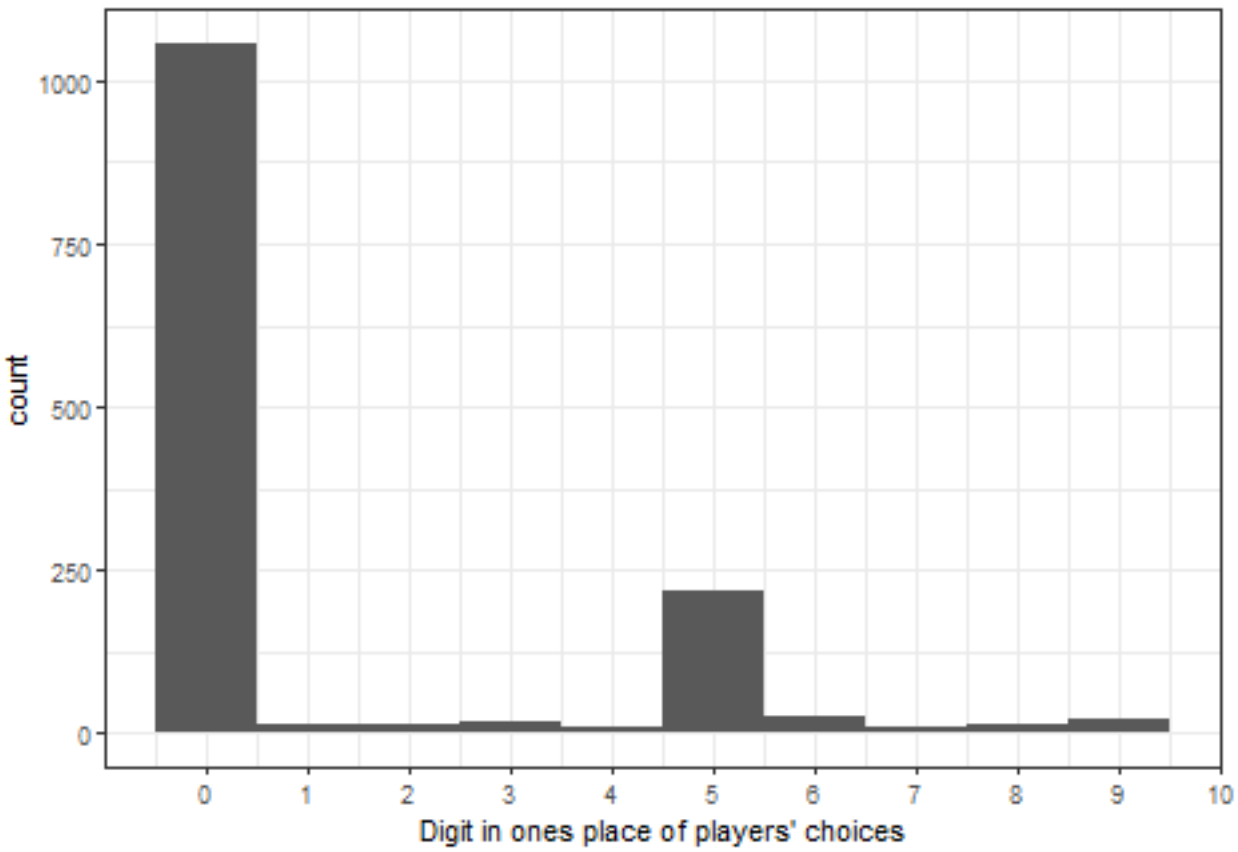


Figure 48: Prevalence of rounding.

```
# payoff parameters for Player 1 in Game 1
p1<-0.7
lb1<-10
ub1<-50

# opponent's choice
x_other<-50


d<-tibble(
  # choice set
  x = seq(lb1,ub1,by=1)
) |>
  rowwise() |>
  mutate(
    u = max(c(0,200-abs(x-p1*x_other)*10))+max(c(0,100-abs(x-p1*x_other)))
  )

(
  ggplot(d,aes(x=x,y=u))
  +geom_path()
  +theme_bw()
  +ylim(c(0,300))
)
```

17.2 The level-k model

17.2.1 The deterministic component of the model

The level-k model (Stahl and Wilson 1994) is a model of introspection in games. The model assumes that there are (possibly infinite) player types (or "levels") indexed by non-negative integers $k = 0, 1, 2, \ldots$. Tye level-0 type is usually assumed to randomize their action uniformly over their action space. The level-1 type best responds to the level-0 type, and so on, so that the level-k type best responds to the level-$(k-1)$ type.[73]

When we take a level-k model to our data, we are usually interested in estimating k. That is, we want to comment on the number of steps of strategic reasoning that our participants are using.[74] Since k can only take on integers, and *Stan* cannot deal with integer parameters, we need to be careful with how we set this up. I will walk you through a few ways of doing this. These will include Bayesian model averaging, where we estimate the model for every possible value of k under consideration, and a mixture specification, where we estimate the fraction of participants who have each value of k. Whichever way seems more appealing to you, we will have to assume that participants can only be one of a finite number of possible values of k. In the Bayesian model averaging case, this is because we cannot estimate "infinity" models! In the mixture model case, this is because we cannot have "infinity" behavioral types in our model. Practically in these games, as k gets large, the prediction approaches Nash equilibrium, and so at some point a k will be "large enough" that behavior should not be too different for any k larger than this. I will proceed with estimating everything assuming that $k \leq 5$, but I have written my code so that this upper bound can be easily changed. If you're playing along at home and your CPU has nothing better to do, all you need to do is change the `kmax` variable to something larger.

[73] There are many permutations of how the level-k model is set up. I am using this one here for simplicity. Stahl and Wilson (1994), for example, assumes that the level-2 type best responds to the mixture of level-0 and level-1 types, rather than just the level-1 type. This alternative assumption is in principle implementable in both the mixture and hierarchical specifications outlined below, but would be more computationally intensive.

[74] We need to be careful here. A participant may be *able* to do many steps of strategic reasoning, but might believe that others only do a few. For example, suppose that a participant can reason their way through all of the steps (possibly infinitely many) to get to the Nash equilibrium prediction, but they believe that their opponent will make $k = 2$ steps. This participant would then behave as if they are a level-3 type.

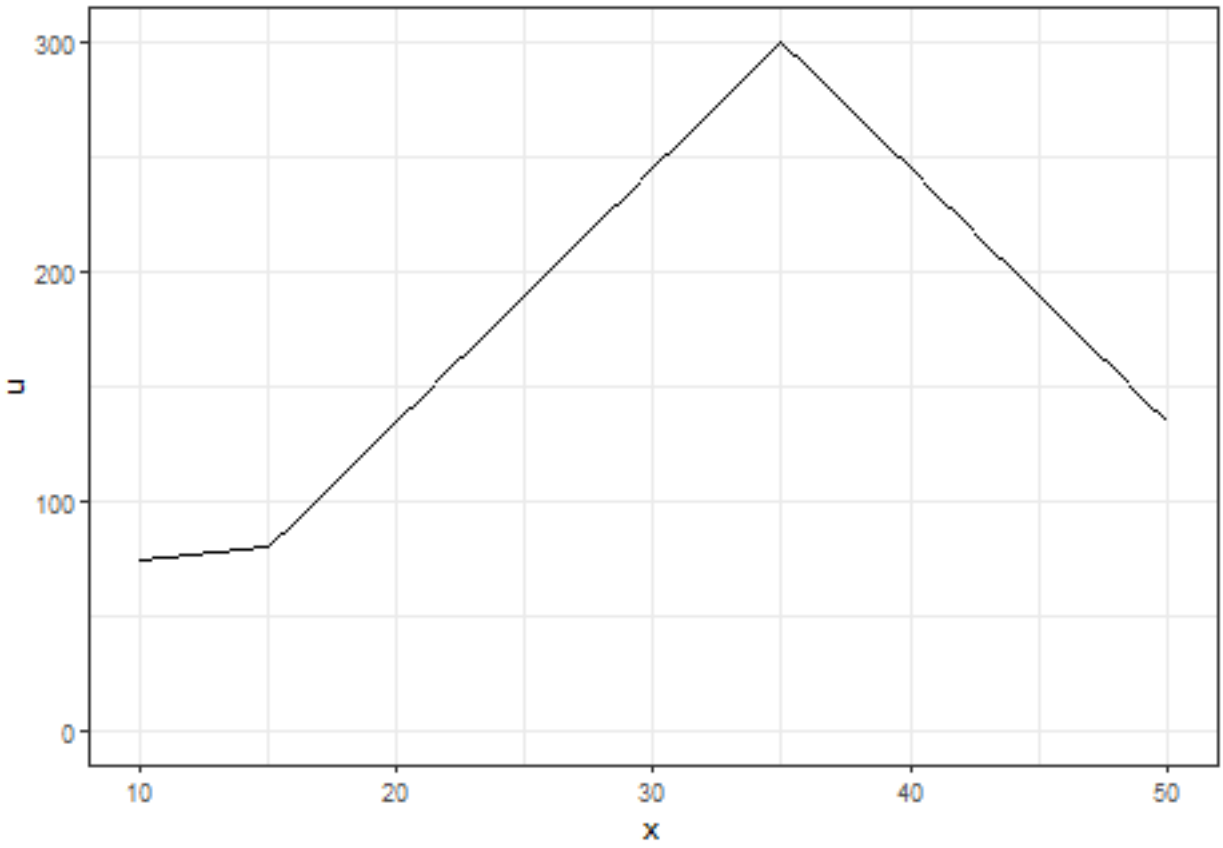


Figure 49: An example of a payoff function.

17.2.2 Exact and probabilistic play

The level-k model (as I have described it above) makes deterministic predictions. This is going to be a problem for our Bayesian model, as we need to have *probabilistic* predictions in order to write down a likelihood function. In fact, Costa-Gomes and Crawford (2006) note that a substantial fraction of their participants choose *exactly* according to one of their models under consideration, at least for a subset of the the games played. That is, it appears that some participants are behaving exactly according to the level-k model, and we should therefore take a model to the data that respects this kind of behavior. The solution Costa-Gomes and Crawford (2006) use, which I also adopt here, is to use a "spike logit" model, which assumes that players *best* respond to their belief aboth their opponent's (possibly mixed) strategy with probability $1-\epsilon$, and with probability ϵ they *logit* respond to this belief. For the games studied here, which have relatively large action spaces, this makes for a mostly smooth probabilistic best response, except for a large mass point on the optimal action.

So how do we code this into a likelihood function? Let's parameterize the model as $\rho = 1-\epsilon \in (0,1)$, the probability that a participant chooses their optimal action, and $\lambda > 0$, logit choice precision. Then let $u_t(a \mid k)$ be the player's expected utility in game t given that they are a level-k type, and $a_t^*(k)$ is their optimal action given that they are a lvele-k type. We can then split up the likelihood contribution into two cases:

1. if they choose their optimal action, then that choice could have been a best response *or* they just happened to logit respond with this action, or
2. if they did not choose their optimal action, then they must have logit responded.

This makes our log-likelihood for an individual choice y_t in Game t:

$$\log p(y_t \mid \rho, \lambda, k) = \begin{cases} \log\left(\rho + (1-\rho)\frac{\exp(\lambda u(y_t|k))}{\sum_a \exp(\lambda u(a|k))}\right) & \text{if } y_t = a_t^*(k) \\ \log\left((1-\rho)\frac{\exp(\lambda u(y_t|k))}{\sum_a \exp(\lambda u(a|k))}\right) & \text{otherwise} \end{cases}$$

17.3 Assigning probabilities to types for each participant separately

One approach to the data, which is probably the closest in this chapter to its original treatment in Costa-Gomes and Crawford (2006), is to estimate the level-k model for each participant separately, assuming various different values of k, then using some kind of model evaluation to select the value of k that best characterizes that participant. In this section, I estimate the model for each participant assuming $k = 1, 2, 3, 4, 5$, and then use Bayes factors and model averaging to determine which k best characterizes their behavior.

17.3.1 The *Stan* program

Here is the *Stan* program I came up with. Importantly, note that each level-k strategy can be pre-computed, which is going to speed things up substantially. You can see this done in the `transformed data` block. The predictions are stored in the arrays `LKpredictions1` for player 1, and `LKpredictions2` for player 2.

Another thing to note is that I am incementing the target by the full likelihood, and not just the kernel. Hence, the extensive use of `target += ...`. This is so that we can compute the log marginal likelihood, which will be needed for model comparison.

```
functions {

  vector payofffun(vector x1,row_vector x2,real p1) {

    /*
    x1 is the action space of the first player

    x2 is the distribution of actions of the second player. That is, for level 0
    player 2, it is the action space (which gets averaged over a uniform
```

```
    distribution), and for level k>0 is just populated by the prediction of this
    type
    */

    int n1 = size(x1);
    int n2 = size(x2);

    matrix[n1,n2] err = fabs(rep_matrix(x1,n2)-p1*rep_matrix(x2,n1));

    vector[n1] pay = (fmax(0.0,200-err*10)+fmax(0.0,100-err))*rep_vector(1.0/n1,n2);

    return pay;


  }

  int which_max(vector x,real tol) {
    /*
    Returns the index of the vector x which corresponds to the maximum in x
    Returns the largest index if there are more than one element equal to this
    */
    int n = size(x);

    int ind = 0;

    for (ii in 1:n) {
      if (abs(x[ii]-max(x))<tol) {
        ind = ii;
      }
    }

    return ind;




  }


}

data {

  int ngames; // number of games

  int k; // level k to evaluate

  // game data
  vector[ngames] P1; // target for player 1
  vector[ngames] P2; // target for player 2
  int LB1[ngames]; // lower bound for player 1
  int UB1[ngames]; // upper bound for player 1
  int LB2[ngames]; // lower bound for player 2
```

```
  int UB2[ngames]; // upper bound for player 2

  // player 1 choice
  int choice1[ngames];

  real<lower=0> prior_probExact[2]; // Beta prior
  real prior_lambda[2]; // lognormal prior

  // tolerance for identifying a maximum in function which_max(,)
  real<lower=0> tol;



}

transformed data {

  int choice1_i[ngames];

  // pre-compute level-k predictions

  vector[k] LKpredictions1[ngames];
  vector[k] LKpredictions2[ngames];




  for (gg in 1:ngames) {
    real p1 = P1[gg];
    real p2 = P2[gg];
    int lb1 = LB1[gg];
    int ub1 = UB1[gg];
    int lb2 = LB2[gg];
    int ub2 = UB2[gg];

    int n1 = ub1-lb1+1;
    int n2 = ub2-lb2+1;

    choice1_i[gg] = choice1[gg]-lb1+1;

    // choice sets
    vector[n1] X1 = linspaced_vector(n1,lb1,ub1);
    row_vector[n2] X2 = linspaced_row_vector(n2,lb2,ub2);

    // choices to get uniform distribution over actions
    vector[n1] x1 = linspaced_vector(n1,lb1,ub1);
    row_vector[n2] x2 = linspaced_row_vector(n2,lb2,ub2);


    for (kk in 1:k) {

      vector[n1] payoff1 = payofffun(X1,x2,p1);
      vector[n2] payoff2 = payofffun(X2',x1',p2);
```

```
      x1 = rep_vector(X1[which_max(payoff1,tol)],n1);
      x2 = rep_row_vector(X2[which_max(payoff2,tol)],n2);

      LKpredictions1[gg][kk] = X1[which_max(payoff1,tol)];
      LKpredictions2[gg][kk] = X2[which_max(payoff2,tol)];
    }

  }

}



parameters {

  // probability the player chooses exactly according to the level-k model
  real<lower=0,upper=1> probExact;
  // logit choice precision for the rest of the choices
  real<lower=0> lambda;

}

model {

  for (gg in 1:ngames) {
    real p1 = P1[gg];
    real p2 = P2[gg];
    int lb1 = LB1[gg];
    int ub1 = UB1[gg];
    int lb2 = LB2[gg];
    int ub2 = UB2[gg];

    int n1 = ub1-lb1+1;
    int n2 = ub2-lb2+1;

    // choice sets for Player 1
    vector[n1] X1 = linspaced_vector(n1,lb1,ub1);

    row_vector[n2] x2;

    // action of Player 2
    if (k==1) {
      x2 = linspaced_vector(n2,lb2,ub2)';
    } else {
      x2 = rep_row_vector(LKpredictions2[gg][k-1],n2);
    }

    // Player 1 payoff
    vector[n1] payoff1 = payofffun(X1,x2,p1);

    if (choice1[gg]==LKpredictions1[gg][k]) {
    // If the choice matches exactly with the prediction, then ...
     target += log(
```

```
       probExact // best respond probability
        +(1.0-probExact)*softmax(lambda*payoff1)[choice1_i[gg]] // logit response
       );

    } else {
      // if the choice does not match exactly with the prediction, then ...
      target += log(
       (1.0-probExact)*softmax(lambda*payoff1)[choice1_i[gg]]
      );
    }

  }



  // priors
  target += beta_lpdf(probExact | prior_probExact[1],prior_probExact[2]);
  target += lognormal_lpdf(lambda | prior_lambda[1],prior_lambda[2]);


}
```

17.3.2 Prior calibration

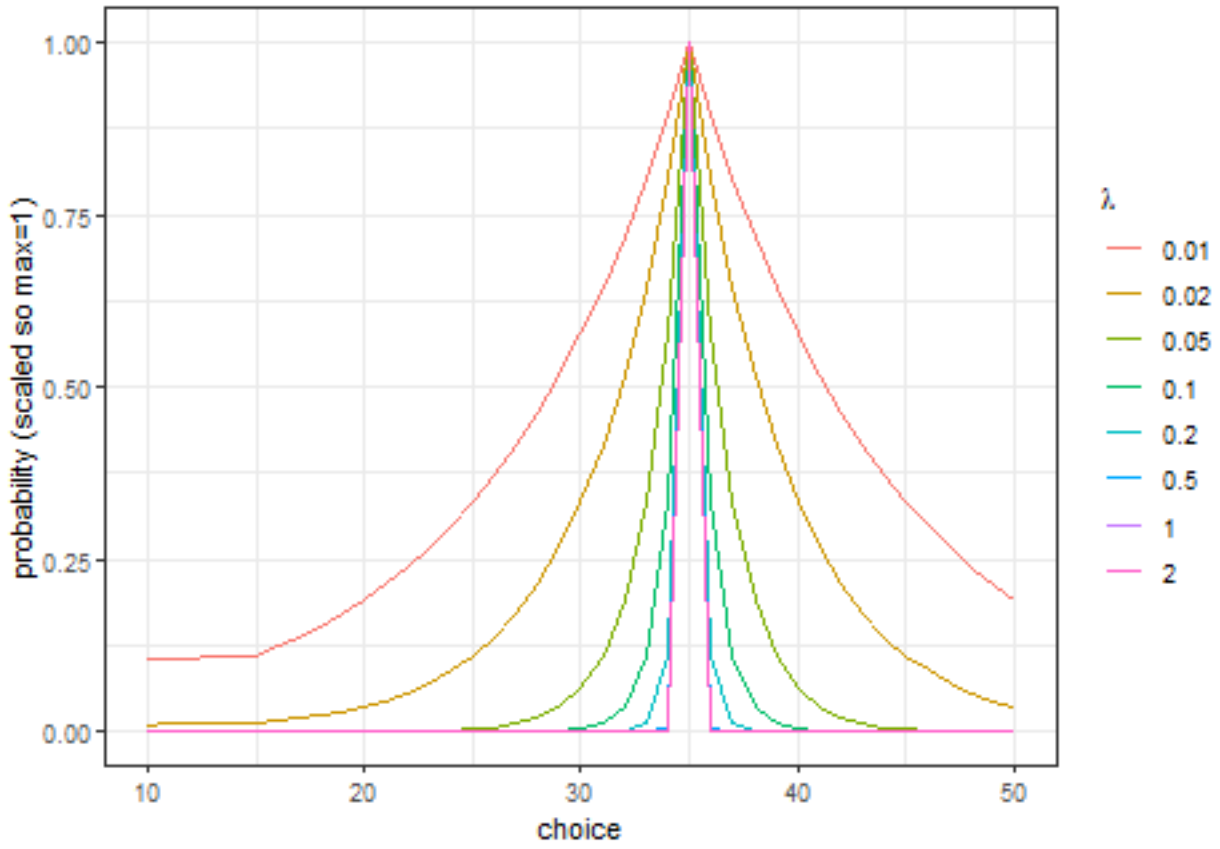
The model has two parameters: $\rho \in (0,1)$ (the probability that the player chooses their best response), and $\lambda > 0$ (logit choice precision).

As ρ is restricted to the unit interval, a Beta prior makes sense here. I ended up unsing the uniform distribution, or $\rho \sim \text{Beta}(1,1)$ for this prior.

For λ, a log-normal distribution makes sure that the parameter is non-negative, so $\log \lambda \sim N(m_\lambda, s_\lambda^2)$. For an idea of what might be a good choice of m_λ and s_λ, let's look at the logit response in Game 1 to player 2 choosing `x_other=500` (as we did for the payoff function above in Figure 49).

```
logitResponse<-expand.grid(
  x = seq(lb1,ub1,by=1),
  lambda = c(0.01,0.02,0.05,0.1,0.2,0.5,1,2)
) |>
  rowwise() |>
  mutate(
    u = max(c(0,200-abs(x-p1*x_other)*10))+max(c(0,100-abs(x-p1*x_other)))
  ) |>
  ungroup() |>
  group_by(lambda) |>
  mutate(
    u = u-min(u),
    pr = exp(lambda*u)/sum(lambda*u),
    pr_scaled = pr/max(pr)
  )

(
  ggplot(logitResponse,aes(x=x,y=pr_scaled,color = as.factor(lambda)))
  +labs(color = expression(lambda),x = "choice",y="probability (scaled so max=1)")
  +geom_path()
  +theme_bw()
```

```
)
```



I then pin down the 2.5th and 97.5th percentiles of the prior with the most extreme cases of this plot:

$$
\begin{aligned}
\log 0.01 &= m_\lambda - 1.96 s_\lambda \\
\log 2 &= m_\lambda + 1.96 s_\lambda \\
2m_\lambda &= \log 0.01 + \log 2 \\
m_\lambda &= \frac{\log 0.01 + \log 2}{2} \approx -1.96 \\
2 \times 1.96 s_\lambda &= \log 2 - \log 0.01 \\
s_\lambda &= \frac{\log 2 - \log 0.01}{2 \times 1.96} \approx 1.35 \\
\implies \log \lambda &\sim N(-1.96, 1.35^2)
\end{aligned}
$$

17.3.3 Results

In this section, I estimate one model for each participant, and for each level $k \in \{1, 2, 3, 4, 5\}$. This takes a while, but runs without any errors with *RStan*'s default options.[75]

```
Estimates<-"Code/CGC2006/EstimatesLevelkRepAgent.rds" |>
  readRDS()

# whoops! I forgot to add in the parameter names into the estimation code.
```

[75] If you haven't found enough shrines to get enough heart containers for the Master Sword, now is your time! This took about a day on my laptop.

```r
# Fortunately this information is stored in the rownames of the estimates
Estimates$parameter<-gsub('[[:digit:]]+', '',rownames(Estimates))


Estimates<-Estimates |>
  group_by(id,parameter) |>
  mutate(
    postprob = exp(logml-max(logml))/sum(exp(logml-max(logml))),
    meantype = sum(k*postprob),
    meanpar = sum(mean*postprob)
  )

(
  ggplot(Estimates |> filter(parameter=="probExact"),aes(x=meantype,y=meanpar))
  +geom_point()
  +xlab("Level-k type")
  +ylab("Probability of choosing exactly accorindg to the model")
  +theme_bw()
)
```

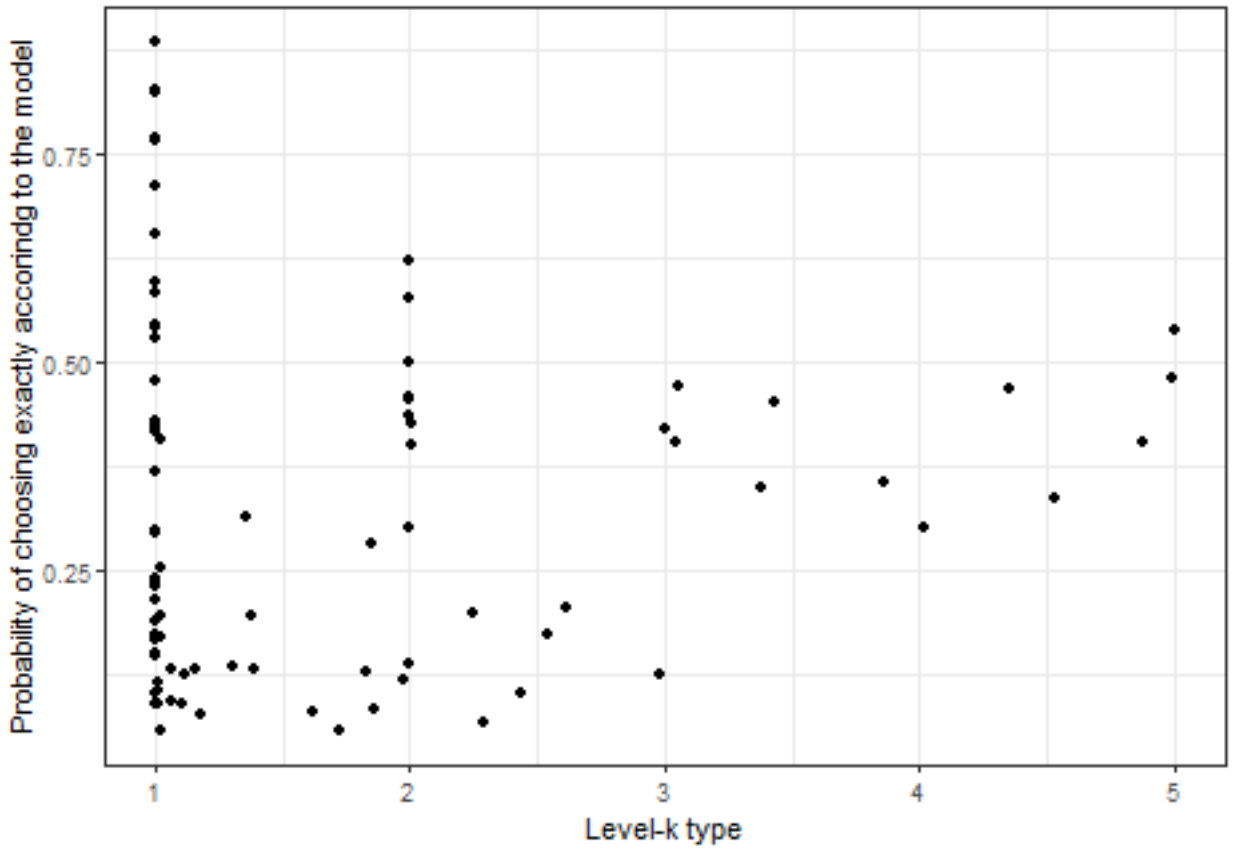


Figure 50: Bayesian model averages for the level-k type (horizontal axis) and the probability of choosing exactly accorindg to the model (vertical axis).

17.4 Doing the averaging within one program

The previous section provided us with a rather tedious way of estimating k for each participant: we estimated the model once for every k under consideration, then averaged over these models. While there is nothing wrong with estimating k this way, it is somewhat inefficient. To see this, note that we could have written down a model prior over the different values of k, and then estimated ρ and λ assuming that k is drawn from this prior. To keep things simple, I will stick with a discrete uniform prior over k, so that each $k \in \{1, 2, 3, 4, 5\}$ has a 20% prior probability of being the true k. Letting $\{\psi_k\}_{k=1}^5$ be these prior probabilities, we can write the likelihood contribution for a participant's 16 decisions as:

$$\begin{aligned}
\log p(y_t \mid \rho, \lambda) &= \log \left(\sum_{k=1}^{5} \psi_k \prod_{t=1}^{16} p(y_t \mid \rho, \lambda, k) \right) \\
&= \log \left(\sum_{k=1}^{5} \exp \left(\log \psi_k + \sum_{t=1}^{16} \log p(y_t \mid \rho, \lambda, k) \right) \right)
\end{aligned}$$

This is enough information to estimate parameters ρ and λ, but what we really care about i k. So how can we go about estimating this? What we first need to do is to take our prior over k, and update it based on decisions, ρ, and λ. That is, using Bayes' rule:

$$\begin{aligned}
p(k \mid y, \rho, \lambda) &= \frac{p(y \mid k, \rho, \lambda) p(k)}{p(y \mid \rho, \lambda)} \\
&\propto p(y \mid k, \rho, \lambda) p(k) \\
\implies p(k \mid y, \rho, \lambda) &= \frac{p(y \mid k, \rho, \lambda) p(k)}{\sum_{\kappa=1}^{5} p(y \mid \kappa, \rho, \lambda) p(\kappa)}
\end{aligned}$$

Note that we can compute everything on the right-hand side once we have posterior draws of ρ and λ, so we can do this in the `generated quantities` block in *Stan*! The posterior means of these quantities will then be marginal posterior probabilities that we are after. That is, taking the expectation over $\rho, \lambda \mid y$:

$$E_{\rho,\lambda|y} \left[p(k \mid y, \rho, \lambda) \right] = p(k \mid y)$$

Further inspection of the final expression for $p(k \mid y, \rho, \lambda)$ also reveals a useful similarity with our expression for the likelihood. In particular:

$$\begin{aligned}
p(y \mid k, \rho, \lambda) p(k) &= \exp(\log p(y \mid k, \rho, \lambda) + \log p(k)) \\
&= \exp(\log \psi_k + \log p(y \mid k, \rho, \lambda)) \\
\implies p(k \mid y, \rho, \lambda) &= \frac{\exp(\log \psi_k + \log p(y \mid k, \rho, \lambda))}{\sum_{\kappa=1}^{5} \exp(\log \psi_\kappa + \log p(y \mid \kappa, \rho, \lambda))}
\end{aligned}$$

which is the `softmax` of the vector $\{\log \psi_k + \log p(y \mid k, \rho, \lambda)\}_{k=1}^5$. All of this is to say, if we compute $\log \psi_k + \log p(y \mid k, \rho, \lambda)$ in the `transformed parameters` block, we can then use it in the `model` block to increment the likelihood, and again in the `generated quantities` block to compute these posterior probabilities.

Once we have an expression for $p(k \mid y, \rho, \lambda)$, it is straightforward to estimate k by taking the expectation in the `generated quantities` block. That is:

$$E \left[k \mid y, \rho, \lambda \right] = \sum_{k=1}^{5} k p(k \mid y, \rho, \lambda)$$

And then the posterior mean of this will be the *marginal* posterior mean of k, which is exactly what we are after:

$$E[k \mid y] = E\left[E[k \mid y, \rho, \lambda] \mid y\right]$$

17.4.1 The *Stan* program

```
functions {

  vector payofffun(vector x1,row_vector x2,real p1) {

    /*
    x1 is the action space of the first player

    x2 is the distribution of actions of the second player. That is, for level 0
    player 2, it is the action space (which gets averaged over a uniform
    distribution), and for level k>0 is just populated by the prediction of this
    type
    */

    int n1 = size(x1);
    int n2 = size(x2);

    matrix[n1,n2] err = fabs(rep_matrix(x1,n2)-p1*rep_matrix(x2,n1));

    vector[n1] pay = (fmax(0.0,200-err*10)+fmax(0.0,100-err))*rep_vector(1.0/n1,n2);

    return pay;


  }

  int which_max(vector x,real tol) {

    int n = size(x);

    int ind = 0;

    for (ii in 1:n) {
      if (abs(x[ii]-max(x))<tol) {
        ind = ii;
      }
    }

    return ind;




  }


}
```

```
data {

  int ngames; // number of games

  int kmax; // max level-k to evaluate

  // game data
  vector[ngames] P1;
  vector[ngames] P2;
  int LB1[ngames];
  int UB1[ngames];
  int LB2[ngames];
  int UB2[ngames];

  // player 1 choice
  int choice1[ngames];

  real<lower=0> prior_probExact[2]; // Beta prior
  real prior_lambda[2]; // lognormal prior
  vector[kmax] prior_k; // categorical prior

  // tolerance for identifying a maximum in function which_max(,)
  real<lower=0> tol;



}

transformed data {

  int choice1_i[ngames];

  // pre-compute level-k predictions

  vector[kmax] LKpredictions1[ngames];
  vector[kmax] LKpredictions2[ngames];




  for (gg in 1:ngames) {
    real p1 = P1[gg];
    real p2 = P2[gg];
    int lb1 = LB1[gg];
    int ub1 = UB1[gg];
    int lb2 = LB2[gg];
    int ub2 = UB2[gg];

    int n1 = ub1-lb1+1;
    int n2 = ub2-lb2+1;

    choice1_i[gg] = choice1[gg]-lb1+1;
```

```
    // choice sets
    vector[n1] X1 = linspaced_vector(n1,lb1,ub1);
    row_vector[n2] X2 = linspaced_row_vector(n2,lb2,ub2);

    // choices to get uniform distribution over actions
    vector[n1] x1 = linspaced_vector(n1,lb1,ub1);
    row_vector[n2] x2 = linspaced_row_vector(n2,lb2,ub2);


    for (kk in 1:kmax) {

      vector[n1] payoff1 = payofffun(X1,x2,p1);
      vector[n2] payoff2 = payofffun(X2',x1',p2);

      x1 = rep_vector(X1[which_max(payoff1,tol)],n1);
      x2 = rep_row_vector(X2[which_max(payoff2,tol)],n2);

      LKpredictions1[gg][kk] = X1[which_max(payoff1,tol)];
      LKpredictions2[gg][kk] = X2[which_max(payoff2,tol)];
    }

  }

}



parameters {

  // probability the player chooses exactly according to the level-k model
  real<lower=0,upper=1> probExact;
  // logit choice precision for the rest of the choices
  real<lower=0> lambda;

}

transformed parameters {


  vector[kmax] like_levelk = log(prior_k);

  for (gg in 1:ngames) {
    real p1 = P1[gg];
    real p2 = P2[gg];
    int lb1 = LB1[gg];
    int ub1 = UB1[gg];
    int lb2 = LB2[gg];
    int ub2 = UB2[gg];

    int n1 = ub1-lb1+1;
    int n2 = ub2-lb2+1;

    // choice sets for Player 1
```

```
    vector[n1] X1 = linspaced_vector(n1,lb1,ub1);

    for (kk in 1:kmax) {

      row_vector[n2] x2;

      // action of Player 2
      if (kk==1) {
        x2 = linspaced_vector(n2,lb2,ub2)';
      } else {
        x2 = rep_row_vector(LKpredictions2[gg][kk-1],n2);
      }

      vector[n1] payoff1 = payofffun(X1,x2,p1);

      if (choice1[gg]==LKpredictions1[gg][kk]) {
      // If the choice matches exactly with the prediction, then ...
      like_levelk[kk] += log(
         probExact
          +(1.0-probExact)*softmax(lambda*payoff1)[choice1_i[gg]]
        );

      } else {
        // if the choice does not match exactly with the prediction, then ...
        like_levelk[kk] += log(
         (1.0-probExact)*softmax(lambda*payoff1)[choice1_i[gg]]
        );
      }

    }

  }

}

model {


  // likelihood

  target += log_sum_exp(like_levelk);


  // priors
  target += beta_lpdf(probExact | prior_probExact[1],prior_probExact[2]);
  target += lognormal_lpdf(lambda | prior_lambda[1],prior_lambda[2]);


}

generated quantities {

  vector[kmax] postprob = softmax(like_levelk);
```

```
  real postk = postprob'*linspaced_vector(kmax,1,kmax);

}
```

17.4.2 Results

```
BMA<-"Code/CGC2006/EstimatesLevelkBMA.rds" |>
  readRDS() |>
  filter(
    parameter=="probExact"
    | parameter == "lambda"
    | parameter == "postk"
  ) |>
  pivot_wider(
    id_cols=id,
    names_from = parameter,
    values_from = mean
  )

(
  ggplot(BMA,aes(x=postk,y=probExact,color = log(lambda)))
  +geom_point()
    +labs(color=expression("log("~lambda~")"))
  +theme_bw()
  +scale_color_viridis()
  +xlab("k")
  +ylab(expression(rho))
)
```

17.5 A mixture model

A natural extension of the participant-specific models estimated in the previous section is to use data from all participants to estimate the fraction of participants who behave according to each type. That is, the ψ_ks in the previous section are now *parameters*, rather than a *prior*, and we instead assign a prior to these new parameters. The nice product we get from this is that instead of using our flat (discrete uniform) prior over k to get our estimate of k for each participant, we now are using all of the data to get a better idea of what the fraction of types actually are in the population. Hence, if there are a lot of level-1 types in the data (which it turns out there are), then this will be reflected in our estimates of ψ, and then be carried forward to our estimate of $E[k_i \mid y_i]$.

17.5.1 *Stan* program

Extending the one-participant program to this one was not too difficult, but it certainly helped to have started there! The trick here was to keep track of the individual likelihoods "`like_choices`" in the `transformed data` block so that I could use them in the `model` block for incrementing the likelihood, and the `generated quantities` block for the posterior type probabilities.

```
functions {

  vector payofffun(vector x1,row_vector x2,real p1) {

    /*
    x1 is the action space of the first player
```

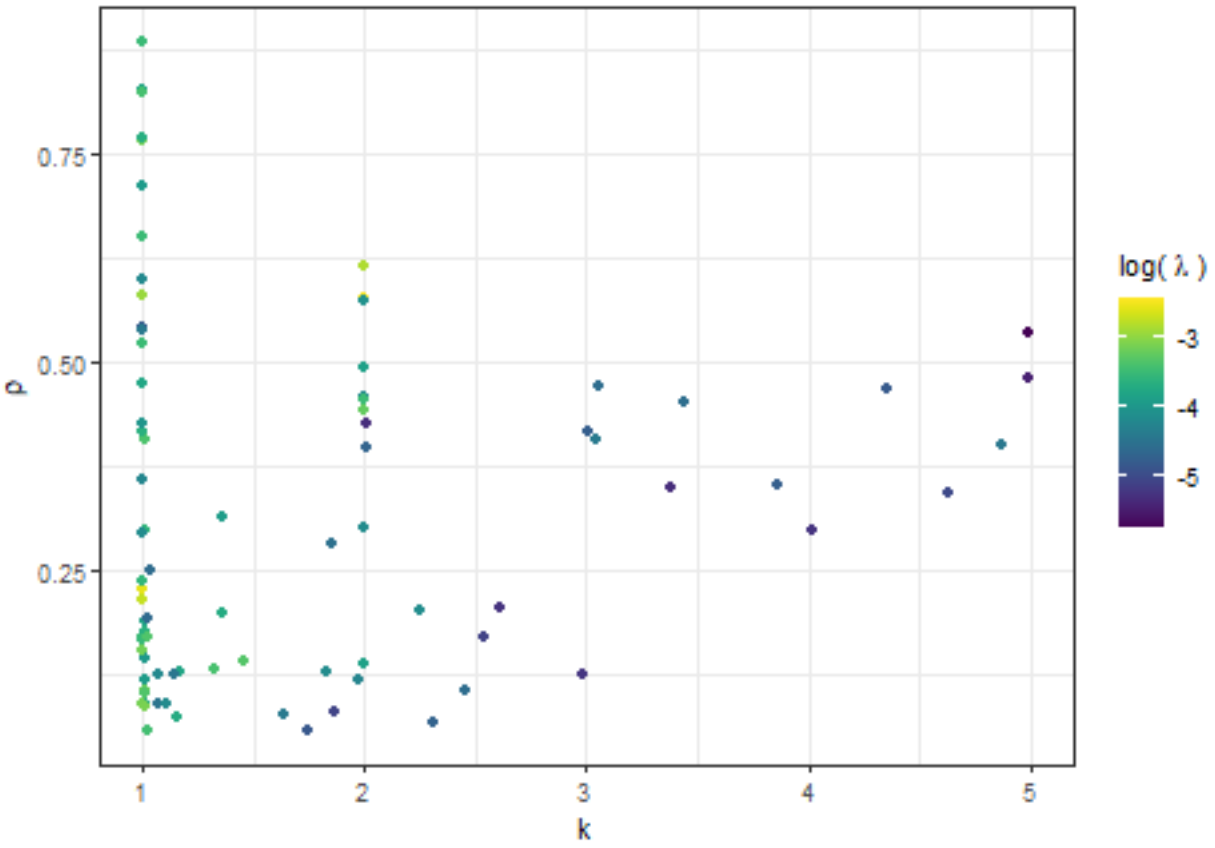


Figure 51: Individual-level estimates from the model assuming a discrete uniform prior over k.

```
    x2 is the distribution of actions of the second player. That is, for level 0
    player 2, it is the action space (which gets averaged over a uniform
    distribution), and for level k>0 is just populated by the prediction of this
    type
    */

    int n1 = size(x1);
    int n2 = size(x2);

    matrix[n1,n2] err = fabs(rep_matrix(x1,n2)-p1*rep_matrix(x2,n1));

    vector[n1] pay = (fmax(0.0,200-err*10)+fmax(0.0,100-err))*rep_vector(1.0/n1,n2);

    return pay;


  }

  int which_max(vector x,real tol) {

    int n = size(x);

    int ind = 0;

    for (ii in 1:n) {
      if (abs(x[ii]-max(x))<tol) {
        ind = ii;
      }
    }

    return ind;




  }


}

data {

  int ngames; // number of games
  int nparticipants; // number of participants

  int kmax; // max level k to evaluate

  // game data
  vector[ngames] P1;
  vector[ngames] P2;
  int LB1[ngames];
  int UB1[ngames];
  int LB2[ngames];
```

```
  int UB2[ngames];

  // player 1 choice
  int choice1[nparticipants,ngames];

  vector[2] prior_probExact;
  vector[2] prior_lambda;

  vector[kmax] prior_mix;

  // tolerance for identifying a maximum in function which_max(,)
  real<lower=0> tol;



}

transformed data {

  int choice1_i[nparticipants,ngames];

  // pre-compute level-k predictions

  vector[kmax] LKpredictions1[ngames];
  vector[kmax] LKpredictions2[ngames];




  for (gg in 1:ngames) {
    real p1 = P1[gg];
    real p2 = P2[gg];
    int lb1 = LB1[gg];
    int ub1 = UB1[gg];
    int lb2 = LB2[gg];
    int ub2 = UB2[gg];

    int n1 = ub1-lb1+1;
    int n2 = ub2-lb2+1;

    for (ii in 1:nparticipants) {
      choice1_i[ii,gg] = choice1[ii,gg]-lb1+1;
    }
    // choice sets
    vector[n1] X1 = linspaced_vector(n1,lb1,ub1);
    row_vector[n2] X2 = linspaced_row_vector(n2,lb2,ub2);

    // choices to get uniform distribution over actions
    vector[n1] x1 = linspaced_vector(n1,lb1,ub1);
    row_vector[n2] x2 = linspaced_row_vector(n2,lb2,ub2);


    for (kk in 1:kmax) {
```

```
      vector[n1] payoff1 = payofffun(X1,x2,p1);
      vector[n2] payoff2 = payofffun(X2',x1',p2);

      x1 = rep_vector(X1[which_max(payoff1,tol)],n1);
      x2 = rep_row_vector(X2[which_max(payoff2,tol)],n2);

      LKpredictions1[gg][kk] = X1[which_max(payoff1,tol)];
      LKpredictions2[gg][kk] = X2[which_max(payoff2,tol)];
    }

  }

}



parameters {

  simplex[kmax] mix;

  real<lower=0,upper=1> probExact;
  real<lower=0> lambda;


}

transformed parameters {

  matrix[nparticipants,kmax] like_choices = rep_matrix(log(mix'),nparticipants);

  for (gg in 1:ngames) {
    real p1 = P1[gg];
    real p2 = P2[gg];
    int lb1 = LB1[gg];
    int ub1 = UB1[gg];
    int lb2 = LB2[gg];
    int ub2 = UB2[gg];

    int n1 = ub1-lb1+1;
    int n2 = ub2-lb2+1;

    // choice sets for Player 1
    vector[n1] X1 = linspaced_vector(n1,lb1,ub1);

    for (kk in 1:kmax) {
      row_vector[n2] x2;

      // action of Player 2
      if (kk==1) {
        x2 = linspaced_vector(n2,lb2,ub2)';
      } else {
        x2 = rep_row_vector(LKpredictions2[gg][kk-1],n2);
      }
```

```
      vector[n1] payoff1 = payofffun(X1,x2,p1);

      for (ii in 1:nparticipants) {
        if (choice1[ii,gg]==LKpredictions1[gg][kk]) {
         // If the choice matches exactly with the prediction, then ...
           like_choices[ii,kk] += log(
           probExact
              +(1.0-probExact)*softmax(lambda*payoff1)[choice1_i[ii,gg]]
            );

          } else {
      // if the choice does not match exactly with the prediction, then ...
           like_choices[ii,kk] += log(
            (1.0-probExact)*softmax(lambda*payoff1)[choice1_i[ii,gg]]
            );
          }
      } // loop over participants ends here

    } // loop over k ends here

  } // loop over games ends here


}

model {


  // priors ------------------------------------------------------------------
  mix ~ dirichlet(prior_mix);
  probExact ~ beta(prior_probExact[1],prior_probExact[2]);
  lambda ~ lognormal(prior_lambda[1],prior_lambda[2]);



  for (ii in 1:nparticipants) {
    target += log_sum_exp(like_choices[ii,]);
  }


}

generated quantities {

  matrix[nparticipants,kmax] postprob;
  vector[nparticipants] postk;

  for (ii in 1:nparticipants) {
    postprob[ii,] = softmax(like_choices[ii,]')';
    postk[ii] = postprob[ii,]*linspaced_vector(kmax,1,kmax);

  }
```

Table 39: Estimates from the mixture model

	mean	se_mean	sd	X2.5.	X25.	X50.	X75.	X97.5.
mix[1]	0.574	0.001	0.055	0.465	0.537	0.575	0.612	0.680
mix[2]	0.248	0.001	0.050	0.158	0.212	0.246	0.281	0.352
mix[3]	0.065	0.000	0.028	0.021	0.045	0.062	0.081	0.128
mix[4]	0.040	0.000	0.025	0.005	0.022	0.036	0.055	0.101
mix[5]	0.073	0.000	0.031	0.023	0.050	0.069	0.092	0.145
probExact	0.331	0.000	0.013	0.306	0.323	0.331	0.340	0.357
lambda	0.007	0.000	0.001	0.006	0.007	0.007	0.008	0.009

```
}
```

17.5.2 A prior for ψ

The parameters λ and ρ have the same interpretation as the previous models, so I keep their priors the same for the mixture model. The new parameter his is ψ, the vector of mixing probabilities. For this, I assume:

$$\psi \sim \text{Dirichlet}(1, 1, 1, 1, 1)$$

Which means that the prior mean for each mixing probability is $\frac{1}{5}$, and the marginal distribution of each mixing probability is:

$$\psi_k \sim \text{Beta}(1, 4) \quad \forall k$$

17.5.3 Results

```
Mix<-summary("Code/CGC2006/EstimatesLevelkMixture.rds" |>
  readRDS()
)$summary |>
  data.frame()


Mix[c("mix[1]","mix[2]","mix[3]","mix[4]","mix[5]","probExact","lambda"),] |>
  dplyr::select(-n_eff,-Rhat) |>
  round(3) |>
  kbl(caption = "Estimates from the mixture model") |>
  kable_classic(full_width=FALSE)
```

Table 39 shows a summary of the posterior distribution of the mixture model. Here we can see that most participants are level-1, with the next most type being level 2. There seems to be a substantial spike in the probabilistic best response function, with ρ estimated to be about 0.33 (0.01).

17.6 A mixture over levels and hierarchical nuisance parameters

Up to this point, we have either estimated one model for each participant, or assumed that the parameters ρ and λ were constant for all participants. We can relax this with a hierarchical specification. That is, here we will preserve the mixture part of the mixture model discussed in the previous section, but add in a hierarchical specification for these parameters.

Before moving to the model specification, which at this point will not be too dissimilar from other hierarchical specifications (just with a different individual-level likelihood function), it is useful to think about *why* we might want to go down this route of taking participant-level heterogeneity seriously on two levels. That is, we will be assuming that participants are heterogeneous because:

1. They have different levels of reasoning. That is, they have different ks, and
2. They have different parameters in the probabilistic choice rule: ρ_i and λ_i.

On the one hand, we might be interested in all of these parameters and types, and how they are different between participants. Estimating a model that allows them to be different is therefore critical for being able to comment on these things. On the other hand, and perhaps more common in the literature on level-k reasoning, we might just be interested in commenting on the distribution of k in our population, and don't really give a hoot about ρ_i and λ_i. In this second case, ρ_i and λ_i are *nuisance parameters*: our goal is to estimate something else, but we have to jointly estimate them in order to do this. If possible, we still want to take participant-level heterogeneity seriously here, because models that assume it away can produce misleading results (see Wilcox (2006) for a good example of this). So even though we might not want to use our estimates of ρ_i and λ_i to answer our research question, we want to worry about heterogeneity in these parameters.

In order to ensure the parameter restrictions for the individual-level parameters $\lambda_i > 0$ and $\rho_i \in (0, 1)$, I used the following transformed normal hierarchical specification:

$$\theta_i \equiv \begin{pmatrix} \Phi^{-1}(\rho_i) \\ \log \lambda_i \end{pmatrix} \sim N(\mu, \text{diag\_matrix}(\tau)\Omega\text{diag\_matrix}(\tau))$$

17.6.1 Prior calibration

Here we need to specify priors over the population-level parameters μ, τ, and Ω. I ended up using:

$$\begin{aligned} \mu_\rho &\sim N(0, 0.25^2) \\ \mu_\lambda &\sim N(-1.96, 1.35^2) \\ \tau_\rho, \tau_\lambda &\sim \text{Cauchy}^+(0, 0.05) \\ \Omega &\sim \text{LKJ}(4) \end{aligned}$$

For μ_λ, I am just using the prior I calibrated for λ for the participant-specific models. If anything, this is probably a bit spread out, as it seems reasonable that μ_λ, which pins down the median λ_i, should be less spread out that the distribution of λ_i itself. For μ_ρ, we now have a probit-normal marginal distribution for ρ_i, so we can't stick with the Beta prior we had in the participant-specific models. Setting the prior mean of this parameter to 0 centers the prior median ρ_i on 50%, and a standard deviation of 0.25 means that the prior distribution is single-peaked (at $\rho_i = 50\%$) and covers the unit interval fairly well.

17.6.2 *Stan* program

Here is my implementation of the hierarchical model in *Stan*. This took about 9 hours to run with 10,000 iterations for each of the four chains (run in parallel).[76] This came as quite a relief to me, as the first Bayesian model averaging approach took over a day on the same machine!

```
functions {

  vector payofffun(vector x1,row_vector x2,real p1) {

    /*
    x1 is the action space of the first player
```

[76] My initial run using *RStan*'s default 2,000 iterations per chain yielded some ESS warnings, which were easily fixed by increasing the number of iterations.

```
    x2 is the distribution of actions of the second player. That is, for level 0
    player 2, it is the action space (which gets averaged over a uniform
    distribution), and for level k>0 is just populated by the prediction of this
    type
    */

    int n1 = size(x1);
    int n2 = size(x2);

    matrix[n1,n2] err = fabs(rep_matrix(x1,n2)-p1*rep_matrix(x2,n1));

    vector[n1] pay = (fmax(0.0,200-err*10)+fmax(0.0,100-err))*rep_vector(1.0/n1,n2);

    return pay;


  }

  int which_max(vector x,real tol) {

    int n = size(x);

    int ind = 0;

    for (ii in 1:n) {
      if (abs(x[ii]-max(x))<tol) {
        ind = ii;
      }
    }

    return ind;




  }


}

data {

  int ngames; // number of games
  int nparticipants; // number of participants

  int kmax; // max level k to evaluate

  // game data
  vector[ngames] P1;
  vector[ngames] P2;
  int LB1[ngames];
  int UB1[ngames];
  int LB2[ngames];
```

```
  int UB2[ngames];

  // player 1 choice
  int choice1[nparticipants,ngames];

  vector[2] prior_MU[2];
  vector[2] prior_TAU;
  real prior_OMEGA;

  vector[kmax] prior_mix;

  // tolerance for identifying a maximum in function which_max(,)
  real<lower=0> tol;



}

transformed data {

  int choice1_i[nparticipants,ngames];

  // pre-compute level-k predictions

  vector[kmax] LKpredictions1[ngames];
  vector[kmax] LKpredictions2[ngames];




  for (gg in 1:ngames) {
    real p1 = P1[gg];
    real p2 = P2[gg];
    int lb1 = LB1[gg];
    int ub1 = UB1[gg];
    int lb2 = LB2[gg];
    int ub2 = UB2[gg];

    int n1 = ub1-lb1+1;
    int n2 = ub2-lb2+1;

    for (ii in 1:nparticipants) {
      choice1_i[ii,gg] = choice1[ii,gg]-lb1+1;
    }
    // choice sets
    vector[n1] X1 = linspaced_vector(n1,lb1,ub1);
    row_vector[n2] X2 = linspaced_row_vector(n2,lb2,ub2);

    // choices to get uniform distribution over actions
    vector[n1] x1 = linspaced_vector(n1,lb1,ub1);
    row_vector[n2] x2 = linspaced_row_vector(n2,lb2,ub2);
```

```
    for (kk in 1:kmax) {

      vector[n1] payoff1 = payofffun(X1,x2,p1);
      vector[n2] payoff2 = payofffun(X2',x1',p2);

      x1 = rep_vector(X1[which_max(payoff1,tol)],n1);
      x2 = rep_row_vector(X2[which_max(payoff2,tol)],n2);

      LKpredictions1[gg][kk] = X1[which_max(payoff1,tol)];
      LKpredictions2[gg][kk] = X2[which_max(payoff2,tol)];
    }

  }

}



parameters {

  simplex[kmax] mix;

  vector[2] MU;
  vector<lower=0>[2] TAU;
  cholesky_factor_corr[2] L_OMEGA;

  matrix[2,nparticipants] z;

}

transformed parameters {

  vector[nparticipants] probExact;
  vector[nparticipants] lambda;

  {

    matrix[2,nparticipants] theta = rep_matrix(MU,nparticipants)
      + diag_pre_multiply(TAU,L_OMEGA)*z;

    probExact = Phi_approx(theta[1,]');
    lambda = exp(theta[2,]');

  }


  matrix[nparticipants,kmax] like_choices = rep_matrix(log(mix'),nparticipants);

  for (gg in 1:ngames) {
    real p1 = P1[gg];
    real p2 = P2[gg];
    int lb1 = LB1[gg];
    int ub1 = UB1[gg];
```

```
    int lb2 = LB2[gg];
    int ub2 = UB2[gg];

    int n1 = ub1-lb1+1;
    int n2 = ub2-lb2+1;

    // choice sets for Player 1
    vector[n1] X1 = linspaced_vector(n1,lb1,ub1);

    for (kk in 1:kmax) {
      row_vector[n2] x2;

      // action of Player 2
      if (kk==1) {
        x2 = linspaced_vector(n2,lb2,ub2)';
      } else {
        x2 = rep_row_vector(LKpredictions2[gg][kk-1],n2);
      }

      vector[n1] payoff1 = payofffun(X1,x2,p1);

      for (ii in 1:nparticipants) {
        if (choice1[ii,gg]==LKpredictions1[gg][kk]) {
         // If the choice matches exactly with the prediction, then ...
           like_choices[ii,kk] += log(
           probExact[ii]
              +(1.0-probExact[ii])*softmax(lambda[ii]*payoff1)[choice1_i[ii,gg]]
            );

          } else {
      // if the choice does not match exactly with the prediction, then ...
           like_choices[ii,kk] += log(
            (1.0-probExact[ii])*softmax(lambda[ii]*payoff1)[choice1_i[ii,gg]]
            );
          }
      } // loop over participants ends here

    } // loop over k ends here

  } // loop over games ends here


}

model {


  // priors ------------------------------------------------------------------
  mix ~ dirichlet(prior_mix);
  for (pp in 1:2) {
    MU[pp] ~ normal(prior_MU[pp][1],prior_MU[pp][2]);
    TAU[pp] ~ cauchy(0.0,prior_TAU[pp]);
  }
```

```
  L_OMEGA ~ lkj_corr_cholesky(prior_OMEGA);

  // hierarchical structure (standardized) -------------------------------------
  to_vector(z) ~ std_normal();

  for (ii in 1:nparticipants) {
    target += log_sum_exp(like_choices[ii,]);
  }


}

generated quantities {

  matrix[2,2] OMEGA = L_OMEGA*L_OMEGA';

  matrix[nparticipants,kmax] postprob;
  vector[nparticipants] postk;

  for (ii in 1:nparticipants) {
    postprob[ii,] = softmax(like_choices[ii,]')';
    postk[ii] = postprob[ii,]*linspaced_vector(kmax,1,kmax);

  }

}
```

17.6.3 Results

To begin with, here are the estimates of the mixing probabilities, shown in Table 40. These are not too different from the mixture model assuming homogeneous λ and ρ, but we didn't know that before we estimated this model.

```
Hierarchical<- summary("Code/CGC2006/EstimatesLevelkHierarchicalMixture.rds" |>
  readRDS())$summary |>
  data.frame()



TAB<-Hierarchical[grepl("mix",rownames(Hierarchical)),]

TAB |>
  dplyr::select(-n_eff,-Rhat) |>
  round(3) |>
  kbl(caption = "Mixing probability estimates from the hierarchical/mixture level-$k$ model. ") |>
  kable_classic(full_width=FALSE)
```

Finally, we can look at the posterior means of the individual-level parameters in Figure 52. These are not too different from our participant-specific estimates (Figure 50).

```
Hierarchical$parameter<-gsub('[[:digit:]]+', '',rownames(Hierarchical))

d<-Hierarchical[grepl("postk",rownames(Hierarchical))
                |  grepl("probExact",rownames(Hierarchical))
                | grepl("lambda",rownames(Hierarchical))
```

Table 40: Mixing probability estimates from the hierarchical/mixture level-k model.

	mean	se_mean	sd	X2.5.	X25.	X50.	X75.	X97.5.
mix[1]	0.604	0	0.056	0.493	0.567	0.605	0.643	0.711
mix[2]	0.221	0	0.048	0.134	0.189	0.219	0.252	0.321
mix[3]	0.069	0	0.029	0.023	0.047	0.066	0.087	0.134
mix[4]	0.033	0	0.023	0.002	0.016	0.028	0.045	0.089
mix[5]	0.073	0	0.030	0.024	0.051	0.069	0.091	0.142

```
                ,]

d$id<-parse_number(rownames(d))

d<-d |>
  pivot_wider(id_cols = id,
              names_from = "parameter",
              values_from = "mean")

(
  ggplot(d,aes(x=`postk[]`,y=`probExact[]`,color = log(`lambda[]`)))
  +geom_point()
    +labs(color=expression("log("~lambda~")"))
  +theme_bw()
  +scale_color_viridis()
  +xlab("k")
  +ylab(expression(rho))
)
```

17.7 A different assumption about mixing

Up to this point, we maintained the simplifying assumption that a level-k type believes that they are playing against the level-$(k-1)$ prediction *without any probabilistic choice*. This allowed us to pre-compute the model's deterministic predictions in the `transformed data` block. Here we will instead assume something much closer to the model proposed by Stahl and Wilson (1994). Specifically, we will assume that the level-k type probabilistically best responds to the aggregate mixed strategy played by all types with lower levels of strategic reasoning. That is, if each level-k type plays mixed strategy σ_k, then this aggregate strategy that the level-k type believes is being played is:

$$\tilde{\sigma}_{k-1} = \frac{\sum_{j=1}^{k-1} \psi_j \sigma_j}{\sum_{j=1}^{k-1} \psi_j}$$

That is, a level-k player's beliefs are consistent with the true distribution of types ψ, if it is truncated to only include types with lower k than the player's.

To keep things simple here, I will drop the "spike" part of the spike-logit probabilistic choice rule (i.e. $\rho = 0$), and assume that all participants have the same choice precision λ. This new model is therefore most closely related to the mixture model *without* the hierarchical specification for λ_i and ρ_i.[77]

[77] Attempts to incorporate a hierarchical specification for λ_i were made, however at the time of writing there has been no joy. The added complication is that we would need to integrate out λ_i to compute the aggregate mixed strategy.

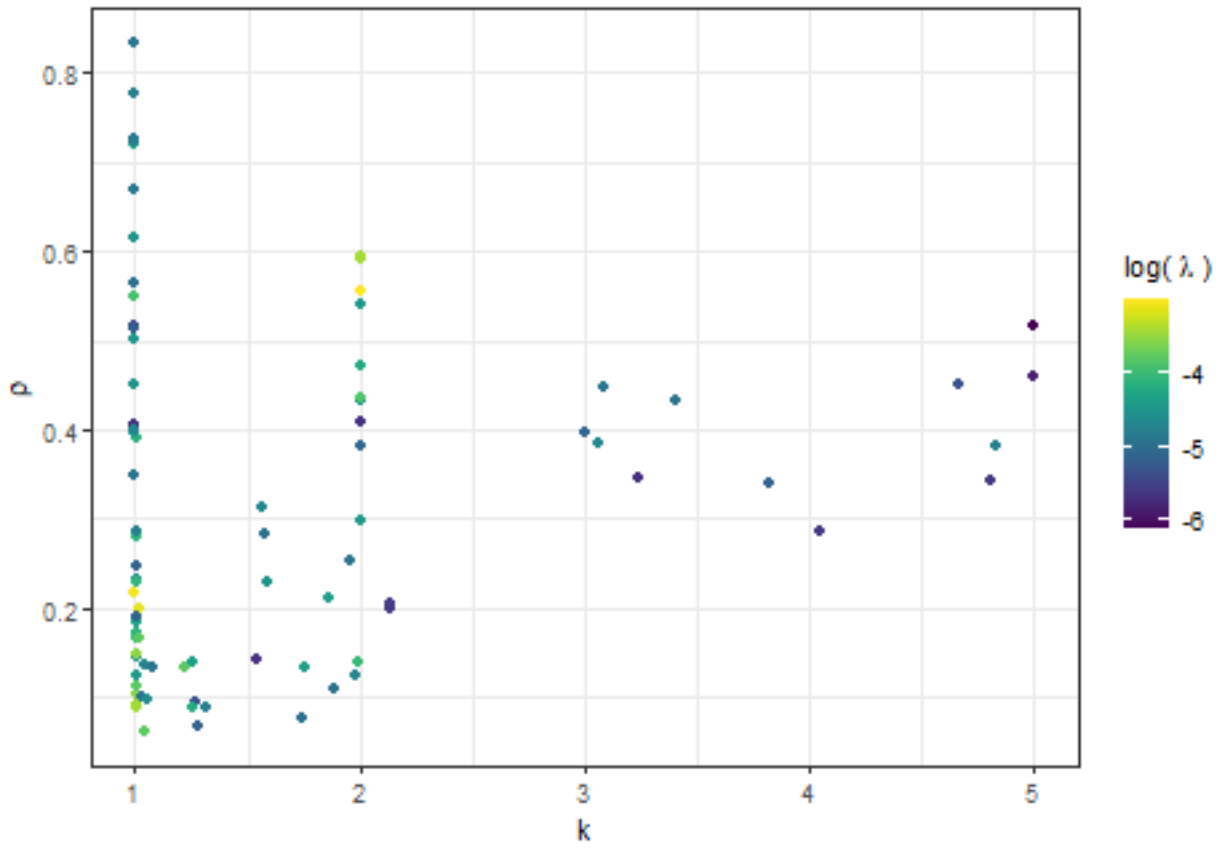


Figure 52: Shrinkage estimates of individual-level parameters from the heirarchical/mixture level-k model.

17.7.1 *Stan* program

Here is the *Stan* file that implements this model. Note now that we have to compute the predictions in the `transformed parameters` block, because they are a function of λ and ψ.

```
functions {

  vector logitresponse(real lambda, vector x1,row_vector x2,vector prob2,real p1) {

    /*
    x1 is the action space of the first player

    x2 is the action space of the second player

    prob2 is the mixed strategy of player 2
    */

    int n1 = size(x1);
    int n2 = size(x2);

    matrix[n1,n2] err = fabs(rep_matrix(x1,n2)-p1*rep_matrix(x2,n1));

    vector[n1] pay = lambda*(fmax(0.0,200-err*10)+fmax(0.0,100-err))*prob2;



    return softmax(pay);


  }

}

data {

  int ngames; // number of games
  int nparticipants; // number of participants

  int kmax; // max level k to evaluate

  // game data
  vector[ngames] P1;
  vector[ngames] P2;
  int LB1[ngames];
  int UB1[ngames];
  int LB2[ngames];
  int UB2[ngames];

  // player 1 choice
  int choice1[nparticipants,ngames];

  vector[2] prior_lambda;

  vector[kmax] prior_mix;
```

```
  // tolerance for identifying a maximum in function which_max(,)
  real<lower=0> tol;



}

transformed data {

  int choice1_i[nparticipants,ngames];




  for (gg in 1:ngames) {
    int lb1 = LB1[gg];

    for (ii in 1:nparticipants) {
      choice1_i[ii,gg] = choice1[ii,gg]-lb1+1;
    }


  }

}



parameters {

  simplex[kmax] mix;

  real<lower=0> lambda;


}

transformed parameters {

  matrix[nparticipants,kmax] like_choices = rep_matrix(log(mix'),nparticipants);

  for (gg in 1:ngames) {
    real p1 = P1[gg];
    real p2 = P2[gg];
    int lb1 = LB1[gg];
    int ub1 = UB1[gg];
    int lb2 = LB2[gg];
    int ub2 = UB2[gg];

    int n1 = ub1-lb1+1;
    int n2 = ub2-lb2+1;

    // choice set for Player 1
```

```
    vector[n1] X1 = linspaced_vector(n1,lb1,ub1);
    // choice set for Player 2
    vector[n2] X2 = linspaced_vector(n2,lb2,ub2);
    // level-0 strategies
    vector[n1] prob1 = rep_vector(1.0/n1,n1);
    vector[n2] prob2 = rep_vector(1.0/n2,n2);

    // aggregate strategies
    vector[n1] sigma1=prob1;
    vector[n2] sigma2=prob2;

    for (kk in 1:kmax) {

      // compute logit response to mixed strategy
      prob1 = logitresponse(lambda, X1,X2',sigma2,p1);
      prob2 = logitresponse(lambda, X2,X1',sigma1,p2);

      // add to likelihood
      like_choices[,kk] += log(prob1)[choice1_i[,gg]];

      // update mixed strategy
      if (kk == 1) {
        // aggregate strategy is the level-1 strategy
        sigma1 = prob1;
        sigma2 = prob2;
      } else {
        // aggregate strategy is a convex combination of what we had before
        // and the logit response

        sigma1 = (mix[kk]*prob1 + sum(mix[1:(kk-1)])*sigma1)/sum(mix[1:kk]);
        sigma2 = (mix[kk]*prob2 + sum(mix[1:(kk-1)])*sigma2)/sum(mix[1:kk]);


      }

    }




  } // loop over games ends here


}

model {


  // priors ------------------------------------------------------------------
  target += dirichlet_lpdf(mix | prior_mix);
  target += lognormal_lpdf(lambda | prior_lambda[1],prior_lambda[2]);
```

```
  for (ii in 1:nparticipants) {
    target += log_sum_exp(like_choices[ii,]);
  }


}



generated quantities {

  matrix[nparticipants,kmax] postprob;
  vector[nparticipants] postk;

  for (ii in 1:nparticipants) {
    postprob[ii,] = softmax(like_choices[ii,]')';
    postk[ii] = postprob[ii,]*linspaced_vector(kmax,1,kmax);

  }



}
```

17.7.2 Results

Estimates from this model are show in Table 41. Probably the most similar model for comparison is to Table 39. Here we have a slightly smaller fraction of level-1 types, and a more uniform distribution over the remaining types.

```
Eq<-summary("Code/CGC2006/EstimatesLevelkEq.rds" |>
  readRDS())$summary |>
  data.frame()

Eq |>
  filter(
    grepl("mix",rownames(Eq))
    | grepl("lambda",rownames(Eq))
  ) |>
  dplyr::select(-n_eff,-Rhat) |>
  round(3)|>
  kbl(caption = "Estimates from the mixture model assuming beliefs consistent with the truncated distril
  kable_classic(full_width=FALSE)
```

17.8 *R* code to estimate the models

17.8.1 Participant-specific estimation conditional on k with Bayesian model averaging

```
library(tidyverse)
library(rstan)
  rstan_options(auto_write = TRUE)
  options(mc.cores = parallel::detectCores())
library(bridgesampling)
```

Table 41: Estimates from the mixture model assuming beliefs consistent with the truncated distribution of level-k types.

	mean	se_mean	sd	X2.5.	X25.	X50.	X75.	X97.5.
mix[1]	0.473	0.001	0.067	0.342	0.428	0.473	0.518	0.604
mix[2]	0.145	0.002	0.117	0.004	0.047	0.114	0.221	0.412
mix[3]	0.133	0.002	0.110	0.004	0.044	0.105	0.198	0.395
mix[4]	0.123	0.002	0.102	0.004	0.043	0.097	0.178	0.376
mix[5]	0.125	0.002	0.104	0.004	0.041	0.098	0.182	0.384
lambda	0.017	0.000	0.001	0.015	0.016	0.017	0.017	0.018

```
kmax<-5

model<-"Code/CGC2006/LevelkRepAgent.stan" |>
  stan_model()

D<-"Code/CGC2006/CGC2006rounded.rds" |>
  readRDS() |>
  mutate(
    # convert choices to adjusted choices
    choice = ifelse(choice<lb1,lb1,choice),
    choice = ifelse(choice>ub1,ub1,choice)
  )

ESTIMATES<-tibble()

for (ii in unique(D$id)) {

  d<- D |> filter(id==ii)

  for (kk in 1:kmax) {

    print(paste("estimating for id",ii,"k =",kk))

    dStan<-list(

      ngames = 16,

      k = kk,

      P1 = d$p_1,
      P2 = d$p_2,
      LB1 = d$lb1,
      UB1 = d$ub1,
      LB2 = d$lb2,
      UB2 = d$ub2,

      choice1 = d$choice,

      prior_probExact = c(1,1),
      prior_lambda = c(-1.96,1.35),
```

```
      tol = 0.001


    )

    Fit<-model |>
      sampling(
        data=dStan,
        seed=42
      )

    lml<-bridge_sampler(Fit,silent=TRUE)

    ESTIMATES <- ESTIMATES |>
      rbind(
        summary(Fit)$summary |>
          data.frame() |>
          mutate(id = ii,
                 k = kk,
                 logml = lml$logml
                 )
      )
  }

  ESTIMATES |>
    saveRDS("Code/CGC2006/EstimatesLevelkRepAgent.rds")
}
```

17.8.2 Participant-specific estimation with a prior over k

```
library(tidyverse)
library(rstan)
  rstan_options(auto_write = TRUE)
  options(mc.cores = parallel::detectCores())


model<-"Code/CGC2006/LevelkBMA.stan" |>
  stan_model()

D<-"Code/CGC2006/CGC2006rounded.rds" |>
  readRDS() |>
  mutate(
    # convert choices to adjusted choices
    choice = ifelse(choice<lb1,lb1,choice),
    choice = ifelse(choice>ub1,ub1,choice)
  )




ESTIMATES<-tibble()

for (ii in unique(D$id)) {
```

```
  d<- D |> filter(id==ii)

    print(paste("estimating for id",ii))

    dStan<-list(

      ngames = 16,

      kmax = 5,

      P1 = d$p_1,
      P2 = d$p_2,
      LB1 = d$lb1,
      UB1 = d$ub1,
      LB2 = d$lb2,
      UB2 = d$ub2,

      choice1 = d$choice,

      prior_probExact = c(1,1),
      prior_lambda = c(-1.96,1.35),
      prior_k = rep(1/5,5),

      tol = 0.001


    )

    Fit<-model |>
      sampling(
        data=dStan,
        seed=42
      )

    addThis<-summary(Fit)$summary |>
      data.frame() |>
      mutate(id = ii)

    addThis$parameter<-rownames(addThis)

    ESTIMATES <- ESTIMATES |>
      rbind(addThis)

  ESTIMATES |>
    saveRDS("Code/CGC2006/EstimatesLevelkBMA.rds")
}
```

17.8.3 Mixture model

```
library(tidyverse)
library(rstan)
  rstan_options(auto_write = TRUE)
  options(mc.cores = parallel::detectCores())
```

```
model<-"Code/CGC2006/LevelkMixture.stan" |>
  stan_model()




D<-"Code/CGC2006/CGC2006rounded.rds" |>
  readRDS() |>
  arrange(id,game) |>
  mutate(
    # convert choices to adjusted choices
    choice = ifelse(choice<lb1,lb1,choice),
    choice = ifelse(choice>ub1,ub1,choice)
  )

choices<-D |>
  pivot_wider(id_cols = "id",
              names_from = game,
              values_from = choice) |>
  select(-id) |>
  as.matrix()

d<-D |> filter(id==101)


dStan<-list(
  ngames = 16,
  nparticipants = dim(choices)[1],

  kmax = 5,
  P1 = d$p_1,
  P2 = d$p_2,
  LB1 = d$lb1,
  UB1 = d$ub1,
  LB2 = d$lb2,
  UB2 = d$ub2,

  choice1 = choices,

  prior_probExact = c(1,1),
  prior_lambda = c(-1.96,1.35),

  prior_mix = rep(1,5),

  tol = 0.001



)

Fit<-model |>
  sampling(
```

```
    data=dStan,
    pars = c("like_choices"),include=FALSE,
    iter=2000,
    seed=42
      )

Fit |>
  saveRDS(file="Code/CGC2006/EstimatesLevelkMixture.rds")
```

17.8.4 Hierarchical model

```
library(tidyverse)
library(rstan)
  rstan_options(auto_write = TRUE)
  options(mc.cores = parallel::detectCores())

model<-"Code/CGC2006/LevelkHierarchicalMixture.stan" |>
  stan_model()




D<-"Code/CGC2006/CGC2006rounded.rds" |>
  readRDS() |>
  arrange(id,game) |>
  mutate(
    # convert choices to adjusted choices
    choice = ifelse(choice<lb1,lb1,choice),
    choice = ifelse(choice>ub1,ub1,choice)
  )

choices<-D |>
  pivot_wider(id_cols = "id",
              names_from = game,
              values_from = choice) |>
  select(-id) |>
  as.matrix()

d<-D |> filter(id==101)


dStan<-list(
  ngames = 16,
  nparticipants = dim(choices)[1],

  kmax = 5,
  P1 = d$p_1,
  P2 = d$p_2,
  LB1 = d$lb1,
  UB1 = d$ub1,
  LB2 = d$lb2,
  UB2 = d$ub2,
```

```
  choice1 = choices,

  prior_MU = list(
    c(0,0.25),
    c(-1.96,1.35)
  ),
  prior_TAU = c(0.05,0.05),
  prior_OMEGA = 4,
  prior_mix = rep(1,5),

  tol = 0.001



)

Fit<-model |>
  sampling(
    data=dStan,
    pars = c("z","L_OMEGA","like_choices"),include=FALSE,
    # I got the bulk and tail ESS warnings when running this with iter = 2000
    iter=10000,
    seed=42
      )

Fit |>
  saveRDS(file="Code/CGC2006/EstimatesLevelkHierarchicalMixture.rds")
```

17.8.5 Mixture model with beliefs consistent with truncated type distribution

```
library(tidyverse)
library(rstan)
  rstan_options(auto_write = TRUE)
  options(mc.cores = parallel::detectCores())

model<-"Code/CGC2006/LevelkEq.stan" |>
  stan_model()




D<-"Code/CGC2006/CGC2006rounded.rds" |>
  readRDS() |>
  arrange(id,game) |>
  mutate(
    # convert choices to adjusted choices
    choice = ifelse(choice<lb1,lb1,choice),
    choice = ifelse(choice>ub1,ub1,choice)
  )

choices<-D |>
  pivot_wider(id_cols = "id",
```

```
              names_from = game,
              values_from = choice) |>
  select(-id) |>
  as.matrix()

d<-D |> filter(id==101)


dStan<-list(
  ngames = 16,
  nparticipants = dim(choices)[1],

  kmax = 5,
  P1 = d$p_1,
  P2 = d$p_2,
  LB1 = d$lb1,
  UB1 = d$ub1,
  LB2 = d$lb2,
  UB2 = d$ub2,

  choice1 = choices,

  prior_probExact = c(1,1),
  prior_lambda = c(-1.96,1.35),

  prior_mix = rep(1,5),

  tol = 0.001



)

Fit<-model |>
  sampling(
    data=dStan,
    pars = c("like_choices"),include=FALSE,
    iter=2000,
    seed=42
      )

Fit |>
  saveRDS(file="Code/CGC2006/EstimatesLevelkEq.rds")
```

18 Application: Estimating risk preferences

Perhaps the most common use of structural models in experimental economics is the estimation of risk preferences. I would therefore not be doing the field justice if I did not devote at least one chapter (and I suspect there will be more later) to this topic. In this chapter, I will show you how to estimate some of the popular specifications that are taken to data from the "battery" binary choice experiment design, where participants are presented with many choices between two lotteries. To my knowledge, this design was first used by Hey and Di Cagno (1990), and has been the workhorse providing us with data to estimate risk

preferences ever since.[78]

We can see individual-estimation of risk preferences using maximum likelihood estimation as early as Hey and Orme (1994), and the econometrics of this kind of experiment took a great leap forward as researchers started to meaningfully aggregate their participants' decisions using mixture models with random parameters specifications (Conte, Hey, and Moffatt 2011), or exploiting correlation between parameters and observed participant characteristics (Harrison and Rutström 2009). These aggregating models are particularly important for the battery design, because typical experiments are under-powered at distinguishing between alternative models (Monroe 2020), but nonetheless can provide us with a *lot* of information about the population distribution. Furthermore, drawing inference about the population may in fact be a more satisfying answer to our research questions than being able to comment on how many participants are classified into each model individually.

18.1 Example dataset

For this application, I will use data from Harrison and Swarthout (2023). Specifically, I will use the subset of this dataset corresponding to undergraduate participants who played in the "house money" (as opposed to "earned endowment") treatment. The 175 participants in this treatment each made 100 decisions over pairs of lotteries. These lotteries were over both gains and losses, however for simplicity here I add in the show-up fee and the pair's "endowment" to ensure that all lottery prizes are strictly positive. Because both lotteries in a pair share the same three prizes, we can represent each lottery pair in a probability triangle, which plots the probability of winning the lowest prize on the horizontal axis, and the probability of winning the highest prize on the vertical axis. This is shown in Figure 53.

```
d<-"Code/RiskPreferences/HS2023Cleaned.rds" |>
  readRDS() |>
  filter(id==1) |>
  mutate(context = paste0("$",prize1," $",prize2," $",prize3))

mmTri<-tibble(
  x    = c(0,0,1),
  y    = c(0,0,0),
  xend = c(1,0,0),
  yend = c(0,1,1)
)

(
  ggplot(d,aes(x=qL1,y=qL3,xend=qR1,yend=qR3,color=frame))
  +geom_segment()
  +facet_wrap(~context)
  +geom_segment(data=mmTri,aes(x=x,y=y,xend=xend,yend=yend),color="black")
  +theme_bw()
  +coord_fixed()
  +xlab("probability of winning smallest prize")
  +ylab("probability of winning largest prize")
)
```

In Figure 53, we can see a lot of the popular features of this kind of experiment design. In particular if we just focus on the "$10 $45 $80" panel of this Figure, we can see that there is a wide range of slopes of the connecting line segments. If we were solely in the realm of Expected Utility Theory (EUT), this slope determines which of the two lotteries a participant will choose (if they choose without noise). Varying the slope allows participants with different amounts of risk-aversion to reveal their preferences to us. But there are also parallel line segments in this triangle. If participants choose according to EUT, then they should

[78] Glenn Harrison's replication archive is an amazing resource if you are looking for datasets using this design. I am indebted to the data sharing norms in Experimental Economics that mean I am able to have easy access to resources like this to do my work.

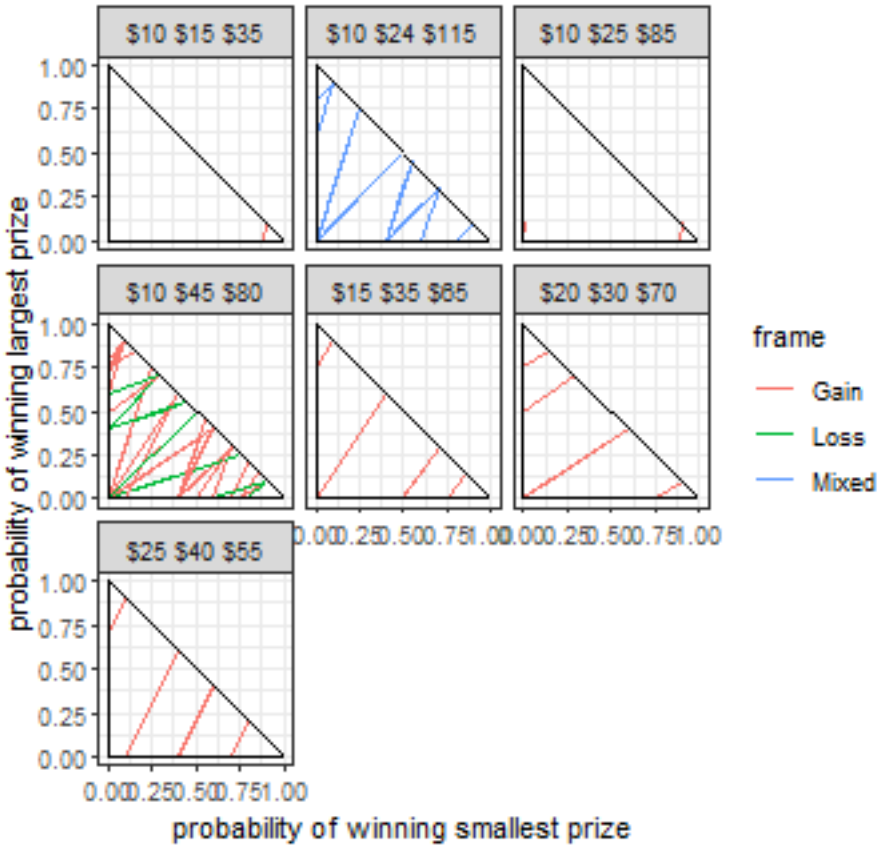


Figure 53: The Harrison and Swarthout (2023) experiment represented in the probability triangle. Prizes for each lottery pair, inclusive of the endowment and show-up fee, are shown at the top of each panel. Some lottery pairs are obscured because their connecting line segments overlap with a segment for another pair.

choose the lotteries at *the same end* of these parallel line segments.[79] Hence, these parallel line segments provide participants with an opportunity to reveal to us that they are *not* making decisions according to EUT. Here we can also see that there are seven different "contexts" of lotteries presented. That is, the dollar amounts of the prizes are varied. This is a somewhat *less* common property of this class of experiment design,[80] but this kind of variation is also useful because it allows us to observe behavior over a wider range of payoffs, possibly getting us a better estimate of the utility function over money.

18.2 We might not just be interested in the parameters

Depending on what we want to get out of our model, the model's parameters may not be the most interesting things to us. This is because economically meaningful quantities are sometimes *transformations* of a model's fundamental parameters. The good news for us is that once we have a posterior simulation of our model's fundamental parameters, it is very easy to get a posterior simulation of these transformations: we simply apply the transformation to the simulated fundamental parameters. In practice, this transformation can be done post estimation, or during estimation in the `generated quantities` block in *Stan*. I am very much in favor of doing this in the `generated quantities` block, because *Stan* will then provide convergence diagnostics for all of these quantities.

For models of decision-making under risk, these transformed quantities of interest could be things like *certainty equivalents*, which convert utility measured in the units von Neumann–Morgenstern (VNM) utility into units of currency. These are much easier to interpret and compare between participants. I also like to interpret the logit choice precision parameter λ by transforming it into a predicted probability. For example, if $\bar{u}$ is the largest possible utility difference in an experiment, the following quantity is the upper bound on the probability that a participant chooses the action that maximizes their utility:

$$(1 + \exp(-\bar{u}\lambda))^{-1}$$

This converts the units of λ, which are *inverse* utility to units, into a probability. I find this much easier to interpret and compare between participants.

18.3 Introducing some important models

18.3.1 Expected utility theory

A good place to start with modeling this kind of data is to assume that participants make choices according to expected utility theory (EUT). This means that we can write their maximization problem for their tth decision as:

$$\max_y E(u_i(X) \mid y)$$

Where $u(X)$ is participant i's utility from monetary prize X. The distribution $X \mid y$ is determined by the lottery they chose.

While there are many ways we could make a simplifying assumption about the utility function $u_i(\cdot)$, a common choice is to assume the constant relative risk-aversion specification:

$$u_i(x) = \frac{x^{1-r_i}}{1-r_i}$$

where r_i is the coefficient of relative risk-aversion. That is, the participant is risk-neutral if $r_i = 0$, risk-averse if $r_i > 0$, and risk-loving when $r_i > 0$.

[79] Not doing so is called a "common ratio" choice pattern. See Blavatskyy, Panchenko, and Ortmann (2023) for a literature review on this choice pattern.

[80] For example Harrison and Ng (2016) keeps their three prizes constant for the 80 decisions made in their battery.

The second parameter in the model comes from specifying a probabilistic choice rule. Here I will use logit choice with the contextual utility normalization (Wilcox 2011), so coding $y_{i,t} = 1$ as indicating that the participant chose the left lottery, we have:

$$p(y_{i,t} \mid r_i, \lambda_i) = \Lambda\left(\lambda_i \left[\frac{E(u(X) \mid L_t) - E(u(X) \mid R_t)}{u_i(\overline{x}_t) - u_i(\underline{x}_t)}\right]\right)$$
$$\Lambda(x) = (1 + \exp(-x))^{-1}$$

where $\lambda_i > 0$ is the choice precision parameter, and $\overline{x}_t$ and $\underline{x}_t$ are the largest and smallest prize in lottery pair t, respectively.

While we might be interested in our estimates of parameters r_i and λ_i themselves, we might also like to know what they imply about other economically meaningful quantities. One of these quantities could be a *certainty equivalent* of a lottery, which is the amount of money a participant would be indifferent between taking and receiving a draw from the lottery. That is, we define the certainty equivalent $C(L)$ for lottery L as:

$$u(C(L)) = E(u(X) \mid L)$$

which for the CRRA EUT model we are using becomes:

$$C(L) = \left(\sum_{k=1}^{3} x_k^{1-r_i} q_k^L\right)^{\frac{1}{1-r_i}}$$

where $\{x_k\}_{k=1}^3$ are the three prizes for the lottery, and $\{q_k\}_{k=1}^3$ are the probabilities of winning those prizes. Certainty equivalents are good at communicating information about participants' preferences over lotteries because they convert units of Von-Neumann Mogernetern utility, which are hard to interpret and impossible to compare between participants, into dollars, which are easy to interpret and compare. Since they are just a function of the model's fundamental parameters, certainty equivalents are also very easy to calculate in *Stan*'s `generated quantities` block, so I will do this for all of the models I estimate. Specifically, I will calculate certainty equivalents for the following lotteries:[81]

L_1 : a 95 percent chance of winning 10, and a 5 percent chance of winning 80

L_2 : a 5 percent chance of winning 10, and a 95 percent chance of winning 80

In order to estimate this model at the individual level, or to estimate a pooled model, we need to choose priors for the parameters r and λ. As I discuss in detail in Bland (2023b), choosing priors for structural models (and particularly for estimating risk preferences) is not something we should take lightly. This is partly because there are many non-linearities in these models, but we should not take this problem less seriously for linear models.

A good first step in formalizing priors is to ask what are the possible values for a parameter, however unlikely. That is, we pin down the *support* of the distribution. For r, the parameter in the CRRA utility function, any real number gives us a valid utility function.[82] In the case where we want our prior to cover the entire real number line, a good choice is to use the normal distribution:

$$r_i \sim N(m_r, s_r^2)$$

[81] These are lotteries where I suspect that the RDU model (discussed below) will produce very different estimates than the EUT model.

[82] This is not the case for all experiments. In our case it works because all lottery prizes (after adding in the show-up fee and endowment) are strictly positive. In other experiments, however, we need to be more careful. For example after adding in the show-up fee to the Hey (2001) dataset, the smallest prize is zero. In this case we would need to restrict r to the interval $(-\infty, 1)$. Alternatively, we could use a different functional form for the utility function. In other instances, we may be specifically interested in participants' treatment of gains and losses, and so in that case our utility function would need to permit negative and positive prizes.

For the choice precision parameter λ_i, we need a prior that restricts this to positive real numbers. This is because negative values imply that the participant will choose the lottery with the *smaller* expected utility more often than the lottery with the larger. While there are of course many families of distribution that cover positive real numbers, I like to choose *log-normal* priors for such parameters. This is mainly because it makes going to a hierarchical model very easy. Therefore, in order to pin down the $\lambda_i > 0$ restriction, we will choose:

$$\log \lambda_i \sim N(m_\lambda, s_\lambda^2)$$

We now need to choose values for m_r, s_r, m_λ, and s_λ. While this is an interesting exercise in itself, I will direct you to previous work I did in Bland (2023b), where I calibrated these priors to be:

$$r_i \sim N(0.27, 0.36^2)$$
$$\log \lambda_i \sim N(3.45, 0.59^2)$$

For r_i, this matches nicely with the distribution of estimates from Holt and Laury (2002). For λ_i, this covers most of the interesting range of choice precisions when choosing between (i) the middle prize for sure and (ii) a 50-50 mix of the low and high prizes.[83] Before moving on, let's investigate one of the implications of this prior just to make sure that it is doing (at least approximately) what we think it is doing.[84] Figure 54 shows the implied prior distribution of the probability of choosing a 50-50 mix of \$80 and \$10 over \$45 for sure. These prizes are for the most frequent context presented to participants in the experiment. Here we can see a large accumulation of mass at zero, which is to be expected, as the vast majority of the prior mass is in the risk-averse range. The rest of the distribution is reasonably spread out over the unit interval. At this point, if this plot looks surprising to you, you should go back and think about whether your prior is implying what you want it to.

```
set.seed(42)

d<-tibble(
  r = rnorm(10000,mean=0.27,sd=0.36),
  lambda = rnorm(10000,mean=3.45,sd=0.59) |> exp()
) |>
  mutate(
    uMin = (10^(1-r))/(1-r),
    uMax = (80^(1-r))/(1-r),
    uMid = (45^(1-r))/(1-r),

    lDU = lambda*(0.5*uMin+0.5*uMax-uMid)/(uMax-uMin),
    prob = 1/(1+exp(-lDU))
  )

(
  ggplot(d,aes(x=prob))
  +geom_histogram(aes(y=after_stat(density)),bins=30)
  +theme_bw()
)
```

Now that we have fully specified the likelihood and prior, we can go ahead and estimate the model. Here is the *Stan* file I wrote to estimate the EUT representative agent model.

```
data {
  int<lower=0> N; // total number of decisions
```

[83] I calibrated these priors with the Harrison and Ng (2016) experiment in mind, but they are similar enough designs with similar enough payoff scales that I think it is still appropriate to use these priors here.

[84] Please have a look at Bland (2023b) for a detailed scrutiny of these priors.

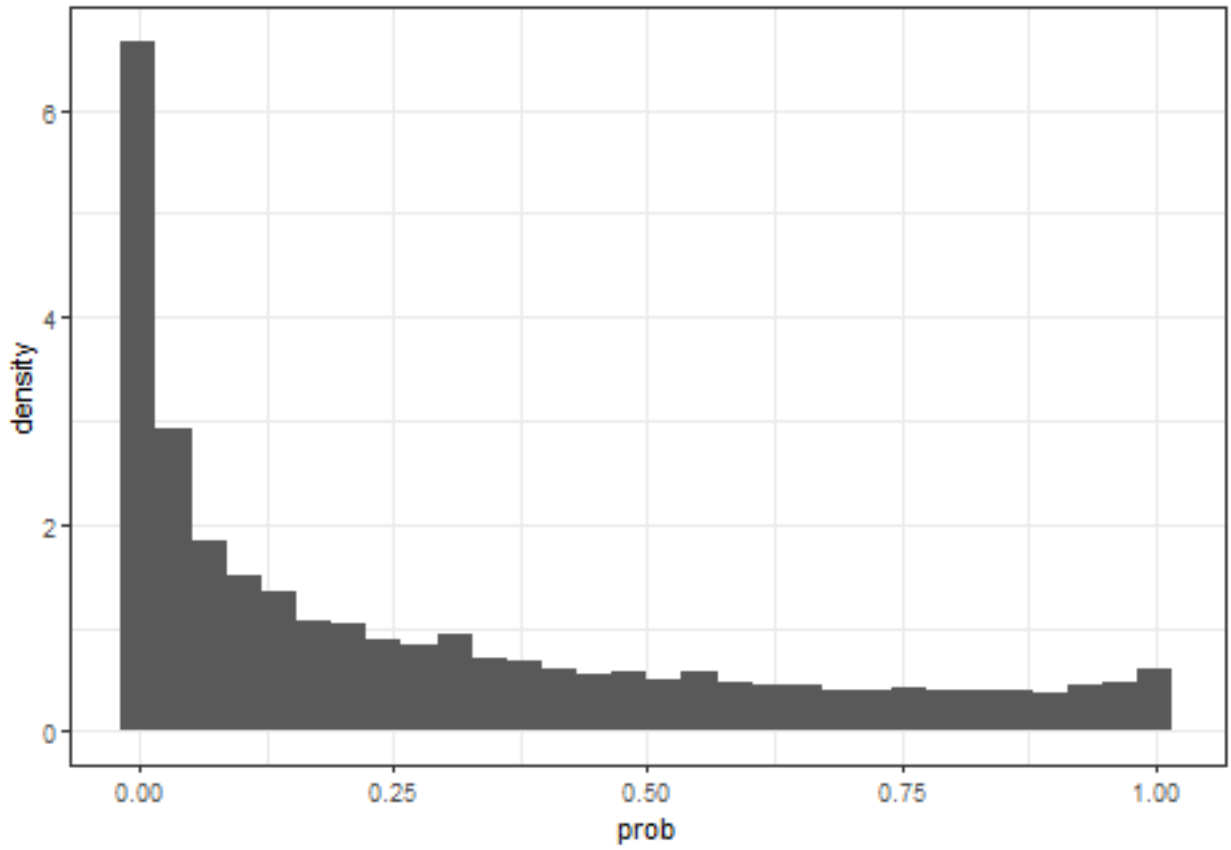


Figure 54: prior distribution of the probability of choosing a 50-50 mix of $80 and $10 over $45 for sure.

```
  int Left[N]; // indicator for chose the Left lottery

  matrix[N,3] pLeft;
  matrix[N,3] pRight;
  matrix[N,3] prizes;
  matrix[N,2] prizerange;

  vector[2] prior_r;
  vector[2] prior_lambda;


  // information about the certainty equivalents
  int nCE; // number of certainty equivalents to evaluate
  vector[3] CEprizes[nCE]; // prizes for the lottery for certainty equivalents
  vector[3] CEprobs[nCE]; // probabilities for the lottery for certainty equivalents



}

transformed data {
  // We are going to have to compute this difference a lot, so it is best to pre-compute it
  matrix[N,3] dprob = pLeft-pRight;

}

parameters {

  real r;
  real<lower=0> lambda;

}


model {

  // priors
  target += normal_lpdf(r | prior_r[1],prior_r[2]);
  target += lognormal_lpdf(lambda | prior_lambda[1],prior_lambda[2]);


  // likelihood

  target += bernoulli_logit_lpmf( Left |
    lambda * ((
      dprob.*pow(prizes,1.0-r)
    )*rep_vector(1.0,3)
    )
    ./ // contextual utility normalization
    (
      pow(prizerange[,2],1.0-r)-pow(prizerange[,1],1.0-r)
    )
  );
```

```
}
generated quantities {

  // calculate certainty equivalents

  vector[nCE] CE;

  for (cc in 1:nCE) {

    CE[cc] = pow(pow(CEprizes[cc],1.0-r)'*CEprobs[cc],1.0/(1.0-r));

  }



}
```

Figure 55 summarizes the parameter estimates from the participant-specific estimations (black curves and red error bars), alongside the prior distribution (blue curves). Compared to the prior, we can see that participants are mostly *less* precise and *more* risk-averse than we expected. I would not read too much into this right now, as we will be taking a more flexible model to the data soon.

```
EUTIndividual<-readRDS("Code/RiskPreferences/Estimates_individual.rds") |>
  filter(model=="EUT") |>
  group_by(parameter) |>
  arrange(mean) |>
  mutate(cumulative = (1:n())/n()) |>
  ungroup()

prior<-tibble(
  parameter = "r",
  x =  seq(-2,3,length=1001),
  cdf = pnorm(x,mean=0.27,sd=0.36)
) |>
  rbind(
    tibble(
      parameter = "lambda",
      x = seq(0,110,length=1001),
      cdf = pnorm(log(x),mean=3.45,sd=0.59)
    )
  )



(
  ggplot()
  +stat_ecdf(data=EUTIndividual |> filter(parameter !="lp__"),aes(x=mean))
  +geom_line(data=prior,aes(x=x,y=cdf),color="blue")
  +theme_bw()
  +geom_errorbar(alpha=0.2,color="red",data=EUTIndividual |> filter(parameter !="lp__"),aes(y=cumulative
  +facet_wrap(~parameter,scales="free")
  +ylab("Cumulative probability")
  +xlab("Estimate")
```

)

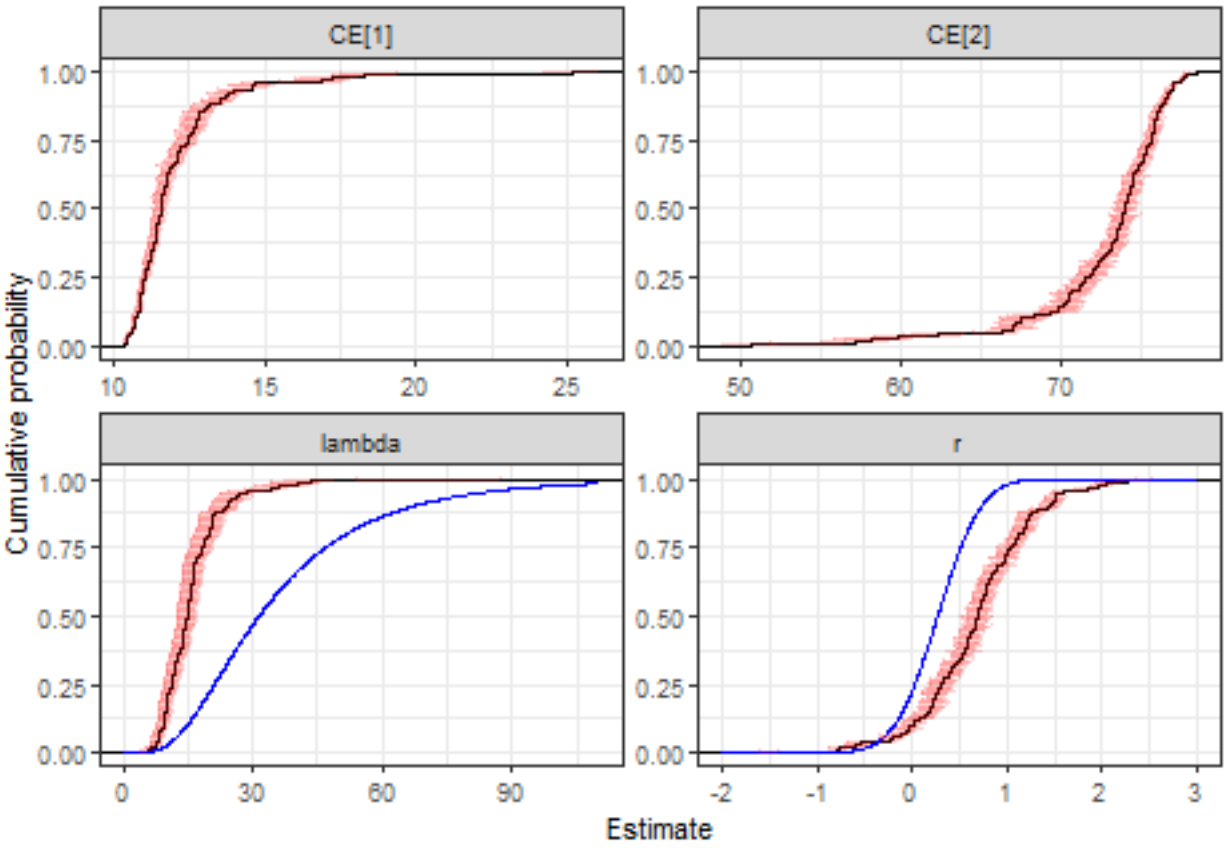


Figure 55: Estimates from individual estimation of the EUT model. Empirical cumulative density functions (black curves) of individual posterior means (black lines) and 50% Bayesian credible regions (red lines). Blue curves show the prior cumulative density functions. CE[1] is the certainty equivalent for a 95% channce of winning \$10, \$80 otherwise. CE[2] is the certainty equivalent for a 5% chance of winning \$10, \$80 otherwise.

18.3.2 Rank-dependent utility (Quiggin 1982)

So where do we go from here? We will of course be estimating a hierarchical model using all of the data, but before then we will stay in the realm of single participant estimation, and estimate a more flexible model. One such model is the *rank-dependent utility* (RDU) model (Quiggin 1982). Compared to the EUT model, where probabilities enter into the utility function *linearly* (hence the "expected" part), the RDU model weights probabilities. In particular, RDU participants distort the *cumulative* distribution of prizes relative to an EUT participant. The RDU model uses a *probability weighting function*, which I will denote $\omega_i(p)$, to distort these cumulative probabilities. That is, when the actual cumulative probability for a prize is p, RDU participants will behave as if it is actually $\omega_i(\mathrm{p})$. In this application, I adopt the convention that prizes are ranked from lowest to highest, but in other applications the reverse is also common.

For this application, I will use the Prelec (1998) specification of the probability-weighting function:[85]

[85] In an early version of Bland (2023b), I used the following re-parameterization of the Prelec weighting function:

$$\omega_i(p) = \exp(-(-\log \rho_i)^{1-\psi_i}(-\log p)^{\psi_i}), \quad \rho_i \in (0,1),\ \psi_i > 0$$

That is, $\eta_i = (-\log \rho_i)^{1-\psi_i}$. This re-parameterization has a useful interpretation in that the weighting function crosses the 45° line at $p = \rho_i$, and the function at this point has a slope of ψ_i. That is: $\omega_i(\rho_i) = \rho_i$, and $\omega_i'(\rho_i) = \psi_i$. EUT is then nested within RDU if and only if $\psi_i = 1$. While I still like this re-parameterization for its *interpretation*, it is a mess for estimation.

$$\omega_i(p) = \exp(-\eta_i(-\log p)^{\psi_i})$$

where $\eta_i, \psi_i > 0$ are the function's parameters. When $\eta_i = \psi_i = 1$ this function becomes $\omega_i(p) = p$, which means that cumulative probabilities are *not* distorted, and so we are back to EUT.

These weighted cumulative probabilities are then converted into (de-cumulated, for want of a better word) "decision-weights" π_i, which then enter into the decision-maker's utility function as follows:

$$\begin{aligned} V_i(q,x) &= \sum_{k=1}^{K} \pi_{i,k}(q) u_i(x_k) \\ \text{where: } \pi_{i,1}(q) &= \omega_i(q_1) \\ \pi_{i,2}(q) &= \omega_i(q_1+q_2) - \omega(q_1) \\ \pi_{i,k}(q) &= \omega_i\left(\sum_{j=1}^{k} q_k\right) - \omega_i\left(\sum_{j=1}^{k-1} q_k\right) \end{aligned}$$

where q_k and x_k are the kth probability and prize of the lottery, respectively.

I will use the same logit choice rule, CRRA parameterization of the utility function, and contextual utility normalization in this model as I do with the EUT models.

Before moving on to estimating the model, it is important to note a computational persnickety-ness that comes with using the Prelec weighting function: it will return an `Inf` or a `NaN` when we try to evaluate it at cumulative probabilities equal to zero or one. This is a real problem, because our lotteries will *always* have a cumulative probability equal to one, and many experiment designs, including Hey (2001), have cumulative probabilities equal to zero. Fortunately, the *limits* as cumulative probabilities approach these value are just zero and one, so we can do a bit of a computational monkey trick to fix the problem. Specifically, instead we can evaluate:

$$\begin{aligned} &\omega_i\left[p + 0.01(I(p=0) - I(p=1)\right]\left(1 - I(p=0) - I(p=1)\right) + I(p=1) \\ &= \begin{cases} \omega_i(p) & \text{if } p \in (0,1) \\ 0 & \text{if } p = 0 \\ 1 & \text{if } p = 1 \end{cases} \end{aligned}$$

where $I(\cdot)$ is the indicator function. That is, inside the function ω, we add a little bit if $p = 0$, and we subtract a little bit if $p = 1$. We then correct this by ensuring the function returns exactly 0 or 1 in these cases. In practice, I have found it a bad idea to ask *Stan* whether a real-valued data element is exactly equal to something. This is because there can be minuscule rounding errors in the calculations that get to this number. Hence, you will see in the *Stan* file below I have `if` statements of the form `if (q<tol)`, where `tol` is a very small positive number, instead of one of the form `if (p==0)`.

Before estimating this model, we need to do a quick prior calibration. Since r_i and λ_i have the same interpretation as the EUT model, I will keep these priors as:

$$\begin{aligned} r_i &\sim N(0.27, 0.36^2) \\ \log \lambda_i &\sim N(3.45, 0.59^2) \end{aligned}$$

This is because there the estimation does badly when ρ_i is close to zero or one. That is, the estimation has trouble when the probability weighting function is close to either entirely above the 45° line, or entirely below the 45° line. Therefor3e *estimating* ρ_i as a fundamental parameter is not a good idea, but you still may want to calculate it in the `generated quantities` block so you can use this nice interpretation.

For η_i and ψ_i, log-normal priors are appropriate because these parameters must be non-negative. Furthermore, centering their medians on $\eta_i = \psi_i = 1$ will center these priors on the EUT model, which seems as good a place as any to start without considering any more information. I ended up using the following priors for these parameters:

$$\log \eta_i \sim N(\log(1), 1^2)$$
$$\log \psi_i \sim N(\log(1), 1^2)$$

Here is the *Stan* file I wrote to estimate the RDU model and calculate the certainty equivalents:

```
functions {

  matrix decisionWeights(
    real psi, real eta,
    data matrix cprob, data matrix cprob0, data matrix cprob1
  ) {

    int n = num_elements(cprob)%/%3;

    /* Weighted cumulative probabilities
    Here we need to watch out for p=0 and p=1 problems, where the weighting
    function is undefined, but is 0 and 1 in the limit respectively. Here
    I avoid getting NaNs by adding a little bit when p=0, and subtracting
    a little bit when p=1. I then substitute in the limit exactly just below
    this
    */

    matrix[n,3] wcprob = exp(
      -eta * pow(-log(cprob+0.01*(cprob0-cprob1)),psi)
    );
    wcprob = wcprob.*(1.0-cprob0-cprob1)+cprob1;

    // convert to decision weights

    matrix[n,3] wprob = wcprob;

    for (kk in 2:3) {
      wprob[,kk] = wcprob[,kk]-wcprob[,kk-1];
    }

    return wprob;
  }

  /* Calculating the certainty equivalent is a little more tricky for RDU, so
  I am writing a function to do it so as not to clutter up the generated quantities
  block
  */

  real calcCE(vector probs, vector prizes,real r, real psi,real eta,real tol) {

    vector[3] cprobs = probs;

    for (pp in 2:3) {
```

```
      cprobs[pp] = cprobs[pp-1]+probs[pp];
    }

    vector[3] wcprobs = exp(
      -eta * pow(-log(cprobs),psi)
    );

    wcprobs[3] = 1.0; // avoids getting a NaN. This mucst be 1
    for (pp in 1:2) {
      if (wcprobs[pp]<tol) {
        wcprobs[pp] = 0.0;
      }
    }

    vector[3] wprobs = wcprobs;
    for (pp in 2:3) {
      wprobs[pp] = wcprobs[pp]-wcprobs[pp-1];
    }


    return pow(pow(prizes,1.0-r)'*wprobs,1.0/(1.0-r));


  }

}


data {
  int<lower=0> N; // total number of decisions

  int Left[N]; // indicator for chose the Left lottery

  matrix[N,3] pLeft;
  matrix[N,3] pRight;
  matrix[N,3] prizes;
  matrix[N,2] prizerange;

  vector[2] prior_r;
  vector[2] prior_lambda;
  vector[2] prior_psi;
  vector[2] prior_eta;

  real<lower=0> tol;

  // information about the certainty equivalents
  int nCE; // number of certainty equivalents to evaluate
  vector[3] CEprizes[nCE]; // prizes for the lottery for certainty equivalents
  vector[3] CEprobs[nCE]; // probabilities for the lottery for certainty equivalents
}

transformed data {
```

```
  matrix[N,3] cprobLeft = pLeft;
  matrix[N,3] cprobRight = pRight;

  matrix[N,3] cprobLeft0 = rep_matrix(0.0,N,3);
  matrix[N,3] cprobLeft1 = rep_matrix(0.0,N,3);
  matrix[N,3] cprobRight0 = rep_matrix(0.0,N,3);
  matrix[N,3] cprobRight1 = rep_matrix(0.0,N,3);

  for (kk in 2:3) {
    cprobLeft[,kk] = cprobLeft[,kk-1]+pLeft[,kk];
    cprobRight[,kk] = cprobRight[,kk-1]+pRight[,kk];
  }

  for (ii in 1:N) {
    for (kk in 1:3) {

      if (cprobLeft[ii,kk]<tol) {
        cprobLeft0[ii,kk] = 1.0;
      }
      if (cprobLeft[ii,kk]>(1.0-tol)) {
        cprobLeft1[ii,kk] = 1.0;
      }

      if (cprobRight[ii,kk]<tol) {
        cprobRight0[ii,kk] = 1.0;
      }
      if (cprobRight[ii,kk]>(1.0-tol)) {
        cprobRight1[ii,kk] = 1.0;
      }

    }
  }


}

parameters {

  real r;
  real<lower=0> lambda;
  real<lower=0> psi;
  real<lower=0> eta;

}


model {

  // priors
  target += normal_lpdf(r | prior_r[1],prior_r[2]);
  target += lognormal_lpdf(lambda | prior_lambda[1],prior_lambda[2]);
  target += lognormal_lpdf(psi | prior_psi[1],prior_psi[2]);
  target += lognormal_lpdf(eta | prior_eta[1],prior_eta[2]);
```

```
  // likelihood

  matrix[N,3] dprob =
    decisionWeights(psi, eta, cprobLeft, cprobLeft0, cprobLeft1)
     -
    decisionWeights(psi, eta, cprobRight, cprobRight0, cprobRight1)
    ;



  target += bernoulli_logit_lpmf( Left |
    lambda * ((
      dprob.*pow(prizes,1.0-r)
    )*rep_vector(1.0,3)
    )
    ./ // contextual utility normalization
    (
      pow(prizerange[,2],1.0-r)-pow(prizerange[,1],1.0-r)
    )
  );


}

generated quantities {

  // calculate certainty equivalents

  vector[nCE] CE;

  for (cc in 1:nCE) {

    CE[cc] = calcCE(CEprobs[cc], CEprizes[cc],r,psi,eta,tol);

  }
}
```

Figure 56 summarizes the estimates from these participant-specific estimations. In particular note that there appears to be a large concentration of estimates of η_i and ψ_i around one, which you can see in this Figure as steep parts of the empirical cdf of posterior means (black lines). This suggests that, at least for a reasonable minority of participants, the $\eta_i = \psi_i = 1$ restriction does not seem too bad.

```
RDUIndividual<-readRDS("Code/RiskPreferences/Estimates_individual.rds") |>
  filter(model=="RDU") |>
  arrange(id,parameter) |>
  group_by(parameter) |>
  arrange(mean) |>
  mutate(cumulative = (1:n())/n()) |>
  ungroup()

prior<-tibble(
  parameter = "r",
  x =  seq(-2,3,length=1001),
```

```
  cdf = pnorm(x,mean=0.27,sd=0.36)
) |>
  rbind(
    tibble(
      parameter = "lambda",
      x = seq(0,110,length=1001),
      cdf = pnorm(log(x),mean=3.45,sd=0.59)
    )
  ) |> rbind(
    tibble(
      parameter = "psi",
      x = seq(0,3,length=1001),
      cdf = pnorm(log(x),mean=0,sd=1)
    )
  ) |> rbind(
    tibble(
      parameter = "eta",
      x = seq(0,8,length=1001),
      cdf = pnorm(log(x),mean=0,sd=1)
    )
  )



(
  ggplot()
  +stat_ecdf(data=RDUIndividual |> filter(parameter !="lp__"),aes(x=mean))
  +geom_line(data=prior,aes(x=x,y=cdf),color="blue")
  +theme_bw()
  +geom_errorbar(alpha=0.3,color="red",data=RDUIndividual |> filter(parameter !="lp__"),aes(y=cumulative
  +facet_wrap(~parameter,scales="free")
  +ylab("Cumulative probability")
  +xlab("Estimate")
)
```

However a closer look at the posterior means for these parameters in Figure 57 shows that there is no real concentration around $\eta_i = \psi_i = 1$. That is, it appears that the RDU model is implying substantially different behavior than EUT.

```
plotThis<-RDUIndividual |>
  filter(parameter == "eta" | parameter == "psi") |>
  pivot_wider(
    id_cols = id,
    names_from = parameter,
    values_from =  c(mean,X25.,X75.)
  )

(
  ggplot(plotThis,aes(x=mean_eta,y=mean_psi))
  +geom_point()
  +geom_errorbar(aes(xmin = X25._eta,xmax=X75._eta),alpha=0.3)
  +geom_errorbar(aes(ymin = X25._psi,ymax=X75._psi),alpha=0.3)
  +theme_bw()
  +xlab(expression(eta))
```

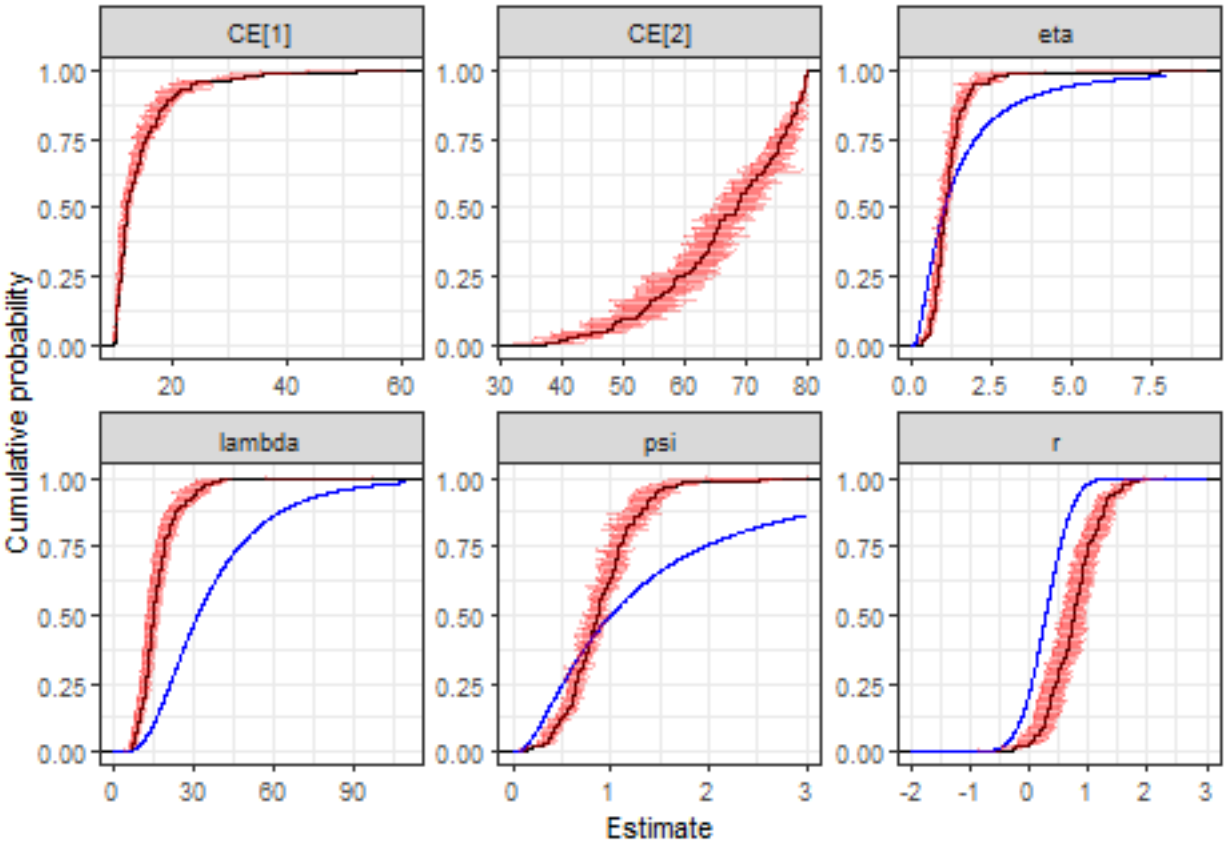


Figure 56: Estimates from individual estimation of the RDU model. Empirical cumulative density functions (black curves) of individual posterior means (black lines) and 50% Bayesian credible regions (red lines). Blue curves show the prior cumulative density functions. CE[1] is the certainty equivalent for a 95% channce of winning \$10, \$80 otherwise. CE[2] is the certainty equivalent for a 5% chance of winning \$10, \$80 otherwise.

```
  +ylab(expression(psi))
  +geom_hline(yintercept = 1.0,linetype="dashed")+geom_vline(xintercept = 1.0,linetype="dashed")
)
```

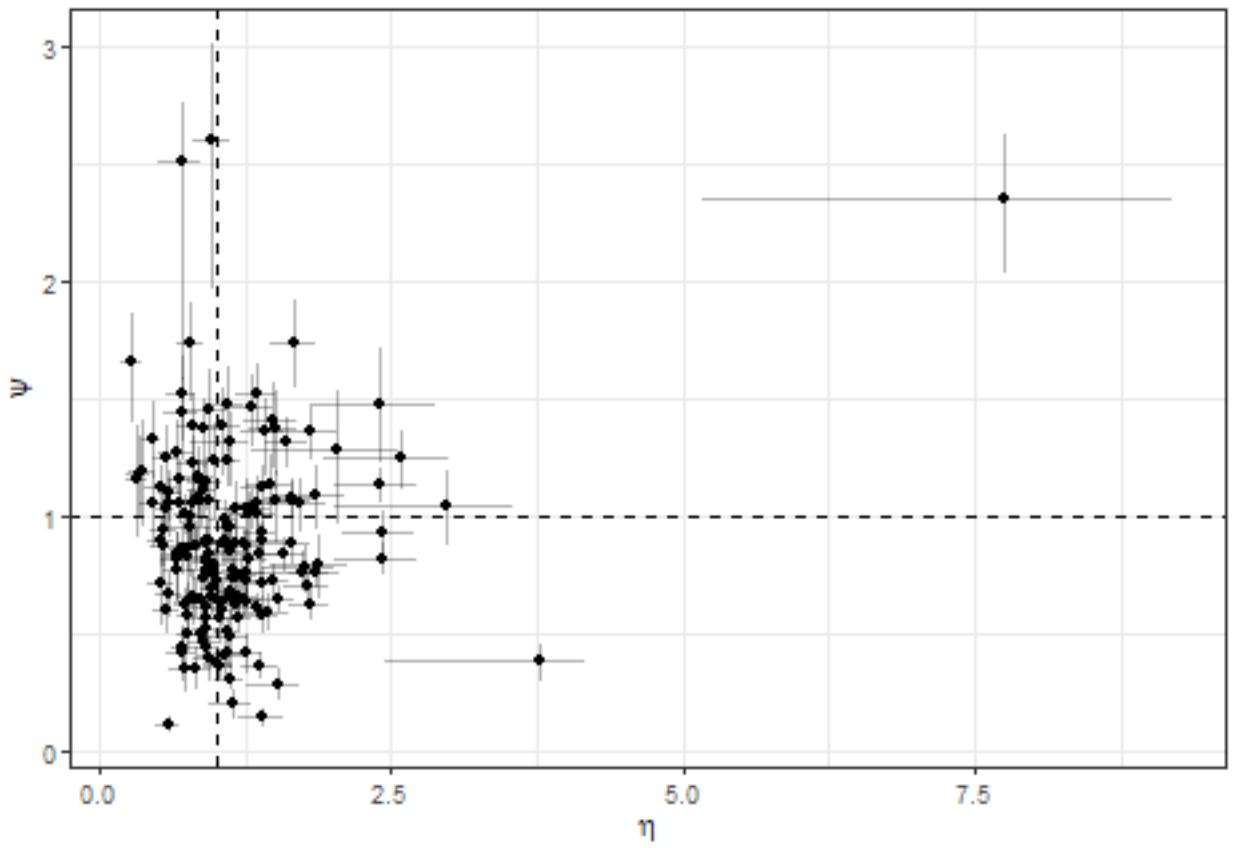


Figure 57: Individual posterior means of η_i and ψ_i (error bars show a 50% Bayesian credible region) from the RDU model. When both of these parameters are equal to one, the RDU model nests EUT.

18.3.3 Comparing the certainty equivalents estimated using EUT and RDU

Before we move on to the hierarchical model in the next section, I was interested to see how differently the EUT and RDU models estimated the certainty equivalent. That is, *a priori* it is not clear whether or not these will produce estimates that are substantially different. This could be for two reasons. Firstly, since EUT is a special case of RDU, it could be that the parameter estimates from the RDU model are close enough to the $\eta_i = \psi_i = 1$ restriction that behavior is effectively EUT. Secondly, even if this parameter restriction does *not* approximately hold, it could be that the estimated preferences over the lotteries are similar, and so we would estimate similar certainty equivalents. To investigate this, Figure 58 plots the estimated certainty equivalents from the EUT model (horizontal coordinate) and the RDU model (vertical coordinate). The dashed red line is a 45° line, and so if the estimates fall on or close to this line the models are estimating similar certainty equivalents. This is especially not the case for "CE[2]", which is the certainty equivalent of a 5% chance of winning \$10, and a 95% chance of winning \$80. In this panel, we can see that the RDU model mostly estimates smaller certainty equivalents than the EUT model. Put differently, the EUT is over-valuing this lottery relative to the RDU model.

```
d<-readRDS("Code/RiskPreferences/Estimates_individual.rds") |>
  filter(grepl("CE",parameter)) |>
  pivot_wider(
```

```
    id_cols = c(id,parameter),
    names_from = model,
    values_from = c(X25.,mean,X75.)
  )


(
  ggplot(d,aes(x=mean_EUT,y=mean_RDU))
  +facet_wrap(~parameter,scales="free")
  +geom_errorbar(aes(ymin = X25._RDU,ymax=X75._RDU),alpha=0.3)
  +geom_errorbar(aes(xmin=X25._EUT,xmax=X75._EUT),alpha=0.3)
  +theme_bw()
  +geom_point()
  +geom_abline(slope=1,intercept=0,color="red",linetype="dashed")
  +xlab("Certainty equivalent estimated using EUT ($)")+ylab("Certainty equivalent estimated using RDU
)
```

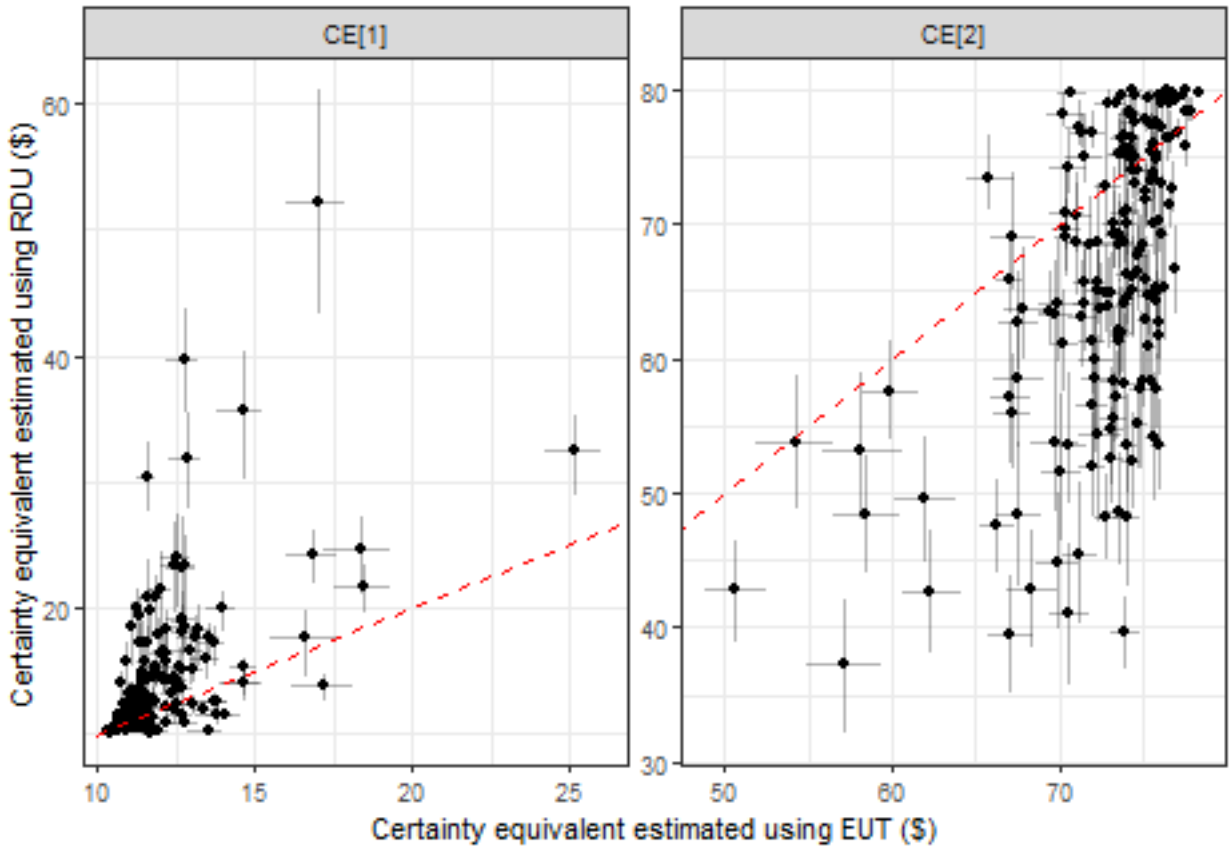


Figure 58: Certainty equivalents estimated using the EUT and RDU models. Dots show posterior means. Error bars show 50% Bayesian credible regions.

18.4 A hierarchical specification

Of course, this wouldn't be the full Bayesian treatment without at least one hierarchical model. Here I am going to estimate the RDU model using data from all 175 participants, allowing each of the RDU model's parameters to be participant-specific random draws from a (transformed) multivariate normal distribution.

This specification therefore has the same likelihood as the individual-specific models (we just add up the likelihood contribution for each participant), and I will not be doing anything too novel by assuming a (transformed) multivariate normal distribution for these individual-level parameters:

$$\begin{pmatrix} r_i \\ \log \lambda_i \\ \log \eta_i \\ \log \psi_i \end{pmatrix} \sim N\left(\mu, \text{diag\_matrix}(\tau)\Omega\text{diag\_matrix}(\tau)\right)$$

and then assign (hyper)-priors to the population-level parameters:

$$\begin{aligned} \mu_r &\sim N(0.27, 0.36^2) \\ \mu_\lambda &\sim N(3.45, 0.59^2) \\ \mu_\eta &\sim N(0, 1) \\ \mu_\psi &\sim N(0, 1) \\ \tau_r,\ \tau_\lambda,\ \tau_\eta,\ \tau_\psi &\sim \text{Cauchy}^+(0, 1) \\ \Omega &\sim \text{LKJ}(4) \end{aligned}$$

Note that I have carried forward my priors for r, λ, η, and μ in the participant-specific RDU model to be the priors for the relevant elements of μ. This means that the "representative agent" of the hierarchical model (i.e. the fictitious median participant with (transformed) parameters μ) has the same priors as the participant-specific model. Really I suspect that we could be *much* more "informative" about these, because they are measures of central tendency, rather than for an individual participant. However the participant-specific priors serve at least as a good starting point for forming a prior for μ. The Cauchy$^+(0, 1)$ (hyper)-priors for the standard deviation parameters (τ) actually cover a *lot* of the parameter space that implies a lot of dispersion in the individual-level parameters (especially for the parameters transformed by logs), so I am not too worried about being overly "informative" here. Finally, the LKJ(4) prior for Ω is centered on, and peaked at, zero correlation, but does not rule out large correlations too much.

Here is is the *Stan* program I wrote to estimate this model. While there is not much here that is not either taken straight from the participant-specific RDU model, or standard Bayesian hierarchical model stuff, it is importantan to note a few things. Firstly, I make sure that r_i, λ_i, η_i, and ψ_i are generated in the `transformed parameters` block so I can access them post estimation (this would not be the case if I calculated them in the `model` block). Also, I am using the data variable `idstatend` to keep track of where a participant's data show up in the pooled data. This requires that the pooled data is sorted by participant.

```
functions {

  matrix decisionWeights(
    real psi, real eta,
    data matrix cprob, data matrix cprob0, data matrix cprob1
  ) {

    int n = num_elements(cprob)%/%3;

    /* Weighted cumulative probabilities
    Here we need to watch out for p=0 and p=1 problems, where the weighting
    function is undefined, but is 0 and 1 in the limit respectively. Here
    I avoid getting NaNs by adding a little bit when p=0, and subtracting
    a little bit when p=1. I then substitute in the limit exactly just below
    this
    */
```

```
    matrix[n,3] wcprob = exp(
      -eta * pow(-log(cprob+0.01*(cprob0-cprob1)),psi)
    );
    wcprob = wcprob.*(1.0-cprob0-cprob1)+cprob1;

    // convert to decision weights

    matrix[n,3] wprob = wcprob;

    for (kk in 2:3) {
      wprob[,kk] = wcprob[,kk]-wcprob[,kk-1];
    }

    return wprob;
  }

  /* Calculating the certainty equivalent is a little more tricky for RDU, so
  I am writing a function to do it so as not to clutter up the generated quantities
  block
  */

  real calcCE(vector probs, vector prizes,real r, real psi,real eta,real tol) {

    vector[3] cprobs = probs;

    for (pp in 2:3) {

      cprobs[pp] = cprobs[pp-1]+probs[pp];
    }

    vector[3] wcprobs = exp(
      -eta * pow(-log(cprobs),psi)
    );

    wcprobs[3] = 1.0; // avoids getting a NaN. This mucst be 1
    for (pp in 1:2) {
      if (wcprobs[pp]<tol) {
        wcprobs[pp] = 0.0;
      }
    }

    vector[3] wprobs = wcprobs;
    for (pp in 2:3) {
      wprobs[pp] = wcprobs[pp]-wcprobs[pp-1];
    }


    return pow(pow(prizes,1.0-r)'*wprobs,1.0/(1.0-r));


  }

}
```

```
data {
  int<lower=0> N; // total number of decisions

  int<lower=0> nparticipants;
  int id[N];
  int idstartend[nparticipants,2];

  int Left[N]; // indicator for chose the Left lottery

  matrix[N,3] pLeft;
  matrix[N,3] pRight;
  matrix[N,3] prizes;
  matrix[N,2] prizerange;

  vector[2] prior_MU[4];
  vector[4] prior_TAU;
  real prior_OMEGA;


  real<lower=0> tol;

  // information about the certainty equivalents
  int nCE; // number of certainty equivalents to evaluate
  vector[3] CEprizes[nCE]; // prizes for the lottery for certainty equivalents
  vector[3] CEprobs[nCE]; // probabilities for the lottery for certainty equivalents
}

transformed data {

  matrix[N,3] cprobLeft = pLeft;
  matrix[N,3] cprobRight = pRight;

  matrix[N,3] cprobLeft0 = rep_matrix(0.0,N,3);
  matrix[N,3] cprobLeft1 = rep_matrix(0.0,N,3);
  matrix[N,3] cprobRight0 = rep_matrix(0.0,N,3);
  matrix[N,3] cprobRight1 = rep_matrix(0.0,N,3);

  for (kk in 2:3) {
    cprobLeft[,kk] = cprobLeft[,kk-1]+pLeft[,kk];
    cprobRight[,kk] = cprobRight[,kk-1]+pRight[,kk];
  }

  for (ii in 1:N) {
    for (kk in 1:3) {

      if (cprobLeft[ii,kk]<tol) {
        cprobLeft0[ii,kk] = 1.0;
      }
      if (cprobLeft[ii,kk]>(1.0-tol)) {
        cprobLeft1[ii,kk] = 1.0;
      }
```

```
      if (cprobRight[ii,kk]<tol) {
        cprobRight0[ii,kk] = 1.0;
      }
      if (cprobRight[ii,kk]>(1.0-tol)) {
        cprobRight1[ii,kk] = 1.0;
      }

    }
  }


}

parameters {

  vector[4] MU;
  vector<lower=0>[4] TAU;
  cholesky_factor_corr[4] L_OMEGA;

  matrix[4,nparticipants] z;

}

transformed parameters {

  vector[nparticipants] r;
  vector[nparticipants] lambda;
  vector[nparticipants] eta;
  vector[nparticipants] psi;

  {

    matrix[4,nparticipants] theta = rep_matrix(MU,nparticipants)+ diag_pre_multiply(TAU,L_OMEGA)*z;

    r = theta[1,]';
    lambda = exp(theta[2,]');
    eta = exp(theta[3,]');
    psi = exp(theta[4,]');

  }

}


model {

  to_vector(z) ~ std_normal();

  // priors
  for (pp in 1:4) {
    MU[pp] ~ normal(prior_MU[pp][1],prior_MU[pp][2]);
    TAU[pp] ~ cauchy(0.0,prior_TAU[pp]);
  }
```

```
  L_OMEGA ~ lkj_corr_cholesky(prior_OMEGA);



  for (ii in 1:nparticipants) {

    int start = idstartend[ii,1];
    int end = idstartend[ii,2];

    real ri = r[ii];
    real lambdai = lambda[ii];
    real etai = eta[ii];
    real psii = psi[ii];

    matrix[end-start+1,3] dprob =
    decisionWeights(psii, etai, cprobLeft[start:end,], cprobLeft0[start:end,], cprobLeft1[start:end,])
     -
    decisionWeights(psii, etai, cprobRight[start:end,], cprobRight0[start:end,], cprobRight1[start:end,]
    ;

    target +=  bernoulli_logit_lpmf( Left[start:end] |
    lambdai * ((
      dprob.*pow(prizes[start:end,],1.0-ri)
    )*rep_vector(1.0,3)
    )
    ./ // contextual utility normalization
    (
      pow(prizerange[start:end,2],1.0-ri)-pow(prizerange[start:end,1],1.0-ri)
    )
  );

  }


}

generated quantities {

  matrix[4,4] OMEGA = L_OMEGA*L_OMEGA';

  // calculate certainty equivalents

  vector[nCE] CE[nparticipants];
  for (ii in 1:nparticipants) {
    for (cc in 1:nCE) {

      CE[ii][cc] = calcCE(CEprobs[cc], CEprizes[cc],r[ii],psi[ii],eta[ii],tol);

    }
  }
}
```

18.4.1 Population-level estimates

Table 42 shows posterior moments of the population-level parameters. Since all of the individual-level parameters except for r_i have *transformed* normal distributions, it is difficult to interpret the numbers corresponding to these parameters, so I will attempt to put them in context below. For r_i, we can see that the mean level of risk-aversion (1.439) is solidly in the "stay in bed" category defined by Holt and Laury (2002), although the standard deviation is large enough that its distribution will cover a wide range of risk preferences. We will check on this later when we look at the individual-level estimates.

```
rounding<-3

RDU<-"Code/RiskPreferences/FitRDU.rds" |>readRDS()

parlabels<-c(r="$r$",lambda="$\\lambda$",eta="$\\eta$",psi="$\\psi$")
partxt<-c(r="r",lambda="\u03bb",eta = "\u03b7",psi = "\u03c8")
popparlabels<-c(MU = "$\\mu$", TAU = "$\\tau$")

RDUsummary<-summary(RDU)$summary |>
  data.frame() |>
  rownames_to_column(var = "par") |>
  mutate(
    parameter = gsub("[^A-Za-z]+", "", par),
    parameter_index = parse_number(par),
    msd = paste0(mean |> round(rounding)," (",sd |> round(rounding),")")
  )

RDUpopulation<-RDUsummary |>
  filter(parameter=="MU" | parameter == "TAU" | parameter=="OMEGA") |>
  mutate(
    parlabel = parlabels[parameter_index]
  )

OMEGA<-RDUpopulation |>
  filter(parameter=="OMEGA") |>
  # get the right row and column indices
  mutate(
    row = floor(parameter_index/10),
    col = parameter_index-10*row
  ) |>
  filter(row<=col) |>
  mutate(
    rowlabel = parlabels[row],
    collabel = parlabels[col]
  ) |>
  pivot_wider(
    id_cols = rowlabel,
    names_from = collabel,
    values_from = msd,
    values_fill = ""
  ) |>
  mutate(
    parameter = paste("$\\Omega$"," - ",rowlabel)
  ) |>
  dplyr::select(-rowlabel)
```

Table 42: Population level estimates (posterior means with standard deviations in parentheses) from the hierarchical RDU model.

parameter	r	λ	η	ψ
μ	1.439 (0.073)	2.662 (0.04)	0.361 (0.03)	-0.391 (0.032)
τ	0.835 (0.065)	0.446 (0.036)	0.287 (0.03)	0.361 (0.029)
Ω - r	1 (0)	0.162 (0.1)	0.214 (0.111)	-0.114 (0.097)
Ω - λ		1 (0)	0.065 (0.123)	-0.084 (0.112)
Ω - η			1 (0)	0.14 (0.127)
Ω - ψ				1 (0)

```
TAB<- RDUpopulation |>
  filter(parameter!="OMEGA") |>
  pivot_wider(id_cols = parameter,
              names_from = parlabel,
              values_from = msd
              ) |>
  mutate(parameter = popparlabels[parameter]) |>
  rbind(OMEGA)

TAB |>
  kbl(caption = "Population level estimates (posterior means with standard deviations in parentheses) fi
  kable_classic(full_width=FALSE) |>
  row_spec(2,hline_after = T,extra_css = "border-bottom: 1px solid")
```

Table 42 also shows the estimated correlations between the (transformed) parameters, however it is difficult to determine from this Table whether or not these are substantially different to zero. Figure 59 shows 95% Bayesian credible regions for these correlation parameters. All of them cover zero.[86]

```
OMEGA<-RDUpopulation |>
  filter(parameter=="OMEGA") |>
  # get the right row and column indices
  mutate(
    row = floor(parameter_index/10),
    col = parameter_index-10*row,
    correlation = paste(partxt[row],"-",partxt[col])
  ) |>
  filter(row<col)

(
  ggplot(OMEGA,aes(x=correlation,ymin=X2.5.,ymax = X97.5.))
  +geom_errorbar()
  +theme_bw()
  +geom_hline(yintercept=0,linetype="dashed")
)
```

Before turning to the individual-level estimates, it is sometimes useful to investigate what these population-level parameters imply about the median or "representative agent" participant in the model. To show you how to do this, we will look at the implications of these estimates for the probability weighting function. Figure 60 shows the estimated probability weighting function for this representative agent. Here we can see

[86]We should be careful here in concluding from this plot that there is no correlation between the parameters. It would be better to re-estimate the model setting the correlation matrix to the identity matrix (i.e. no correlation), and then interpret the Bayes factor of these two models.

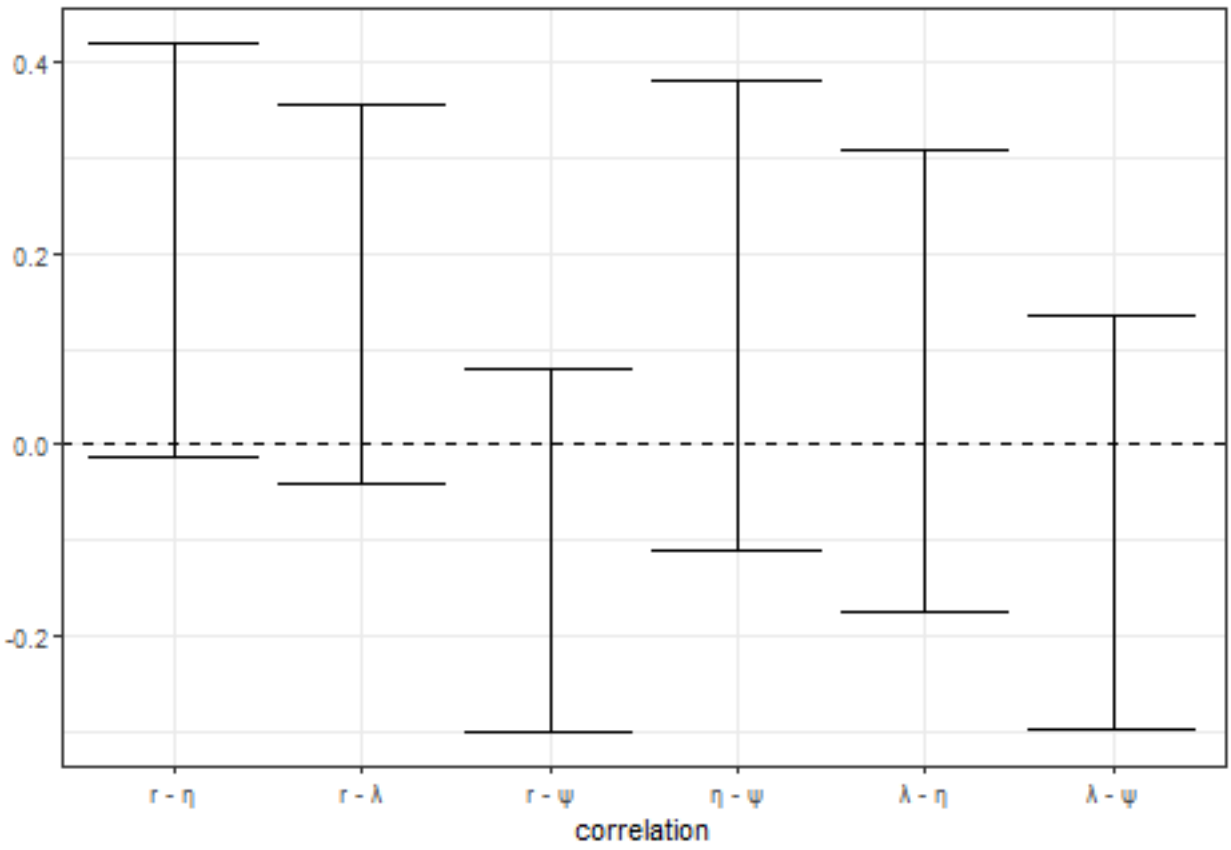


Figure 59: 95% Bayesian credible regions (2.5th-95th percentile) for the correlation parameters Ω

that most cumulative probabilities are under-weighted, especially those cumulative probabilities close to one.

```
# a probability-weighting function for every step of the posterior sample
d<-tibble(
  eta=exp(extract(RDU)$MU[,3]),
  psi=exp(extract(RDU)$MU[,4])
  ) |>
  mutate(
    s = 1:n()
  )|>
  expand_grid(
    p = c(0.001,seq(0.01,0.99,length=99),0.999)
  ) |>
  mutate(
    w = exp(-eta*(-log(p))^psi)
  )
# a summary of the posterior sample
dsum<-d |>
  group_by(p) |>
  summarize(
    mean = mean(w),
    min95 = quantile(w,0.025),
    max95 = quantile(w,0.975)
  )


(
  ggplot()
  +theme_bw()
  +geom_line(data=dsum,aes(x=p,y=mean))
  +geom_ribbon(data=dsum,aes(x=p,ymin=min95,ymax=max95),fill="black",alpha=0.2)
  +geom_abline(slope=1,intercept=0,linetype="dashed",color="blue")
  +ylab(expression(omega(p)))
)
```

18.4.2 Participant-level estimates

But hierarchical models also provide us with estimates for the participant-level parameters. In fact, we may be *more* interested in these, and the population-level parameters could just be means to an end in estimating participant-level quantities.[87] Figure 61 shows the individual-level estimates (black curves and red error bars) alongside the estimated population distributions (blue shaded regions, i.e. those implied by μ and τ). The interpretation of these estimates (excluding the estimated population distributions) is similar to those shown in Figure 56, which is from the participant-specific estimation done using the RDU model. In this case, though, our posterior estimates for each participant are informed by *all* of the data, not just the data from that participant. That is, we learn something about participant i's parameters not just through i's own decisions, but also because all the other participants $j \neq i$ help inform our prior for estimating i's parameters.

```
RDUi<-RDUsummary |>
  filter(
    parameter == "r" | parameter == "lambda" | parameter == "eta" | parameter=="psi" | parameter == "CE
  ) |>
  mutate(
```

[87] This, of course, depends on the inferential goal. If we are interested in making statements about the population as a whole, then we are primarily interested in μ, τ, and Ω. On the other hand, if we want to learn things about specific participants, then we will be more interested in the augmented data and the quantities derived from it.

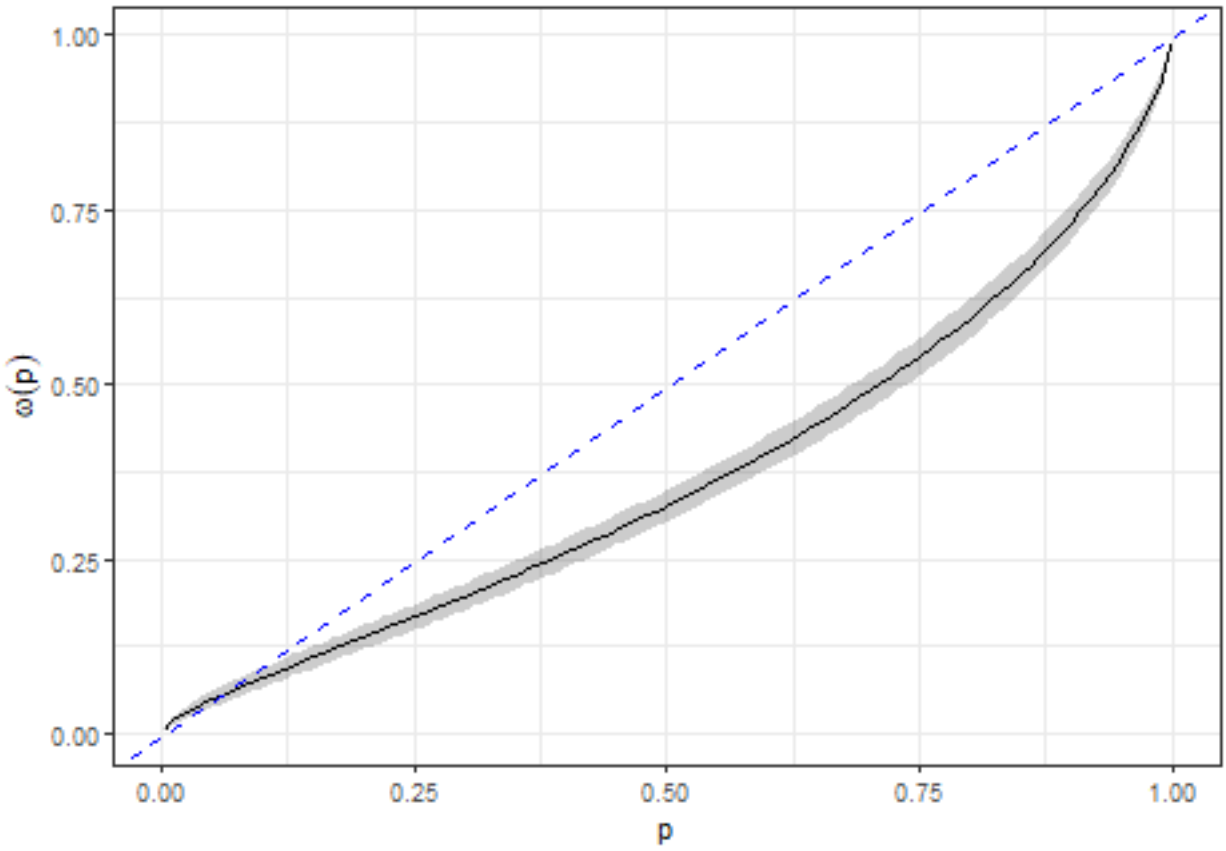


Figure 60: Probability weighting function for the median ('representative agent') participant in the hierarchical RDU model. Black curve shows the posterior mean. The shaded region is a 95% Bayesian credible region (2.5th-97.5th percentile), and the blue dashed line is a 45° line. The 45° line is also the probability weighting function for an EUT participant.

```
    id = ifelse(parameter=="CE",
                floor(parameter_index/10)
                ,
                parameter_index
                ),
    CEindex = ifelse(parameter=="CE",
                     parameter_index - floor(parameter_index/10)*10,
                     NA
                     ),
    parlabel = ifelse(parameter=="CE",paste0("CE[",CEindex,"]"),
                      partxt[parameter]
    )
  ) |>
  group_by(parlabel) |>
  arrange(mean) |>
  mutate(
    cumul = (1:n())/n()
  )

popcdf_MU<-tibble(
  MU_r = extract(RDU)$MU[,1],
  MU_lambda = extract(RDU)$MU[,2],
  MU_eta = extract(RDU)$MU[,3],
  MU_psi = extract(RDU)$MU[,4]
  ) |>
  mutate(
    s = 1:n()
  )|>
  expand_grid(
    p = seq(0.01,0.99,length=99)
  ) |>
  pivot_longer(
    cols = MU_r:MU_psi,
    names_to = "parameter",
    values_to = "MU"
  ) |>
  mutate(
    parameter = str_replace(parameter,"MU_","")
  )


popcdf_TAU<-tibble(
  TAU_r = extract(RDU)$TAU[,1],
  TAU_lambda = extract(RDU)$TAU[,2],
  TAU_eta = extract(RDU)$TAU[,3],
  TAU_psi = extract(RDU)$TAU[,4]
  ) |>
  mutate(
    s = 1:n()
  )|>
  expand_grid(
    p = seq(0.01,0.99,length=99)
  ) |>
```

```
  pivot_longer(
    cols = TAU_r:TAU_psi,
    names_to = "parameter",
    values_to = "TAU"
  )|>
  mutate(
    parameter = str_replace(parameter,"TAU_","")
  )

pltlims<-RDUi |>
  group_by(parameter) |>
  summarize(
    min = min(X25.),
    max = max(X75.)
  ) |>
  filter(parameter!="CE")

popcdf<-popcdf_MU |>
  full_join(popcdf_TAU,
            by = c("s","p","parameter")
            ) |>
  full_join(
    pltlims,
    by = "parameter"
  ) |>
  mutate(
    x = ifelse(parameter=="r",min +(max-min)*p,log(min +(max-min)*p)),
    cdf = pnorm((x-MU)/TAU),
    x = ifelse(parameter=="r",x,exp(x))
  ) |>
  group_by(parameter,x) |>
  summarize(
    cdf_LB = quantile(cdf,probs=0.025),
    cdf_UB = quantile(cdf,probs=0.975)
  ) |>
  mutate(
    parlabel = partxt[parameter]
  )

(
  ggplot(RDUi,aes(x=mean))
  +stat_ecdf()
  +geom_errorbar(aes(xmin = X25.,xmax = X75.,y=cumul),color="red",alpha = 0.3)
  +geom_ribbon(data=popcdf,aes(x=x,ymin=cdf_LB,ymax=cdf_UB),alpha = 0.2,fill="blue")
  +facet_wrap(~parlabel,scales="free")
  +theme_bw()
  +xlab("Estimate")
  +ylab("Cumulative density")
)
```

18.5 *R* code used to estimate these models

Script to do the participant-specific estimation:

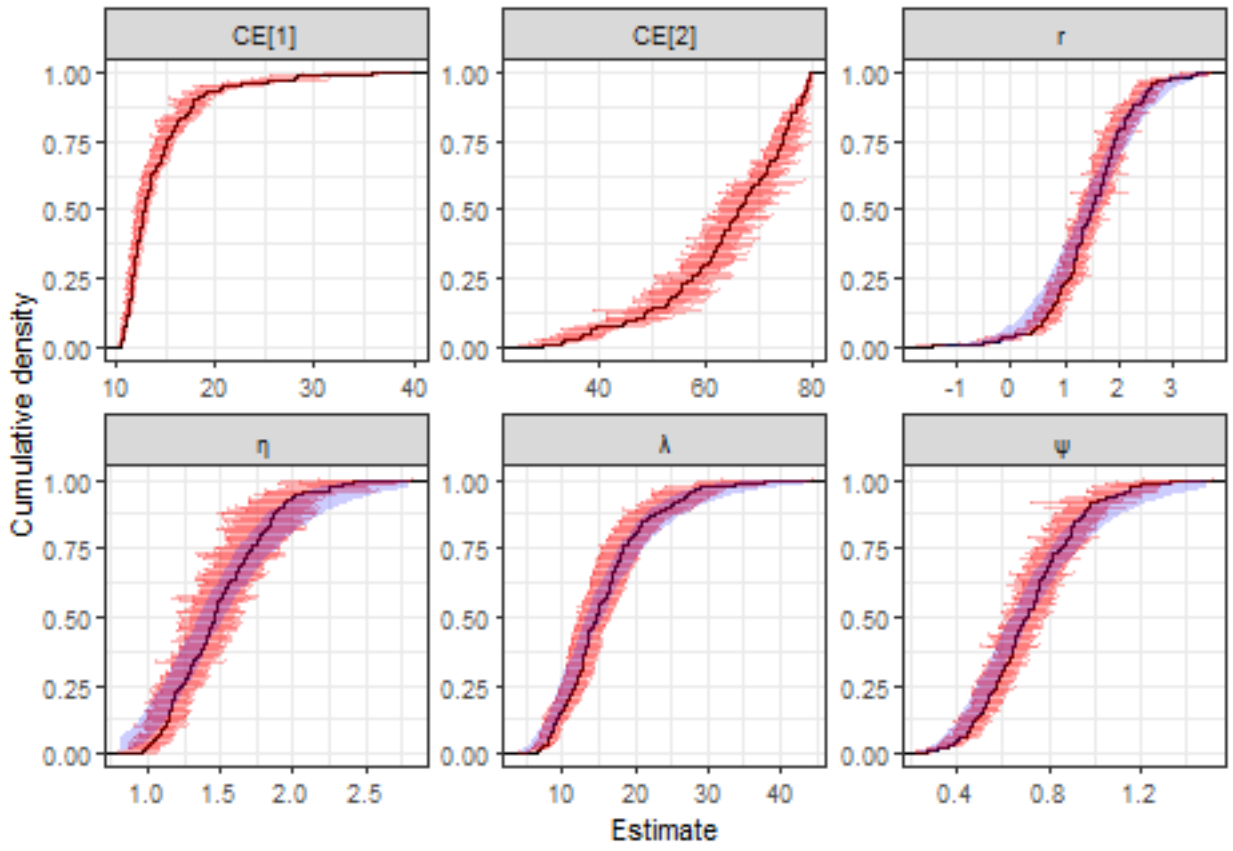


Figure 61: Participant-level parameter estimates from the hierarchical RDU model. The black curve shows the empirical cumulative density of posterior means. Red error bars are a 50% Bayesian credible region (25th-75th percentile). Blue shaded region is a 95% Bayesian credible region (2.5th-97.5th percentile) for the poplation cumulative density function for each parameter (not shown for certainty equivalets).

```
library(tidyverse)
library(rstan)
  rstan_options(auto_write = TRUE)
  options(mc.cores = parallel::detectCores())
  rstan_options(threads_per_chain = 1)

D<-"Code/RiskPreferences/HS2023Cleaned.rds" |>
  readRDS()


models<-list(

  EUT = "Code/RiskPreferences/EUT_individual.stan" |> stan_model(),
  RDU = "Code/RiskPreferences/RDU_individual.stan" |> stan_model()

)

EstimatesSummary<-tibble()


for (ii  in (D$id |> unique())) {

  d<-D |>
    filter(id==ii)

  dStan<-list(

    N = dim(d)[1],
    Left = d$Left,

    pLeft = cbind(d$qL1,d$qL2,d$qL3),
    pRight = cbind(d$qR1,d$qR2,d$qR3),
    prizes = cbind(d$prize1,d$prize2,d$prize3),
    prizerange = cbind(d$prize1,d$prize3),

    prior_r = c(0.27,0.36),
    prior_lambda = c(log(30),0.5),
    prior_psi = c(0,1),
    prior_eta = c(0,1),

    tol = 1e-6,

    nCE = 2,
    CEprizes = list(c(10,45,80),c(10,45,80)),
    CEprobs = list(c(0.95,0,0.05),c(0.05,0,0.95))

  )

  for (mm in names(models)) {

    Fit<-models[[mm]] |>
      sampling(data=dStan,seed=42,
               iter=4000,
```

```
                control=list(adapt_delta=0.999))

    addThis<-summary(Fit)$summary |>
      data.frame()

    addThis$parameter<-rownames(addThis)

    EstimatesSummary<-EstimatesSummary |>
      rbind(
        addThis |>
          mutate(
            id = ii,
            model = mm
          )
      )

  }


  EstimatesSummary |>
    saveRDS("Code/RiskPreferences/Estimates_Individual.rds")

}
```

Script to estimate the hierarchical models

```
library(tidyverse)
library(rstan)
  rstan_options(auto_write = TRUE)
  options(mc.cores = parallel::detectCores())
  rstan_options(threads_per_chain = 1)

D<-"Code/RiskPreferences/HS2023Cleaned.rds" |>
  readRDS() |>
   mutate(id = paste("-",id,"-") |> as.factor() |> as.numeric()) |>
  arrange(id)

idstartend<- D|>
  ungroup() |>
  mutate(rownum = 1:n()) |>
  group_by(id) |>
  summarize(
    start = min(rownum),
    end = max(rownum)
  )

ReRunRDU<-FALSE
ReRunEUT<-FALSE
ReRunRDU_t<-TRUE

# Hierarchical RDU model with student t distribution

file <-"Code/RiskPreferences/FitRDU_t.rds"
```

```r
if (!file.exists(file) | ReRunRDU_t) {

  print(paste("Estimating RDU hierarchical model with Student t distribution. Output will go into: ",fi


  RDU_t <- "Code/RiskPreferences/RDU_hierarchical_student_t.stan" |> stan_model()


  dStan<-list(

    N = dim(D)[1],
    Left = D$Left,

    id = D$id,

    nparticipants = dim(idstartend)[1],

    idstartend = idstartend |> select(start,end),

    pLeft = cbind(D$qL1,D$qL2,D$qL3),
    pRight = cbind(D$qR1,D$qR2,D$qR3),
    prizes = cbind(D$prize1,D$prize2,D$prize3),
    prizerange = cbind(D$prize1,D$prize3),

    prior_MU<-list(
      c(0.27,0.36),
      c(3.45,0.59),
      c(0,1),
      c(0,1)

    ),

    prior_TAU = c(1,1,1,1),
    prior_OMEGA = 4,
    prior_nu = c(15,7.5),

    tol = 1e-6,

    nCE = 2,
    CEprizes = list(c(10,45,80),c(10,45,80)),
    CEprobs = list(c(0.95,0,0.05),c(0.05,0,0.95))

  )

  FitRDU_t<-RDU_t |>
    sampling(
      data = dStan,seed=43,
      pars="theta",include=FALSE,
      iter=20000,
      control = list(adapt_delta=0.99999)
    )

  FitRDU_t |>
```

```
    saveRDS(file)

} else {
  print(paste("skipping RDU_t,",file,"already exists"))
}


## RDU hierarchical model with multivariate normal distribution

file <-"Code/RiskPreferences/FitRDU.rds"

if (!file.exists(file) | ReRunRDU) {

  print(paste("Estimating RDU hierarchical model. Output will go into: ",file))


  RDU <- "Code/RiskPreferences/RDU_hierarchical.stan" |> stan_model()


  dStan<-list(

      N = dim(D)[1],
      Left = D$Left,

      id = D$id,

      nparticipants = dim(idstartend)[1],

      idstartend = idstartend |> select(start,end),

      pLeft = cbind(D$qL1,D$qL2,D$qL3),
      pRight = cbind(D$qR1,D$qR2,D$qR3),
      prizes = cbind(D$prize1,D$prize2,D$prize3),
      prizerange = cbind(D$prize1,D$prize3),

      prior_MU<-list(
        c(0.27,0.36),
        c(3.45,0.59),
        c(0,1),
        c(0,1)

      ),

      prior_TAU = c(1,1,1,1),
      prior_OMEGA = 4,

      tol = 1e-6,

      nCE = 2,
      CEprizes = list(c(10,45,80),c(10,45,80)),
      CEprobs = list(c(0.95,0,0.05),c(0.05,0,0.95))

  )
```

```
  FitRDU<-RDU |>
    sampling(
      data = dStan,seed=43,
      pars="z",include=FALSE
    )

  FitRDU |>
    saveRDS(file)

} else {
  print(paste("skipping RDU,",file,"already exists"))
}

file <-"Code/RiskPreferences/FitEUT.rds"

if (!file.exists(file) | ReRunEUT) {

  print(paste("Estimating EUT hierarchical model. Output will go into: ",file))


  EUT <- "Code/RiskPreferences/EUT_hierarchical.stan" |> stan_model()


  dStan<-list(

    N = dim(D)[1],
    Left = D$Left,

    id = D$id,

    nparticipants = dim(idstartend)[1],

    idstartend = idstartend |> select(start,end),

    pLeft = cbind(D$qL1,D$qL2,D$qL3),
    pRight = cbind(D$qR1,D$qR2,D$qR3),
    prizes = cbind(D$prize1,D$prize2,D$prize3),
    prizerange = cbind(D$prize1,D$prize3),

    prior_MU<-list(
      c(0.27,0.36),
      c(3.45,0.59)

    ),

    prior_TAU = c(1,1),
    prior_OMEGA = 4,

    tol = 1e-6

  )

  FitEUT<-EUT |>
```

```
    sampling(
      data = dStan,seed=43,
      pars = "z",include=FALSE
    )

  FitEUT |>
    saveRDS(file)

} else {
  print(paste("skipping EUT,",file,"already exists"))
}
```

19 Application: Meta-analysis using (some of) the METARET data

Sometimes, you have more than one dataset and you want to make inferences about how similar or dissimular these datasets are, or how much you can learn from the data as a whole. This is where you want to do a *meta-analysis*, and Bayesian techniques are particularly well-suited to these, for exactly the same reason that they handle hierarchical models well. While the hierarchical models presented up to this point has two levels of parameters (i.e. population-level and participant-level), we will now have three or more levels, as there could be meaningful variation at the population level, between experiments, and between participants within experiments. Quantifying this variation is useful, and some small additions to our understanding of hierarchical models will help us understand this.

Hopefully you will see that adding this additional level of variation is not much of a leap forward as far as hierarchical models go. We now simply have (i) parameters describing the distribution between experiments. These feed into (ii) parameters describing the distribution *within* experiments, and (iii) participant-specific parameters.

I suspect and hope that this kind of analysis is going to become *very* common in Experimental Economics. This is because we have had a long history of good data-sharing norms, and it is therefore not too difficult to obtain data from published experiments. Furthermore, as experimental economists often design new experiments by using older experiments as a baseline, there are a lot of opportunities for the same (or similar enough) designs to show up a lot more than once.

19.1 Data

Here we will be using a subset of the METARET datset, which is a collection of data on risk elicitation tasks.[88] I will focus just on the subset of this dataset corresponding to the Holt and Laury (2002) risk elicitation task. Table 43 summarizes this slice of the data. as you can see, there are a lot of experiments, and a lot of participants!

```
D<-"Code/METARET/Data/data.csv" |>
  read.csv() |>
  filter(task=="HL")|>
  mutate(exptID = bibkey |> as.factor())

exptList<-D |>
  group_by(exptID) |>
  summarize(
    experiment = bibkey[1],
    id = exptID[1] |> as.numeric(),
```

[88] I downloaded this dataset on 2024-09-12. As METARET is an ongoing project, you may get different estimates to me.

```
    participants = n()
  )

exptList |>
  dplyr::select(
    experiment,participants
  ) |>
  kbl(caption = paste("The ",dim(exptList)[1],"experiments using the Holt and Laury (2002) task in the l
  kable_classic(full_width=FALSE)
```

Table 44 shows you a random sample of the data we will be using. The variable `bibkey` will identify the experiment. Our outcome of interest will be the variable `r`, wich is an estimate of the coefficient of relative risk aversion. However those of you familiar with Holt and Laury (2002) will know that this task only gives you an *interval* range for r. This interval is a function of the variable `choice`, which is the switch-point observed for a participant in the task. In corresponding with Paolo Crosetto, I found out that the `r` variable was uniformly assigned within this range (you can also see this in the METARET replication files). We can see that these intervals are exactly the same by looking at the empirical cumulative density function of the `r` variable, which I show in Figure 62:

```
(
  ggplot(D,aes(x=r,color=as.factor(choice)))
  +stat_ecdf()
  +theme_bw()
)
```

For simplicity we will begin by treating `r` as if it is the exact value of r for this participant, but then we will relax this assumption so that it is interval-valued.

```
set.seed(42)
D |>
  sample_n(20) |>
  dplyr::select(bibkey,choice,r,task) |>
  kbl(caption = "A random sample of the METARET Holt and Laury (2002) data.") |>
  kable_classic(full_width=FALSE)
```

19.2 A basic model

To begin with, assume that the `r` in the dataset is measured perfectly, and so we do not have to worry about it really being an interval. Let $r_{i,e}$ be the risk-aversion value measured for participant i in experiment e. *Within* each experiment e, assume that $r_{i,e} \sim N(\mu_e, \sigma^2)$, where μ_e is an experiment-specific mean of $r_{i,e}$ and σ is common to all experiments (we will relax this later). *Between* experiments, we will assume that the experiment means μ_is are normally distributed, $\mu_i \sim N(\bar{\mu}, \bar{\sigma}^2)$. That is, we have between-experiment variation measured by $\bar{\sigma}$, and within-experiment variation measured by σ.

As we might be interested in the relative importance of these sources of variation, it is useful to decompose the total variance of an observation by looking at the ratio of between-experiment variance to within-experiment variance, which is:

$$\rho = \frac{\bar{\sigma}^2}{\bar{\sigma}^2 + \sigma^2}$$

That is if ρ is close to zero, then $\bar{\sigma}^2$ is relatively small. This will mean that there is not much difference in the means of $r_{e,i}$ between experiments. On the other hand, if ρ is close to one, then most of the variation in $r_{e,i}$ is attributable to variation between experiments. In this case, it would be hard to predict the mean of $r_{e,i}$ in a new experiment.

Table 43: The 52 experiments using the Holt and Laury (2002) task in the METARET dataset

experiment	participants
Abdellaoui2011	36
Andersen2010	89
Barrera2012	98
Bauernschuster2010	174
Bellemare2010	84
Branas2011	145
Carlsson2009	213
Casari2009	78
Chakravarty2011	37
Charness2019	157
Chen2013	72
Cobo-Reyes2012	76
Crosetto2016	73
Dave2010	802
Deck2012a	47
Dickinson2009	126
Drichoutis2012	57
Duersch2012	200
Eckel2006	198
Fiedler2012	29
Fiore2009	40
Frey2017	1331
Gloeckner2012	159
Gloeckner2012a	38
Grijalva2011	77
Harrison2005	152
Harrison2007b	18
Harrison2008	100
harrison2012b	90
Holt2002	199
Holzmeister2019	198
Jacquemet2008	87
jamison2008	130
Lange2007	121
Lange2007a	172
Levy-Garboua2012	54
Lusk2005	47
Masclet2009	79
Mueller2012	116
Nieken2012	287
Pogrebna2011	57
Ponti2009	33
Rosaz2012	112
Rosaz2012a	279
Ryvkin2011	42
Schram2011	137
Shafran2010	64
Slonim2010	116
Sloof2010	86
Szrek2012	198
Wakolbinger2009	131
Yechiam2012	11

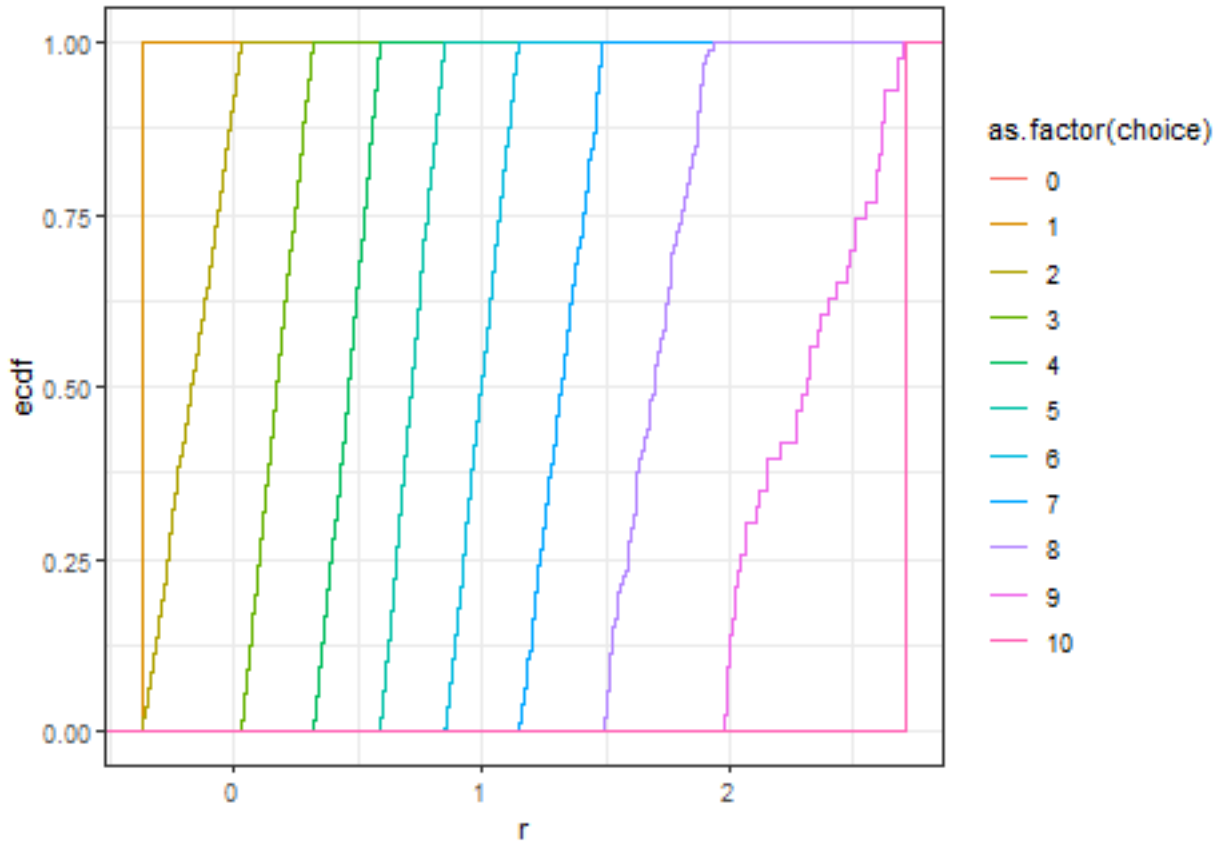


Figure 62: Empirical cumulative density functions of r, broken down by choice in the Holt and Laury (2002) task

Table 44: A random sample of the METARET Holt and Laury (2002) data.

bibkey	choice	r	task
harrison2012b	5	0.7974468	HL
Duersch2012	6	1.1191757	HL
Lange2007a	5	0.6087734	HL
Grijalva2011	6	1.1030496	HL
Szrek2012	5	0.6409813	HL
Jacquemet2008	6	0.9068044	HL
Frey2017	3	0.0869766	HL
Shafran2010	7	1.3751412	HL
Gloeckner2012	2	-0.2327934	HL
Grijalva2011	8	1.4956184	HL
Andersen2010	4	0.5473514	HL
Branas2011	5	0.8301501	HL
Bauernschuster2010	4	0.4462529	HL
Holzmeister2019	3	0.0638232	HL
Casari2009	5	0.7920361	HL
Carlsson2009	1	-0.3700000	HL
Frey2017	2	-0.2673258	HL
Frey2017	5	0.7556473	HL
Duersch2012	4	0.3521928	HL
Wakolbinger2009	7	1.3328647	HL

Here is my formal statement of the model's structure and its priors:

$$
\begin{aligned}
\text{Participant-level variation: } & r_{i,e} \sim N(\mu_e, \sigma^2) \\
\text{Experiment-level variation: } & \mu_e \sim N(\bar{\mu}, \bar{\sigma}^2) \\
\text{priors: } & \bar{\mu} \sim N(0, 10) \\
& \bar{\sigma} \sim \text{Cauchy}^{+}(0, 10) \\
& \sigma \sim \text{Cauchy}^{+}(0, 10)
\end{aligned}
$$

And here is the *Stan* program I wrote to estimate it. Note that we are augmenting the data by the μ_is. Therefore we will get shrinkage estimates of these from the estimation process.

```
// 2level.stan
// a 2-level hierarchical model for aggregating risk preferences
data {
  int<lower=0> N;
  vector[N] r;

  int nexpt;
  int exptID[N];

  vector[2] prior_MU;
  real prior_sigma_expt;
  real prior_sigma_i;
}
parameters {
  real MU;
```

```
  real<lower=0> sigma_expt;
  real<lower=0> sigma_i;

  vector[nexpt] mu_expt;
}

model {

  // prior

  target += normal_lpdf(MU | prior_MU[1],prior_MU[2]);
  target += cauchy_lpdf(sigma_expt | 0.0,prior_sigma_expt);
  target += cauchy_lpdf(sigma_i | 0.0,prior_sigma_i);

  // hierarchical structure

  target+= normal_lpdf(mu_expt | MU,sigma_expt);

  // likelihood

  target += normal_lpdf(r | mu_expt[exptID], sigma_i);

}

generated quantities {

  real rho = pow(sigma_expt,2.0)/(pow(sigma_expt,2.0)+pow(sigma_i,2.0));
}
```

Table 45 shows the estimates from this model. In particular, note the estimate for ρ, which is much closer to zero than one. This suggests that within-experiment variation in $r_{e,i}$ is larger than between-experiment variation in its means.

```
Fit<-summary(
  "Code/METARET/Fit.rds" |>
  readRDS()
  )$summary |>
  data.frame()|>
  rownames_to_column(var="parameter") |>
  mutate(
    exptID = parse_number(parameter)
  )

Fit |>
  filter(is.na(exptID)) |>
  dplyr::select(-exptID) |>
  kbl(digits=3,caption = "Parameter estimates from the basic model") |>
  kable_classic(full_width=FALSE)
```

19.3 But the data are really interval-valued!

Now we will take the interval-valued property of Holt and Laury (2002) data more seriously. Once we code up the bounds (you can see how I do that at the end of this chapter), it is really easy to augment our data again by the unobserved (i.e. latent) $r^*_{i,i}$, which we know will be somewhere between a lower bound of $\underline{r}_{e,i}$,

Table 45: Parameter estimates from the basic model

parameter	mean	se_mean	sd	X2.5.	X25.	X50.	X75.	X97.5.	n_eff	Rh
MU	0.584	0.00	0.030	0.524	0.564	0.584	0.604	0.643	5618.174	1.0
sigma_expt	0.207	0.00	0.025	0.165	0.189	0.206	0.223	0.261	4218.509	1.0
sigma_i	0.550	0.00	0.005	0.541	0.547	0.550	0.553	0.559	7046.200	1.0
rho	0.125	0.00	0.026	0.082	0.106	0.122	0.141	0.184	4256.043	1.0
lp___	-6203.000	0.13	5.223	-6214.325	-6206.201	-6202.631	-6199.302	-6193.737	1611.583	1.0

and an upper bound of $\overline{r}_{e,i}$. Here is the modification to the model statement we need to make:

$$
\begin{aligned}
\text{Participant-level variation: } & r^*_{i,e} \sim \text{Truncated Normal}(\mu_e, \sigma^2, (\underline{r}_{i,e}, \overline{r}_{i,e})) \\
\text{Experiment-level variation: } & \mu_e \sim N(\bar{\mu}, \bar{\sigma}^2) \\
\text{priors: } & \bar{\mu} \sim N(0, 10) \\
& \bar{\sigma} \sim \text{Cauchy}^+(0, 10) \\
& \sigma \sim \text{Cauchy}^+(0, 10)
\end{aligned}
$$

And here is the *Stan* program that estimates it. Note now that `r` is a parameter, rather than data.

```
// 2level_interval.stan
// a 2-level hierarchical model for aggregating risk preferences
// treating choices as revealing intervals
data {
  int<lower=0> N; // number of observations
  vector[N] rlb; // lower bound for r
  vector[N] rub; // upper bound of r

  int nexpt; // number of experiments
  int exptID[N]; // experiment id

  vector[2] prior_MU; // mean of r in population
  real prior_sigma_expt; // sd of r at experiment level
  real prior_sigma_i; // sd of r at individual level
}
parameters {
  real MU;

  real<lower=0> sigma_expt;
  real<lower=0> sigma_i;

  vector[nexpt] mu_expt;

  vector<lower=rlb,upper=rub>[N] r;
}

model {

  // prior

  target += normal_lpdf(MU | prior_MU[1],prior_MU[2]);
  target += cauchy_lpdf(sigma_expt | 0.0,prior_sigma_expt);
  target += cauchy_lpdf(sigma_i | 0.0,prior_sigma_i);
```

Table 46: Parameter estimates from the interval-valued model

parameter	mean	se_mean	sd	X2.5.	X25.	X50.	X75.	X97.5.	n_e
MU	0.567	0.001	0.034	0.501	0.544	0.568	0.591	0.632	4096.2
sigma_expt	0.230	0.000	0.027	0.183	0.211	0.228	0.247	0.290	3068.3
sigma_i	0.590	0.000	0.005	0.580	0.587	0.590	0.594	0.601	5038.2
rho	0.133	0.000	0.027	0.087	0.113	0.130	0.149	0.193	3050.1
lp__	-30049.240	2.011	73.787	-30196.315	-30099.434	-30049.547	-29998.176	-29905.889	1346.3

```
  // hierarchical structure

  target+= normal_lpdf(mu_expt | MU,sigma_expt);

  // likelihood

  target += normal_lpdf(r | mu_expt[exptID], sigma_i);

}

generated quantities {

  real rho = pow(sigma_expt,2.0)/(pow(sigma_expt,2.0)+pow(sigma_i,2.0));
}
```

Table 46 shows the estimates from this model. These are not substantially different from those in Table 45, but we didn't know that until estimating this model.

```
Fit_interval<-summary(
  "Code/METARET/Fit_interval.rds" |>
  readRDS()
  )$summary |>
  data.frame()|>
  rownames_to_column(var="parameter") |>
  mutate(
    exptID = parse_number(parameter)
  )

Fit_interval |>
  filter(is.na(exptID)) |>
  dplyr::select(-exptID) |>
  kbl(digits=3,caption = "Parameter estimates from the interval-valued model") |>
  kable_classic(full_width=FALSE)
```

19.4 Heterogeneous standard deviations

Of course, these experiments could be different not just because they have different means of risk-aversion, but also because they have different standard deviations of risk-aversion. In order to investigate how and whether this affects our conclusions, here is a model that assumes a hierarchical structure for σ_e as well as μ_e:

$$
\begin{aligned}
\text{Participant-level variation: } r^*_{i,e} &\sim \text{Truncated Normal}(\mu_e, \sigma_e^2, (\underline{r}_{i,e}, \overline{r}_{i,e})) \\
\text{Experiment-level variation: } \mu_e &\sim N(\bar{\mu}, \bar{\sigma}^2) \\
\log \sigma_e &\sim N(M, S^2) \\
\text{priors: } \bar{\mu} &\sim N(0, 10) \\
\bar{\sigma} &\sim \text{Cauchy}^+(0, 10) \\
M &\sim N(0, 10) \\
S &\sim \text{Cauchy}^+(0, 10)
\end{aligned}
$$

And here is the the *Stan* file that implements it. Note that we are augmenting the data by σ_e *and* μ_e now. Also, since we don't have *one* ρ anymore, I am calculating an *average* ρ in the `generated quantities` block using Monte Carlo integration.

```
// 2level_interval_sd.stan
// a 2-level hierarchical model for aggregating risk preferences
// treating choices as revealing intervals
data {
  int<lower=0> N; // number of observations
  vector[N] rlb; // lower bound for r
  vector[N] rub; // upper bound of r

  int nexpt; // number of experiments
  int exptID[N]; // experiment id

  vector[2] prior_MU; // mean of r in population
  real prior_sigma_expt; // sd of r at experiment level
  vector[2] prior_sigma_i_M;  // m and sd of log-median sigma_i
  real<lower=0> prior_sigma_i_SD; // sd for sigma_i

  int nsim;
}
parameters {
  real MU;

  real<lower=0> sigma_expt;

  real sigma_i_M;
  real<lower=0> sigma_i_SD;


  // augmentted data
  vector<lower=0>[nexpt]sigma_i;
  vector[nexpt] mu_expt;
  vector<lower=rlb,upper=rub>[N] r;
}

model {

  // prior

  target += normal_lpdf(MU | prior_MU[1],prior_MU[2]);
  target += cauchy_lpdf(sigma_expt | 0.0,prior_sigma_expt);
  target += normal_lpdf(sigma_i_M | prior_sigma_i_M[1],prior_sigma_i_M[2]);
```

```
  target += cauchy_lpdf(sigma_i_SD | 0.0, prior_sigma_i_SD);


  // hierarchical structure

  target+= normal_lpdf(mu_expt | MU,sigma_expt);
  target+= lognormal_lpdf(sigma_i | sigma_i_M, sigma_i_SD);

  // likelihood

  target += normal_lpdf(r | mu_expt[exptID], sigma_i[exptID]);

}

generated quantities {

  real rho;

  {
    vector[nsim] sim_sigma_i =exp(sigma_i_M+sigma_i_SD*to_vector(normal_rng(rep_vector(0.0,nsim),rep_ve

    rho = mean(pow(sigma_expt,2.0)/(pow(sigma_expt,2.0)+pow(sim_sigma_i,2.0)));
  }
}
```

Table 47 shows the estimates of the non-augmented parameters of the model. We don't see much of a difference in what we can compare to the previous model. However note that `sigma_i_sd` is substantially away from zero, indicating a good amount of variation in the experiment-specific standard deviations.

```
Fit_interval_sd<-summary(
  "Code/METARET/Fit_interval_sd.rds" |>
  readRDS()
  )$summary |>
  data.frame()|>
  rownames_to_column(var="parameter") |>
  mutate(
    exptID = parse_number(parameter)
  )

Fit_interval_sd |>
  filter(is.na(exptID)) |>
  dplyr::select(-exptID) |>
  kbl(digits=3,caption = "Parameter estimates from the model assuming heterogeneous within-experiment s
  kable_classic(full_width=FALSE)
```

Now that we have *two* sets of augmented parameters, μ_e and σ_e, we can visualize their relationship in a plot, which I do in Figure 63. These are the "shrinkage" estimates of these parameters, in that they are informed both by the values of $r_{i,e}$ within their respective experiments, and also by the population distributions that are jointly estimated alongside them.

```
d<-Fit_interval_sd |>
  filter(!is.na(exptID)) |>
  mutate(parameter = gsub("[^a-zA-Z]", "", parameter)) |>
  pivot_wider(
    id_cols = exptID,
```

Table 47: Parameter estimates from the model assuming heterogeneous within-experiment standard deviations

parameter	mean	se_mean	sd	X2.5.	X25.	X50.	X75.	X97.5.	n_
MU	0.550	0.000	0.028	0.496	0.531	0.550	0.570	0.606	3834.9
sigma_expt	0.186	0.000	0.024	0.144	0.169	0.184	0.201	0.239	2679.9
sigma_i_M	-0.521	0.001	0.044	-0.608	-0.551	-0.521	-0.490	-0.436	4478.5
sigma_i_SD	0.316	0.001	0.037	0.253	0.290	0.313	0.339	0.396	3509.7
rho	0.104	0.000	0.025	0.064	0.086	0.101	0.118	0.159	2969.9
lp___	-29589.388	2.016	73.080	-29738.524	-29637.172	-29587.289	-29540.121	-29451.549	1314.4

```
    names_from = "parameter",
    values_from = c("mean","X25.","X75.")
  ) |>
  mutate(id = exptID |> as.numeric()) |>
  full_join(
    exptList,
    by = "id"
  )

(
  ggplot(d,aes(x=mean_muexpt,y=mean_sigmai,label=experiment))
  #+geom_text(size=2)
  +geom_point(aes(size=participants))
  +geom_errorbar(aes(xmin = X25._muexpt,xmax=X75._muexpt),alpha=0.3)
  +geom_errorbar(aes(ymin = X25._sigmai,ymax=X75._sigmai),alpha=0.3)
  +xlab(expression(mu[e]~": mean of risk-aversion within experiemnt"))+ylab(expression(sigma[e]~": stand
  +theme_bw()
)
```

19.5 Student-t distributions, because why not?

One assumption we have maintained here is that the experiment- and participant-level distributions are both normals. While this is a great place to start, but if we suspect that this is a bad assumption then we might want to take a more flexible model to our data. A simple one-parameter relaxation of the normal distribution is Student's t distribution. The new "degrees of freedom" parameter $\nu > 0$ measures how closely this distribution matches the normal: as $\nu \to \infty$ the t approaches the normal, and $\nu < 30$ is a good rule of thumb for the range of ν where the distributions differ appreciably.

Here I will assume that both the experiment-specific μ_is and the participant-specific $r_{e,i}$s have t distributions. Here is the new model:

$$
\begin{aligned}
\text{Participant-level variation: } r^*_{i,e} &\sim \text{Truncated Student t}(\nu, \mu_e, \sigma_e^2, (\underline{r}_{i,e}, \overline{r}_{i,e})) \\
\text{Experiment-level variation: } \mu_e &\sim \text{Student t}(\bar{\nu}, \bar{\mu}, \bar{\sigma}^2) \\
\log \sigma_e &\sim N(M, S^2) \\
\text{priors: } \bar{\mu} &\sim N(0, 10) \\
\bar{\sigma} &\sim \text{Cauchy}^+(0, 10) \\
M &\sim N(0, 10) \\
S &\sim \text{Cauchy}^+(0, 10)
\end{aligned}
$$

Note that I have kept the experiment-specific augmented σ_es, but for simplicity I will not be allowing ν to vary between experiments.

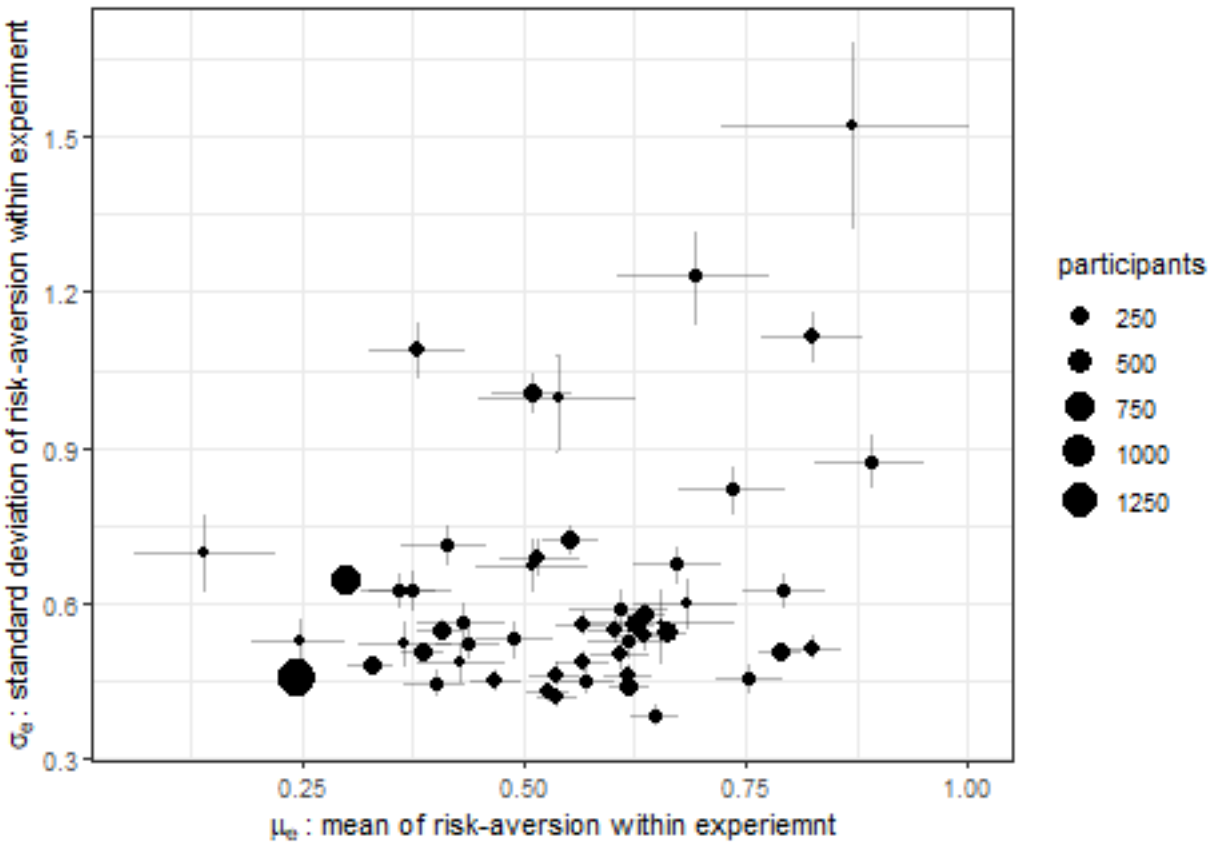


Figure 63: Experiment-level estimates of mean and standard deviation of risk-aversion. Error bars show 50% Bayesian credible regions

We need to be especially careful here when calculating ρ, the fraction of the variance of $r_{e,i}$ that is due to between-experiment variation. This is because the variance of the t-distribution is not equal to the parameter σ. In fact, it is:

$$V(t(\nu,\mu,\sigma)) = \begin{cases} \frac{\nu}{\nu-2}\sigma^2 & \text{if } \nu > 2 \\ \text{undefined} & \text{otherwise} \end{cases}$$

I will therefore not comment on ρ for this estimation.

You can see this in the `generated quantities` block at the bottom of this *Stan* program:

```
// 2level_interval_sd_t.stan
// a 2-level hierarchical model for aggregating risk preferences
// treating choices as revealing intervals
data {
  int<lower=0> N; // number of observations
  vector[N] rlb; // lower bound for r
  vector[N] rub; // upper bound of r

  int nexpt; // number of experiments
  int exptID[N]; // experiment id

  vector[2] prior_MU; // mean of r in population
  real prior_sigma_expt; // sd of r at experiment level
  vector[2] prior_sigma_i_M;  // m and sd of log-median sigma_i
  real<lower=0> prior_sigma_i_SD; // sd for sigma_i

  real<lower=0> prior_nu_expt; // prior for nu at expt level
  real<lower=0> prior_nu_i; // prior for nu at individual level

}
parameters {
  real MU;

  real<lower=0> sigma_expt;

  real sigma_i_M;
  real<lower=0> sigma_i_SD;

  real<lower=0> nu_i;
  real<lower=0> nu_expt;


  // augmentted data
  vector<lower=0>[nexpt]sigma_i;
  vector[nexpt] mu_expt;
  vector<lower=rlb,upper=rub>[N] r;
}

model {

  // prior

  target += normal_lpdf(MU | prior_MU[1],prior_MU[2]);
```

Table 48: Parameter estimates from the model assuming heterogeneous within-experiment standard deviations

parameter	mean	se_mean	sd	X2.5.	X25.	X50.	X75.	X97.5.	n
MU	0.535	0.000	0.024	0.488	0.519	0.535	0.552	0.584	5507
sigma_expt	0.156	0.000	0.022	0.116	0.140	0.154	0.170	0.203	4285
sigma_i_M	-0.885	0.001	0.047	-0.978	-0.916	-0.884	-0.854	-0.790	4005
sigma_i_SD	0.298	0.001	0.038	0.231	0.272	0.296	0.321	0.383	3427
nu_i	2.927	0.004	0.150	2.650	2.822	2.921	3.025	3.232	1623
nu_expt	55.167	4.231	236.743	2.936	8.234	15.859	36.389	367.124	3131
lp___	-29308.651	2.077	72.261	-29452.428	-29357.689	-29306.838	-29258.540	-29170.925	1210

```
  target += cauchy_lpdf(sigma_expt | 0.0,prior_sigma_expt);
  target += normal_lpdf(sigma_i_M | prior_sigma_i_M[1],prior_sigma_i_M[2]);
  target += cauchy_lpdf(sigma_i_SD | 0.0, prior_sigma_i_SD);
  target += cauchy_lpdf(nu_i| 0.0, prior_nu_i);
  target += cauchy_lpdf(nu_expt|0.0,prior_nu_expt);


  // hierarchical structure

  target+= student_t_lpdf(mu_expt |nu_expt, MU,sigma_expt);
  target+= lognormal_lpdf(sigma_i | sigma_i_M, sigma_i_SD);

  // likelihood

  target += student_t_lpdf(r |nu_i, mu_expt[exptID], sigma_i[exptID]);

}
```

Table 48 shows estimates from this model. Of particular interest are the new parameters $\bar{\nu}$ (`nu_expt`) and ν (`nu_i`). Recall that $\nu < 30$ suggests that a t distribution departs meaningfully from a normal, and we see this for the median estimates of both νs (although the posterior *mean* estimate of $\bar{nu}$ is well above 30). I would conclude from this that the normal assumption probably isn't a good one to make with these data, both at the experiment level (i.e. μ_i is not normally distributed), and at the participant level (i.e. $r_{i,e}$ is not normally distributed within an experiment)

```
Fit_interval_sd_t<-summary(
  "Code/METARET/Fit_interval_sd_t.rds" |>
  readRDS()
  )$summary |>
  data.frame()|>
  rownames_to_column(var="parameter") |>
  mutate(
    exptID = parse_number(parameter)
  )

Fit_interval_sd_t |>
  filter(is.na(exptID)) |>
  dplyr::select(-exptID) |>
  kbl(digits=3,caption = "Parameter estimates from the model assuming heterogeneous within-experiment s
  kable_classic(full_width=FALSE)
```

19.6 *R* code to estimate the models

```
library(tidyverse)
library(rstan)
rstan_options(auto_write = TRUE)
options(mc.cores = parallel::detectCores())

D<-"Code/METARET/Data/data.csv" |>
  read.csv() |>
  filter(task=="HL")|>
  mutate(exptID = bibkey |> as.factor()) |>
  group_by(choice) |>
  mutate(
    rlb = min(r),
    rub = max(r)
  )|>
ungroup() |>
  mutate(
    rlb = ifelse(choice<=1,-Inf,rlb),
    rub = ifelse(choice==10,Inf,rub)
  )


D$bibkey |> unique()




dStan<-list(
  N = dim(D)[1],
  r = D$r,
  rlb =D$rlb,
  rub = D$rub,

  nexpt = D$exptID |> unique() |> length(),
  exptID = D$exptID |> as.numeric(),

  prior_MU = c(0,10),
  prior_sigma_expt = 10,
  prior_sigma_i = 10,
  prior_sigma_i_M = c(0,10),
  prior_sigma_i_SD = 10,
  prior_nu_expt = 10,
  prior_nu_i = 10,

  nsim = 1000
)

model_interval_sd_t<-"Code/METARET/2level_interval_sd_t.stan" |>
  stan_model()

Fit_interval_sd_t <- model_interval_sd_t |>
  sampling(
    data=dStan,seed=42,
```

```
    iter = 2000
    ,
    par = "r",include=FALSE
  )

Fit_interval_sd_t |>
  saveRDS(
    "Code/METARET/Fit_interval_sd_t.rds"
  )


model_interval_sd<-"Code/METARET/2level_interval_sd.stan" |>
  stan_model()

Fit_interval_sd <- model_interval_sd |>
  sampling(
    data=dStan,seed=42,
    iter = 2000,
    par = "r",include=FALSE
  )

Fit_interval_sd |>
  saveRDS(
    "Code/METARET/Fit_interval_sd.rds"
  )

model<-"Code/METARET/2level.stan" |>
  stan_model()

Fit<-model |>
  sampling(data=dStan,seed=42,
           iter = 2000
  )



Fit |>
  saveRDS("Code/METARET/Fit.rds")

model_interval<-"Code/METARET/2level_interval.stan" |>
  stan_model()


Fit_interval<-model_interval |>
  sampling(data=dStan,seed=42,
           iter = 2000,
           par = "r",include=FALSE
  )

Fit_interval |>
  saveRDS("Code/METARET/Fit_interval.rds")
```

20 Application: choice bracketing

This topic evokes some strong (mostly good) feelings for me, as I wrote my job market paper on it (Bland 2019a). Choice bracketing is the idea that people might partition (or "bracket") their decisions into more than one group. When I was on the job market, I would use the analogy of choosing what to eat and drink at a restaurant. The (unboundedly) rational economic agent who brackets *broadly* picks their favorite *combination* of food and drink. However this may be a difficult problem if the food and drink menus are large. Instead, the decision-maker might look only at the food menu and pick their favorite food, then look only at the drink menu and pick their favorite drink. This behavior is called *narrow bracketing*, and can lead to mistakes relative to the rational model because the decision-maker ignores that food and drink can be complements. For example, a narrowly bracketing decision-maker runs the risk at a restaurant of choosing red wine with fish, which is often regarded as a poor pairing.[89]

While we as humans usually pay some attention to the combinations of food and drink we are consuming, we may be more likely to narrowly bracket in other domains. One domain that has received a lot of attention in the experimental/behavioral economics and psychology literature is choice under risk (see for example Rabin and Weizsäcker (2009)). In this environment "mistakes" are sometimes easier to recognize because they can be identified by (e.g. first-order stochastically) dominated choices. However Ellis and Freeman (2024) argue that the existing literature typically provides evidence against broad bracketing, rather than in favor of narrow bracketing. That is, the broad bracketing model does not organize the data well, but the narrow bracketing model does not make any testable predictions for existing experiments. It is this experiment of Ellis and Freeman (2024) that I use as the example dataset in this chapter.

20.1 Data and model

For this application, I use data from the social preferences experiment in Ellis and Freeman (2024). In this experiment, made five decisions that involved allocating tokens to two other people. In decisions 2, 4, and 5, this was a simple convex budget set problem of allocating I tokens between person A and person B. Tokens were worth $\$v_A$ to person A and $\$v_B$ to person B. Figure 64 shows these decisions along with dicisions 1 and 3, which I will get to later. For decisions 2, 4, and 5, the red line shows the budget line, and the red dot shows the choice made by participant 6 in the experiment. Here we can see that in decision 4 participants faced a choice where tokens were equally valuable to person A and person B. In this instance participant 6 chose to split them equally. This is also approximately true for decision 2 (where tokens were also equally valuable). On the other hand in decision 5, tokens are worth more to person B than person A, and participant 6 chose to give all of the tokens to person B.

```
lookup<-rbind(
  # decision  part  I vA  vB
  c(1,1,10,1,1.20),
  c(1,2,16,1,1),
  c(2,1,14,2,2),
  c(3,1,10,1,1),
  c(3,2,10,1,1.20),
  c(4,1,16,1,1),
  c(5,1,10,1,1.20)
) |>
  data.frame()
colnames(lookup)<-c("decision","part","I","vA","vB")

D<-"Data/EF2024/SocialExperimentData.csv" |>
  read.csv()|>
  mutate(
```

[89] While I was on the job market, after giving this analogy, a former professor of mine interrupted my talk to very loudly proclaim: "James, red wine is *never* a mistake!" Another former professor later volunteered that while he was on the job market, he ended up ordering fish and Coca-Cola, which for him turned out to be a particularly bad pairing.

```
    uid = paste(Location,Session,SubjectID) |> as.factor() |> as.numeric()
  ) |>
  pivot_longer(
    cols = R1A1.A:R5A1.B,
    names_to = "name",
    values_to = "allocation"
  )   |>
      mutate(
        decision = substr(name,2,2) |> as.integer(),
        part =  substr(name,4,4) |> as.integer(),
        player = substr(name,6,6)
  )|>
  filter(player=="A")|>
  full_join(lookup,by=c("decision","part")) |>
  mutate(
    decision = paste("decision",decision),
    part = paste("part",part)
  )


(
  ggplot(D |> filter(uid==6))
  +geom_segment(
    aes(
      x = I*vA,y=0,
      xend = 0, yend=I*vB,
      color=part
                ),linewidth=0.5)
  +facet_wrap(~decision)
  +geom_point(aes(x=allocation*vA,y=(I-allocation)*vB,color=part))
  +geom_vline(xintercept=0)+geom_hline(yintercept=0)
  +theme_minimal()
  +xlab("allocation to person A")+ylab("allocation to person B")
  +coord_fixed()
  +scale_color_manual(values = c("red", "blue"))
  +theme(legend.title = element_blank())
  #+geom_abline(slope=1,intercept=0,linetype="dashed")
)
```

Decisions 1 and 3 were slightly more complicated, in that the decision-maker faced *two* budget sets, with different endowments (I) and values to each person (v_A,v_B). If one of these decisions was chosen for payment, then *both* parts of the decision would be added to person A and B's earnings. For participant 6 in Figure 64 we can see that in decision 3, they allocated all of their part 2 tokens (blue) to person B, and all of their part 1 (red) tokens to person A.

To see how we might (qualitatively at least) distinguish between broad and narrow bracketing, I found it helpful to also look at the choice data for participant 1, which are shown in Figure 65. In particular, note in decision 3 that this participant chose close to the equal split for both parts of this decision. This could be rationalized under narrow bracketing by assuming that participant 1 likes payoff to be equal. However is cannot be rationalized under broad bracketing if we also assume that, all else held equal, participant 1 would like person A and B to get more money. This is becuase participant 1 could have allocated more part 1 tokens to A and more part 2 tokens to B, and both A and B could have received more earnings.

```
(
  ggplot(D |> filter(uid==1))
```

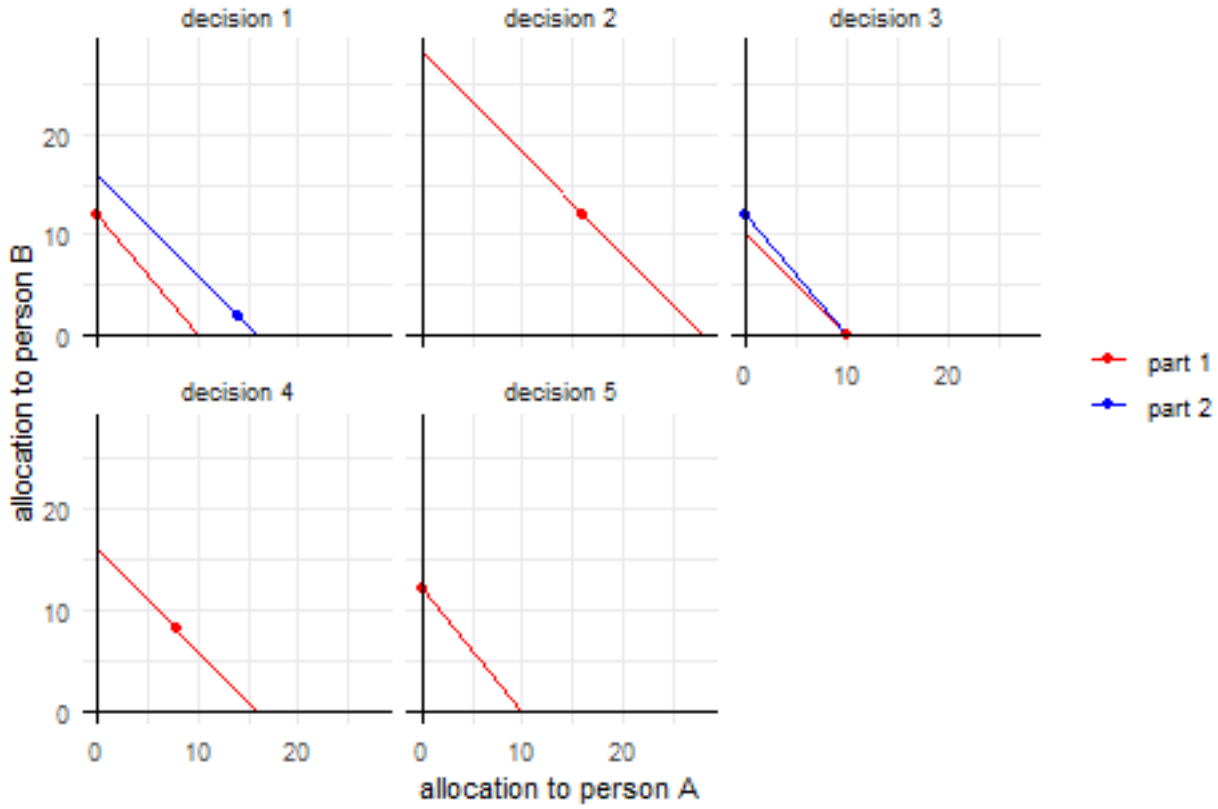


Figure 64: Choice data for participant 6, narrow framing.

```
  +geom_segment(
    aes(
      x = I*vA,y=0,
      xend = 0, yend=I*vB,
      color=part
                ),linewidth=0.5)
  +facet_wrap(~decision)
  +geom_point(aes(x=allocation*vA,y=(I-allocation)*vB,color=part))
  +geom_vline(xintercept=0)+geom_hline(yintercept=0)
  +theme_minimal()
  +xlab("allocation to person A")+ylab("allocation to person B")
  +coord_fixed()
  +scale_color_manual(values = c("red", "blue"))
  +theme(legend.title = element_blank())
)
```

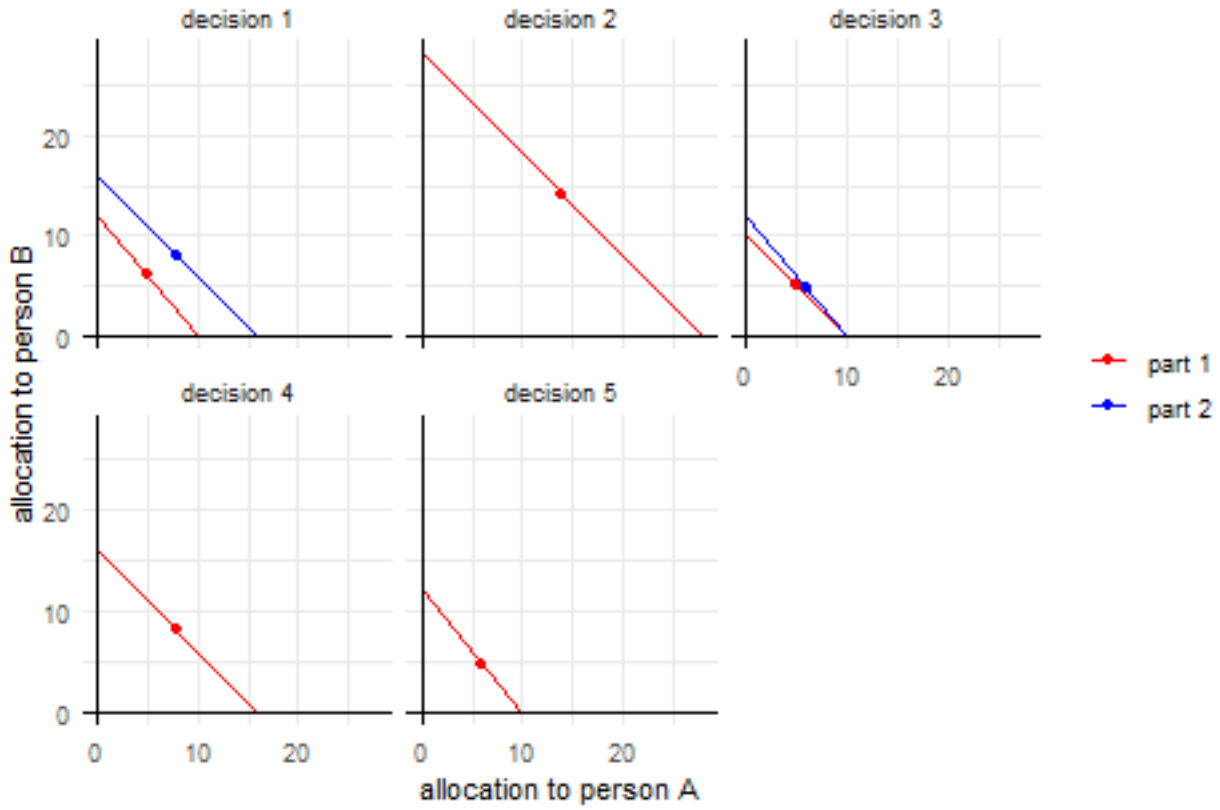


Figure 65: Choice data for participant 1, narrow framing.

In total, 102 participants made these five decisions. Choice sets were discretized to integer multiples of tokens, so in my analysis, I will assume logit choice over the discretized choice set.

So that's the dataset, now we need a model for bracketing behavior. For this, I will use the α-*partial bracketing* model (Barberis, Huang, and Thaler (2006), Barberis and Huang (2009), and Rabin and Weizsäcker (2009)). For this application, suppose that decision-makers have a utility function over earnings for person A and person B. which we denote $u(x_A, x_B)$, where x_A and x_B are the monetary earnings of A and B, respectively. In a 1-part decision, the decision-maker's utility-maximization problem is identical irrespective of how they

bracket:

$$\max_{y_A+y_B\leq I_d} u(v_{A,d}y_A,\ v_{B,d}y_B)$$

where I_d is token endowment in decision d, and $v_{A,d}$ and $v_{B,d}$ are the values of these tokens to person A and person B, respectively (I am writing this as a continuous choice problem, but when I get to the econometric model this will of course be a discrete choice problem).

For the 2-part decisions, the decision-maker's maximization problem if they broadly bracket is:

$$\max_{y^1_A+y^1_B\leq I^1_d,\ y^2_A+y^2_B\leq I^2_d} u(v^1_{A,d}y^1_A + v^2_{A,d}y^2_A,\ v^1_{B,d}y^1_B + v^2_{B,d}y^2_B)$$

That is, a broadly-bracketing decision-maker adds up the payoffs that each recipient will get from each part of the decision. A narrowly-bracketing decision-maker, on the other hand, maximizes utility for each part of the decision separately. Therefore we can write their maximization problem as:

$$\max_{y^1_A+y^1_B\leq I^1_d,\ y^2_A+y^2_B\leq I^2_d} u(v^1_{A,d}y^1_A,\ v^1_{B,d}y^1_B) + u(v^2_{A,d}y^2_A,\ v^2_{B,d}y^2_B)$$

Note that, while I have written this as joint maximization over the sum of two utility functions, this problem can be solved in two parts because they are additively separable.

Now that we have a (deterministic) model of both narrow and broad bracketing, we can get to the partial bracketing part of the model. Here we just take a convex combination of the two utility functions:

$$\max_{y^1_A+y^1_B\leq I^1_d,\ y^2_A+y^2_B\leq I^2_d} \alpha\left(u(v^1_{A,d}y^1_A,\ v^1_{B,d}y^1_B) + u(v^2_{A,d}y^2_A,\ v^2_{B,d}y^2_B)\right) + (1-\alpha)u(v^1_{A,d}y^1_A + v^2_{A,d}y^2_A,\ v^1_{B,d}y^1_B + v^2_{B,d}y^2_B)$$

where $\alpha\in[0,1]$ is the extent of narrow bracketing, and $\alpha=1$ ($\alpha=0$) corresponds to full narrow (broad) bracketing.

In order to make this an econometric model, we need to do two things. First, we need to assume a function form for $u(\cdot,\cdot)$. I chose to go with a constant elasticity of substitution (CES) utility function here with equal weights on each of person A and B's payoffs. Equal weights seems reasonable here because these were anonymous people in the lab, and so the decision-maker has no reason to value one of these people over another. Therefore, the functional form for the utility function will be:

$$u(x_A,x_B) = \left(x_A^\beta + x_B^\beta\right)^{\frac{1}{\beta}}$$

where $\beta>0$ is another parameter to be estimated.

Finally, we need to add a probabilistic component to this deterministic model. Here I am going to use logit choice over the action space, and use $\lambda>0$ as the choice precision parameter.

20.2 Representative agent and individual estimation

To begin with, I estimate a representative agent model, which I then also use to estimate on each participants' data separately. These can both use the same *Stan* program: we just pass different data to it. In the *Stan* program below, you can see that I chose to hard-code the payoffs for each of the five decisions in *Stan*. While I usually try to avoid this kind of thing, my previous iteration of this was having to re-construct the choice set and payoffs for every decision. This is because the size of the choice set varies between decision. As such, the data passed to *Stan* is the allocation of tokens made to Person A (`y1_1` and `y1_2` for decision 1, and so on). In order to avoid a mess of code in the `model` block, I added two functions in the `functions` block. `ll_1part` computes the log-likelihood of a decision for decisions with only one part, and `ll_2part` does the same for decisions with two parts.

```
functions {

    real ll_1part(
      data vector payA, data vector payB,
      real beta, real lambda,
      data int y
    ) {

      int dim = dims(payA)[1];

      vector[dim] U = lambda*pow(pow(payA,beta)+pow(payB,beta),1.0/beta);




      return log_softmax(U)[y+1];

    }


    real ll_2part(
      data matrix payA1, data matrix payB1,
      data matrix payA2, data matrix payB2,
      real alpha, real beta, real lambda,
      data int y1, data int y2
    )  {

      int dim[2] = dims(payA1);

      // utility if DM brackets broadly
      matrix[dim[1],dim[2]] Ubroad = pow(
        pow(payA1+payA2,beta)+pow(payB1+payB2,beta)
      ,1.0/beta);

      // utility if DB brackets narrowly
      matrix[dim[1],dim[2]] Unarrow =
        pow(pow(payA1,beta)+pow(payB1,beta),1.0/beta)
        +pow(pow(payA2,beta)+pow(payB2,beta),1.0/beta)
        ;

      // partial bracketing utility
      matrix[dim[1],dim[2]] U = lambda*(alpha*Unarrow+(1.0-alpha)*Ubroad);

      return U[y1+1,y2+1]-log_sum_exp(U);

    }
}

data {
  int N;

  vector[2] prior_alpha;
  vector[2] prior_beta;
```

```
  vector[2] prior_lambda;

  // allocations to player A
  int y1_1[N];
  int y1_2[N];
  int y2[N];
  int y3_1[N];
  int y3_2[N];
  int y4[N];
  int y5[N];

}

transformed data {

  /* Here I am going to hard-code the experiment parameters and choice sets
  This is becuase each decision has different endowments (I),
  and so it saves a lot of matrix-creation on the fly in the
  model block
  */

  // decision 1, 2 parts --------------------
  int I1_1 = 10;
  int I1_2 = 16;
  real vA1_1 = 1;
  real vB1_1 = 1.2;
  real vA1_2 = 1;
  real vB1_2 = 1;

  matrix[I1_1+1,I1_2+1] X1_1 = rep_matrix(linspaced_vector(I1_1+1, 0, I1_1),I1_2+1);
  matrix[I1_1+1,I1_2+1] X1_2 = rep_matrix(linspaced_row_vector(I1_2+1, 0, I1_2),I1_1+1);

  matrix[I1_1+1,I1_2+1] payA1_1 = vA1_1*X1_1;
  matrix[I1_1+1,I1_2+1] payB1_1 = vB1_1*(I1_1-X1_1);
  matrix[I1_1+1,I1_2+1] payA1_2 = vA1_2*X1_2;
  matrix[I1_1+1,I1_2+1] payB1_2 = vB1_2*(I1_2-X1_2);

  // decision 2, 1 part -------------------
  int I2 = 14;
  real vA2 = 2;
  real vB2 = 2;

  vector[I2+1] X2 = linspaced_vector(I2+1, 0, I2);

  vector[I2+1] payA2 = vA2*X2;
  vector[I2+1] payB2 = vB2*(I2-X2);

  // decision 3, 2 parts --------------------
  int I3_1 = 10;
  int I3_2 = 10;
  real vA3_1 = 1;
  real vB3_1 = 1;
  real vA3_2 = 1;
```

```
  real vB3_2 = 1.2;

  matrix[I3_1+1,I3_2+1] X3_1 = rep_matrix(linspaced_vector(I3_1+1, 0, I3_1),I3_2+1);
  matrix[I3_1+1,I3_2+1] X3_2 = rep_matrix(linspaced_row_vector(I3_2+1, 0, I3_2),I3_1+1);

  matrix[I3_1+1,I3_2+1] payA3_1 = vA3_1*X3_1;
  matrix[I3_1+1,I3_2+1] payB3_1 = vB3_1*(I3_1-X3_1);
  matrix[I3_1+1,I3_2+1] payA3_2 = vA3_2*X3_2;
  matrix[I3_1+1,I3_2+1] payB3_2 = vB3_2*(I3_2-X3_2);

  // decision 4, 1 part -------------------
  int I4 = 16;
  real vA4 = 1;
  real vB4 = 1;

  vector[I4+1] X4 = linspaced_vector(I4+1, 0, I4);

  vector[I4+1] payA4 = vA4*X4;
  vector[I4+1] payB4 = vB4*(I4-X4);

   // decision 5, 1 part -------------------
  int I5 = 10;
  real vA5 = 1;
  real vB5 = 1.2;

  vector[I5+1] X5 = linspaced_vector(I5+1, 0, I5);

  vector[I5+1] payA5 = vA5*X5;
  vector[I5+1] payB5 = vB5*(I5-X5);


}


parameters {

  real<lower=0,upper=1> alpha;
  real<lower=0> beta;
  real<lower=0> lambda;

}


model {

  // priors------------------------------------------------------------------

  alpha ~ beta(prior_alpha[1],prior_alpha[2]);
  beta ~ lognormal(prior_beta[1],prior_beta[2]);
  lambda ~ lognormal(prior_lambda[1],prior_lambda[2]);

  // likelihood --------------------------------------------------------------
```

```
  for (ii in 1:N) {


    // decision 1, 2 parts
    target += ll_2part(
      payA1_1, payB1_1,
      payA1_2, payB1_2,
      alpha,beta,lambda,
      y1_1[ii],y1_2[ii]
    );



    // decision 2, 1 part
    target += ll_1part(
      payA2, payB2,
      beta, lambda,
      y2[ii]
    );

     // decision 3, 2 parts
    target += ll_2part(
      payA3_1, payB3_1,
      payA3_2, payB3_2,
      alpha,beta,lambda,
      y3_1[ii],y3_2[ii]
    );

    // decision 4, 1 part
    target += ll_1part(
      payA4, payB4,
      beta, lambda,
      y4[ii]
    );

    // decision 5, 1 part
    target += ll_1part(
      payA5, payB5,
      beta, lambda,
      y5[ii]
    );


  }




}
```

Table 49 summarizes the posterior distribution of the representative agent model. Of primary interest here is the estimate for α, which is fairly precisely estimated to be about 20%. That is, the representative agent partially brackets, but their behavior is closer to broad bracketing than narrow bracketing.

Table 49: Summary of posterior distribution from the representative agent model.

				percentiles						
par	mean	se_mean	sd	X2.5.	X25.	X50.	X75.	X97.5.	n_eff	Rhat
alpha	0.208	0.001	0.033	0.148	0.185	0.206	0.228	0.274	3830.117	1.002
beta	0.515	0.001	0.036	0.440	0.492	0.516	0.539	0.579	2897.681	1.005
lambda	0.729	0.002	0.120	0.499	0.645	0.727	0.808	0.969	2970.372	1.005
lp___	-1514.550	0.022	1.253	-1517.772	-1515.102	-1514.220	-1513.636	-1513.127	3228.642	1.002

```
"Code/EF2024/social_repagent.csv" |>
  read.csv() |>
  dplyr::select(-X) |>
  kbl(digits=3,caption = "Summary of posterior distribution from the representative agent model.") |>
  kable_classic(full_width=FALSE) |>
  add_header_above(c("","","","","percentiles"=5,"",""))
```

Next, I loop through each participant individually. Figure 66 shows the empirical cumulative density function of the posterior mean estimates of α. Here we can see that the majority of participants narrowly bracket to an extent of at least 50%, but no participant is close to fully narrowly bracketing their decisions. There are also a few participants with α very close to zero, indicating that they broadly bracket their decisions.[90]

```
d<-"Code/EF2024/social_individual.csv" |>
  read.csv() |>
  filter(par=="alpha") |>
  mutate(
    cumul = rank(mean)/n()-0.5/n()
  )

(
  ggplot(d,aes(x=mean))
  +stat_ecdf()
  +theme_bw()
  +geom_errorbar(aes(xmin = X25.,xmax = X75., y=cumul),alpha=0.5,color="red")
  +xlim(c(0,1))
  +xlab(expression(alpha~"(extent of narrow bracketing)"))
  +ylab("cumulative density of posterior mean")
)
```

While we might not be too interested in the other parameters in the model, it doesn't hurt to take a quick look at them and make sure that there are no surprises. While I don't have much intuition about what a reasonable value for λ is,[91] *a priori* it seems like we should mostly be estimating $0 < \beta < 1$. $0 < \beta$ will always be true because I encoded it into the prior, but $\beta < 1$ is particularly important becuase it indicates that the utility function is quasiconcave. For us, this means that our participants have convex preferences over person A and B's earnings, which seems reasonable here. Let's check it in Figure 67:

```
d<-"Code/EF2024/social_individual.csv" |>
  read.csv() |>
  filter(par=="beta")  |>
  mutate(
    cumul = rank(mean)/n()-0.5/n()
  )
```

[90] The model would *never* estimate that a participant either fully brackets broadly ($\alpha = 0$) or fully brackets narrowly ($\alpha = 1$). This is becuase prior weight is placed on the interior of the unit interval.

[91] Read: "Today I am too lazy to make a few plots to help my intuition here."

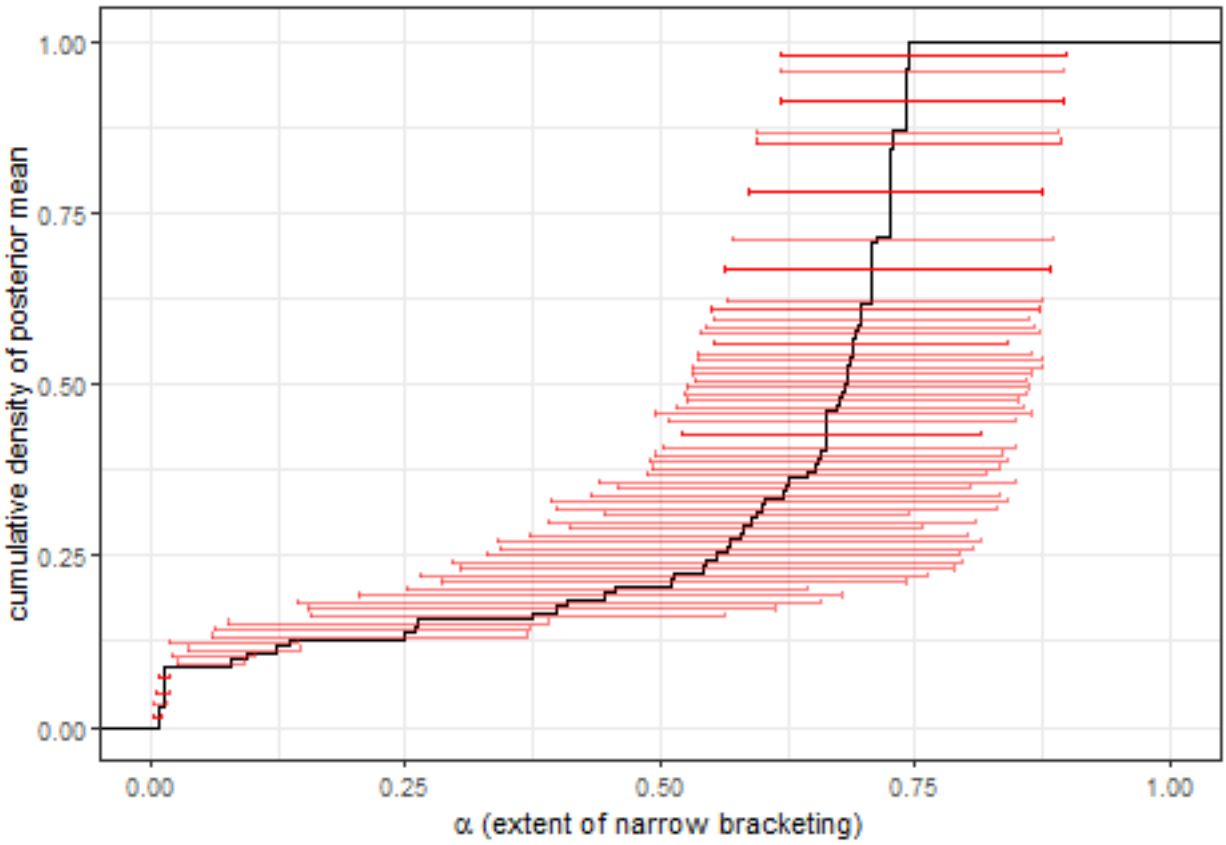


Figure 66: Estimates of extent of narrow bracketing at the individual level. Black line shows the empirical cumulative distribution function of the posterior mean. Red error bars show 50% Bayesian credible regions (25th-75th percentile)

```
(
  ggplot(d,aes(x=mean))
  +stat_ecdf()
  +theme_bw()
  +geom_errorbar(aes(xmin = X25.,xmax = X75., y=cumul),alpha=0.5,color="red")
  #+xlim(c(0,1))
  +xlab(expression(beta))
  +ylab("cumulative density of posterior mean")
)
```

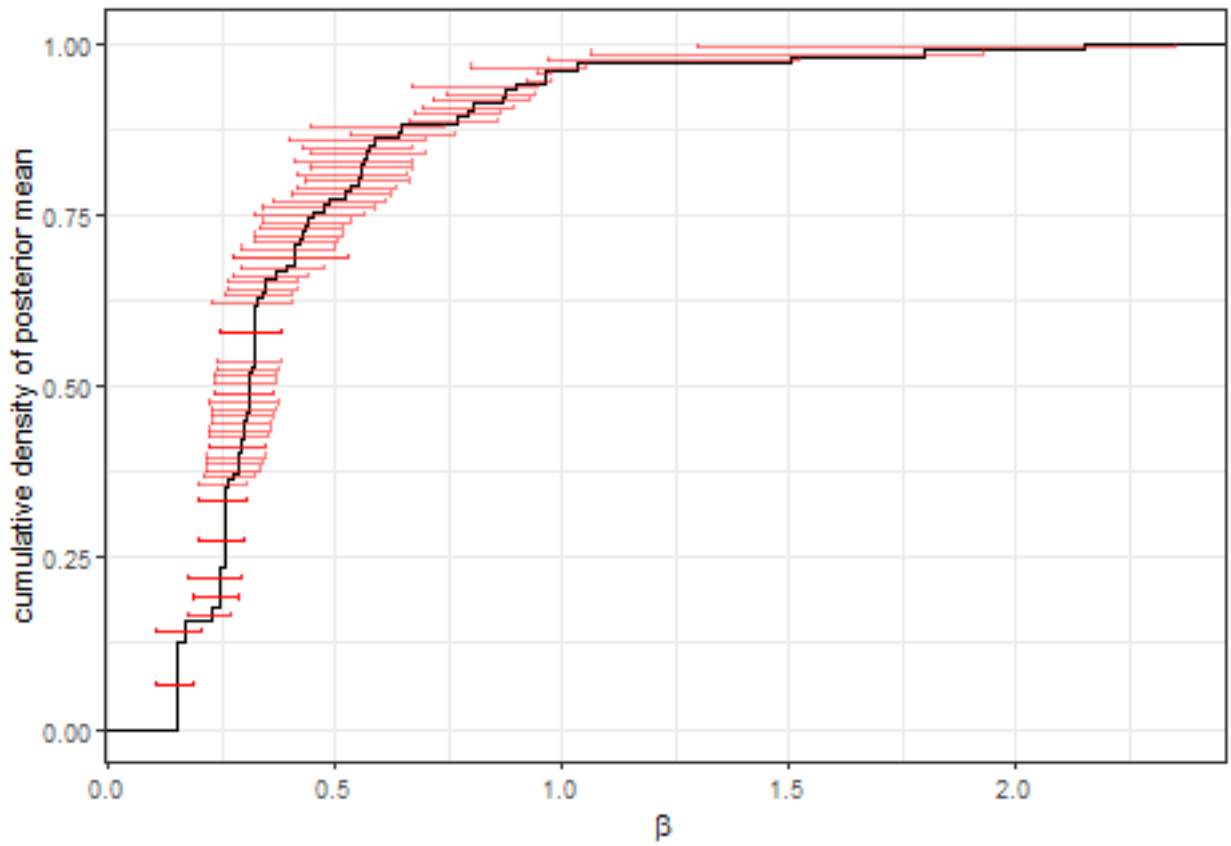


Figure 67: Estimates of β at the individual level. Black line shows the empirical cumulative distribution function of the posterior mean. Red error bars show 50% Bayesian credible regions (25th-75th percentile)

$\beta < 1$ for the vast majority of participants, good!

Before moving on to the hierarchical model, I want to stress an important thing that we would have missed if we only estimated the representative agent model. Recall that in the representative agent model we estimated $\alpha \approx 0.2$. However a quick-and-dirty sample mean of the posterior means from the individual-level estimations yields:

```
d$mean |> mean()
```

```
## [1] 0.4122716
```

So our conclusions about bracketing would be *vastly* different depending on which of these summary statistics we believed! The representative agent model would have us believe that narrow bracketing is only a little bit important, and that decisions are closer to broad bracketing. On the other hand, the individual-level

estimation suggests that most people bracket substantially closer to narrow than broad.

20.3 Hierarchical model

Here is the *Stan* program I wrote to estimate the hierarchical model. It of course looks very similar to the representative agent model except for the bits where the individual-level parameters go in. This is why I always want you to write and estimate the representative agent model first, even if you're not going to use it: it is a very important stepping stone in getting to the richer hierarchical model, and will probably help you iron out a lot of the problems that are specific to your application (for this one, it was working out an efficient way to hard-code the experiment parameters).

```
functions {

    real ll_1part(
      data vector payA, data vector payB,
      real beta, real lambda,
      data int y
    ) {

      int dim = dims(payA)[1];

      vector[dim] U = lambda*pow(pow(payA,beta)+pow(payB,beta),1.0/beta);




      return log_softmax(U)[y+1];

    }


    real ll_2part(
      data matrix payA1, data matrix payB1,
      data matrix payA2, data matrix payB2,
      real alpha, real beta, real lambda,
      data int y1, data int y2
    )  {

      int dim[2] = dims(payA1);

      // utility if DM brackets broadly
      matrix[dim[1],dim[2]] Ubroad = pow(
        pow(payA1+payA2,beta)+pow(payB1+payB2,beta)
      ,1.0/beta);

      // utility if DB brackets narrowly
      matrix[dim[1],dim[2]] Unarrow =
        pow(pow(payA1,beta)+pow(payB1,beta),1.0/beta)
        +pow(pow(payA2,beta)+pow(payB2,beta),1.0/beta)
        ;

      // partial bracketing utility
      matrix[dim[1],dim[2]] U = lambda*(alpha*Unarrow+(1.0-alpha)*Ubroad);
```

```
      return U[y1+1,y2+1]-log_sum_exp(U);

    }
}

data {
  int N; // number of participants

  vector[3] prior_MU[2];
  vector[3] prior_TAU;
  real prior_Omega;

  // allocations to player A
  int y1_1[N];
  int y1_2[N];
  int y2[N];
  int y3_1[N];
  int y3_2[N];
  int y4[N];
  int y5[N];

}

transformed data {

  /* Here I am going to hard-code the experiment parameters and choice sets
  This is becuase each decision has different endowments (I),
  and so it saves a lot of matrix-creation on the fly in the
  model block
  */

  // decision 1, 2 parts --------------------
  int I1_1 = 10;
  int I1_2 = 16;
  real vA1_1 = 1;
  real vB1_1 = 1.2;
  real vA1_2 = 1;
  real vB1_2 = 1;

  matrix[I1_1+1,I1_2+1] X1_1 = rep_matrix(linspaced_vector(I1_1+1, 0, I1_1),I1_2+1);
  matrix[I1_1+1,I1_2+1] X1_2 = rep_matrix(linspaced_row_vector(I1_2+1, 0, I1_2),I1_1+1);

  matrix[I1_1+1,I1_2+1] payA1_1 = vA1_1*X1_1;
  matrix[I1_1+1,I1_2+1] payB1_1 = vB1_1*(I1_1-X1_1);
  matrix[I1_1+1,I1_2+1] payA1_2 = vA1_2*X1_2;
  matrix[I1_1+1,I1_2+1] payB1_2 = vB1_2*(I1_2-X1_2);

  // decision 2, 1 part -------------------
  int I2 = 14;
  real vA2 = 2;
  real vB2 = 2;

  vector[I2+1] X2 = linspaced_vector(I2+1, 0, I2);
```

```
  vector[I2+1] payA2 = vA2*X2;
  vector[I2+1] payB2 = vB2*(I2-X2);

  // decision 3, 2 parts --------------------
  int I3_1 = 10;
  int I3_2 = 10;
  real vA3_1 = 1;
  real vB3_1 = 1;
  real vA3_2 = 1;
  real vB3_2 = 1.2;

  matrix[I3_1+1,I3_2+1] X3_1 = rep_matrix(linspaced_vector(I3_1+1, 0, I3_1),I3_2+1);
  matrix[I3_1+1,I3_2+1] X3_2 = rep_matrix(linspaced_row_vector(I3_2+1, 0, I3_2),I3_1+1);

  matrix[I3_1+1,I3_2+1] payA3_1 = vA3_1*X3_1;
  matrix[I3_1+1,I3_2+1] payB3_1 = vB3_1*(I3_1-X3_1);
  matrix[I3_1+1,I3_2+1] payA3_2 = vA3_2*X3_2;
  matrix[I3_1+1,I3_2+1] payB3_2 = vB3_2*(I3_2-X3_2);

  // decision 4, 1 part -------------------
  int I4 = 16;
  real vA4 = 1;
  real vB4 = 1;

  vector[I4+1] X4 = linspaced_vector(I4+1, 0, I4);

  vector[I4+1] payA4 = vA4*X4;
  vector[I4+1] payB4 = vB4*(I4-X4);

   // decision 5, 1 part -------------------
  int I5 = 10;
  real vA5 = 1;
  real vB5 = 1.2;

  vector[I5+1] X5 = linspaced_vector(I5+1, 0, I5);

  vector[I5+1] payA5 = vA5*X5;
  vector[I5+1] payB5 = vB5*(I5-X5);


}


parameters {

  vector[3] MU;
  vector<lower=0>[3] TAU;
  cholesky_factor_corr[3] L_Omega;

  matrix[3,N] z;

}
```

```
transformed parameters {

  vector[N] alpha;
  vector[N] beta;
  vector[N] lambda;

  {
    matrix[3,N] theta = rep_matrix(MU,N)
                    //+diag_matrix(TAU)*z;
                    +diag_pre_multiply(TAU,L_Omega)*z;

    alpha = Phi_approx(theta[1,]');
    beta = exp(theta[2,]');
    lambda = exp(theta[3,]');

  }

}


model {

  // hierarchical structure with noncentered parameterization ------------------
  to_vector(z) ~ std_normal();

  // priors--------------------------------------------------------------------
  MU ~ normal(prior_MU[1],prior_MU[2]);
  TAU ~ cauchy(0.0,prior_TAU);
  L_Omega ~ lkj_corr_cholesky(prior_Omega);


  // likelihood ----------------------------------------------------------------

  for (ii in 1:N) {

    real a = alpha[ii];
    real b = beta[ii];
    real l = lambda[ii];

    // decision 1, 2 parts
    target += ll_2part(
      payA1_1, payB1_1,
      payA1_2, payB1_2,
      a,b,l,
      y1_1[ii],y1_2[ii]
    );



    // decision 2, 1 part
    target += ll_1part(
      payA2, payB2,
      b,l,
```

```
      y2[ii]
    );

     // decision 3, 2 parts
    target += ll_2part(
      payA3_1, payB3_1,
      payA3_2, payB3_2,
      a,b,l,
      y3_1[ii],y3_2[ii]
    );

    // decision 4, 1 part
    target += ll_1part(
      payA4, payB4,
      b,l,
      y4[ii]
    );

    // decision 5, 1 part
    target += ll_1part(
      payA5, payB5,
      b,l,
      y5[ii]
    );


  }




}
```

Figure 68 shows the shrinkage estimates for α from the hierarchical model. (These have the same interpretation as the individual-level estimates in Figure 66). What struck me as surprising here (and I really should have seen it when I looked at the individual-level estimates in Figure @ref(fig:EF2024-individualecdf,)) was the bi-modality of a distribution. In an ecdf you can see the modes of a distribution as very steep parts. There appears to be one at about $\alpha = 0.4$, and another at $\alpha = 0.75$. This may pose a problem for the hierarchical model, which (and I'm waving my hands a lot here) is really hoping for a uni-modal distribution, or even better yet, a continuous *normal* distribution (of the transformed variable). This can be addressed by selecting a more flexible distribution, but I will leave this for another day.

```
d<-"Code/EF2024/social_hierarchical.csv" |>
  read.csv() |>
  filter(grepl("alpha",par)) |>
  mutate(
    cumul = rank(mean)/n()-0.5/n()
  )

(
  ggplot(d,aes(x=mean))
  +stat_ecdf()
  +theme_bw()
```

```
  +geom_errorbar(aes(xmin = X25.,xmax = X75., y=cumul),alpha=0.5,color="red")
  +xlim(c(0,1))
  +xlab(expression(alpha~"(extent of narrow bracketing)"))
  +ylab("cumulative density of posterior mean")
)
```

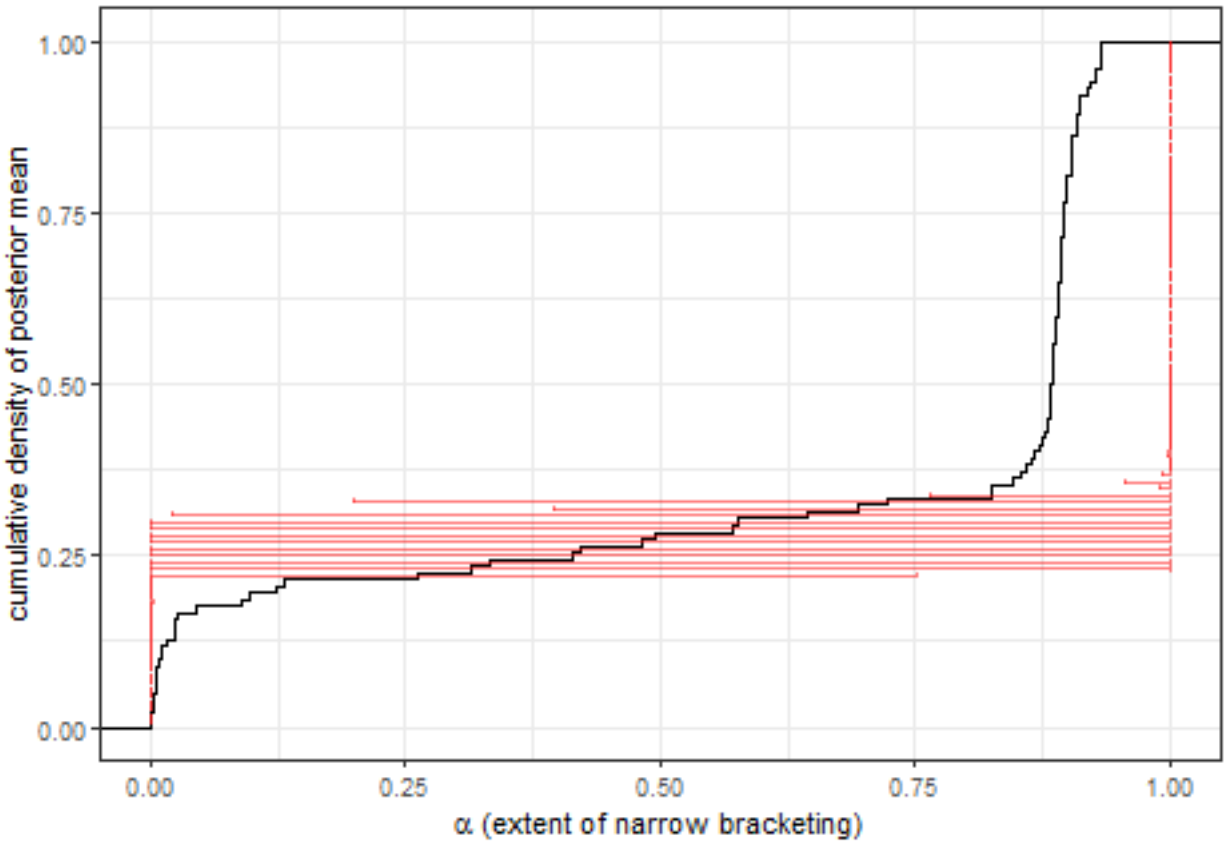


Figure 68: Shrinkage estimates of extent of narrow bracketing at the individual level from the hierarchical model. Black line shows the empirical cumulative distribution function of the posterior mean. Red error bars show 50% Bayesian credible regions (25th-75th percentile)

20.4 A mixture model

One thing that I was worried about while doing the above work was that there were only five decisions per participant. Furthermore only two of these decisions had two parts, and so it was really only these decisions that provided us with information about α, the extent of narrow bracketing.[92] This is certainly reflected in the large credible regions for α for the individual-level estimation (recall that these were only 50% credible regions), and so we are not estimating α particularly precisely. Instead of assuming the partial bracketing model, we could simplify things by instead assuming that participants either fully narrowly bracketed or fully broadly bracketed. That is, we have a 2-type (discrete) mixture model. At the individual level, this turns the problem from an *estimation* (of α) problem to a *classification* (into $\alpha = 0$ or $\alpha = 1$) problem. Our model will now tell us the *fraction* of narrow bracketers in the population, and we will be able to assign a posterior probability to each participant that they are a narrow bracketer.

[92]It is not that the other three decisions provided us with nothing. They help pin down β and λ so that less uncertainty in these variables propogates into uncertainty in α.

Here is the *Stan* program I wrote to estimate this mixture model. Importantly, I am preserving the hierarchical structure for the parameters β and λ, which if we are just interested in bracketing behavior, could be considered nuisance parameters for this inferential goal.[93] Hence, our model will be in some sense robust to participant-level heterogeneity in these parameters. Note that I have moved a lot of the calculations into the `transofrmed parameters` block so I can use the `ll` variable into both the `model` (to increment the log-posterior) and `generated quantities` (to calculate posterior type probabilities) blocks.

```
functions {

    real ll_1part(
      data vector payA, data vector payB,
      real beta, real lambda,
      data int y
    ) {

      int dim = dims(payA)[1];

      vector[dim] U = lambda*pow(pow(payA,beta)+pow(payB,beta),1.0/beta);




      return log_softmax(U)[y+1];

    }


    real ll_2part(
      data matrix payA1, data matrix payB1,
      data matrix payA2, data matrix payB2,
      real alpha, real beta, real lambda,
      data int y1, data int y2
    )  {

      int dim[2] = dims(payA1);

      // utility if DM brackets broadly
      matrix[dim[1],dim[2]] Ubroad = pow(
        pow(payA1+payA2,beta)+pow(payB1+payB2,beta)
      ,1.0/beta);

      // utility if DB brackets narrowly
      matrix[dim[1],dim[2]] Unarrow =
        pow(pow(payA1,beta)+pow(payB1,beta),1.0/beta)
        +pow(pow(payA2,beta)+pow(payB2,beta),1.0/beta)
        ;

      // partial bracketing utility
      matrix[dim[1],dim[2]] U = lambda*(alpha*Unarrow+(1.0-alpha)*Ubroad);

      return U[y1+1,y2+1]-log_sum_exp(U);
```

[93] If instead we wanted to (say) assign a welfare measure to narrow bracketing (e.g. "how much utility does someone give up by bracketing narrowly?") then these might not be nuisance parameters. See Bland (2019a) and Bland (2019b) for examples of this.

```
    }
}

data {
  int N; // number of participants

  vector[2] prior_MU[2];
  vector[2] prior_TAU;
  real prior_Omega;
  vector<lower=0.0>[2] prior_mix;

  // allocations to player A
  int y1_1[N];
  int y1_2[N];
  int y2[N];
  int y3_1[N];
  int y3_2[N];
  int y4[N];
  int y5[N];

}

transformed data {

  /* Here I am going to hard-code the experiment parameters and choice sets
  This is becuase each decision has different endowments (I),
  and so it saves a lot of matrix-creation on the fly in the
  model block
  */

  // decision 1, 2 parts --------------------
  int I1_1 = 10;
  int I1_2 = 16;
  real vA1_1 = 1;
  real vB1_1 = 1.2;
  real vA1_2 = 1;
  real vB1_2 = 1;

  matrix[I1_1+1,I1_2+1] X1_1 = rep_matrix(linspaced_vector(I1_1+1, 0, I1_1),I1_2+1);
  matrix[I1_1+1,I1_2+1] X1_2 = rep_matrix(linspaced_row_vector(I1_2+1, 0, I1_2),I1_1+1);

  matrix[I1_1+1,I1_2+1] payA1_1 = vA1_1*X1_1;
  matrix[I1_1+1,I1_2+1] payB1_1 = vB1_1*(I1_1-X1_1);
  matrix[I1_1+1,I1_2+1] payA1_2 = vA1_2*X1_2;
  matrix[I1_1+1,I1_2+1] payB1_2 = vB1_2*(I1_2-X1_2);

  // decision 2, 1 part --------------------
  int I2 = 14;
  real vA2 = 2;
  real vB2 = 2;

  vector[I2+1] X2 = linspaced_vector(I2+1, 0, I2);
```

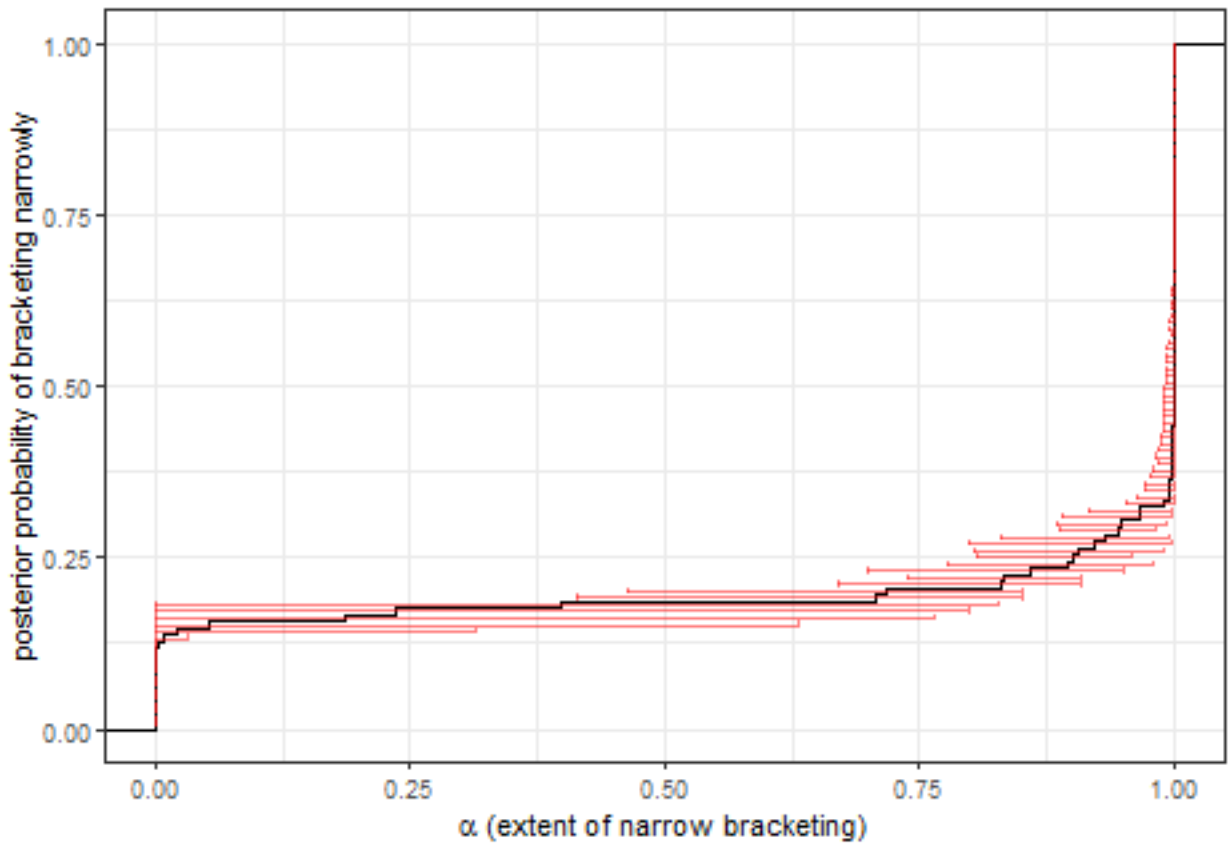
```
  vector[I2+1] payA2 = vA2*X2;
  vector[I2+1] payB2 = vB2*(I2-X2);

  // decision 3, 2 parts --------------------
  int I3_1 = 10;
  int I3_2 = 10;
  real vA3_1 = 1;
  real vB3_1 = 1;
  real vA3_2 = 1;
  real vB3_2 = 1.2;

  matrix[I3_1+1,I3_2+1] X3_1 = rep_matrix(linspaced_vector(I3_1+1, 0, I3_1),I3_2+1);
  matrix[I3_1+1,I3_2+1] X3_2 = rep_matrix(linspaced_row_vector(I3_2+1, 0, I3_2),I3_1+1);

  matrix[I3_1+1,I3_2+1] payA3_1 = vA3_1*X3_1;
  matrix[I3_1+1,I3_2+1] payB3_1 = vB3_1*(I3_1-X3_1);
  matrix[I3_1+1,I3_2+1] payA3_2 = vA3_2*X3_2;
  matrix[I3_1+1,I3_2+1] payB3_2 = vB3_2*(I3_2-X3_2);

  // decision 4, 1 part -------------------
  int I4 = 16;
  real vA4 = 1;
  real vB4 = 1;

  vector[I4+1] X4 = linspaced_vector(I4+1, 0, I4);

  vector[I4+1] payA4 = vA4*X4;
  vector[I4+1] payB4 = vB4*(I4-X4);

   // decision 5, 1 part -------------------
  int I5 = 10;
  real vA5 = 1;
  real vB5 = 1.2;

  vector[I5+1] X5 = linspaced_vector(I5+1, 0, I5);

  vector[I5+1] payA5 = vA5*X5;
  vector[I5+1] payB5 = vB5*(I5-X5);


}


parameters {

  vector[2] MU;
  vector<lower=0>[2] TAU;
  cholesky_factor_corr[2] L_Omega;
  real<lower=0,upper=1> mix_narrow;

  matrix[2,N] z;

}
```

```
transformed parameters {

  vector[N] beta;
  vector[N] lambda;

  matrix[N,2] ll;

  {
    matrix[2,N] theta = rep_matrix(MU,N)
                    //+diag_matrix(TAU)*z;
                    +diag_pre_multiply(TAU,L_Omega)*z;

    beta = exp(theta[1,]');
    lambda = exp(theta[2,]');



    // likelihood computation ------------------------------------------------------------------

  for (ii in 1:N) {

    row_vector[2] ll_i = [log(mix_narrow),log(1.0-mix_narrow)];

    real b = beta[ii];
    real l = lambda[ii];

    // decision 1, 2 parts,
    //narrow
    ll_i[1] += ll_2part(
      payA1_1, payB1_1,
      payA1_2, payB1_2,
      1.0,b,l,
      y1_1[ii],y1_2[ii]
    );
    // broad
    ll_i[2] += ll_2part(
      payA1_1, payB1_1,
      payA1_2, payB1_2,
      0.0,b,l,
      y1_1[ii],y1_2[ii]
    );

    // decision 2, 1 part
    ll_i += ll_1part(
      payA2, payB2,
      b,l,
      y2[ii]
    );

     // decision 3, 2 parts
     // narrow
     ll_i[1] += ll_2part(
      payA3_1, payB3_1,
```

```
      payA3_2, payB3_2,
      1.0,b,l,
      y3_1[ii],y3_2[ii]
    );
    // broad
     ll_i[2] += ll_2part(
      payA3_1, payB3_1,
      payA3_2, payB3_2,
      0.0,b,l,
      y3_1[ii],y3_2[ii]
    );

    // decision 4, 1 part
    ll_i += ll_1part(
      payA4, payB4,
      b,l,
      y4[ii]
    );

    // decision 5, 1 part
    ll_i += ll_1part(
      payA5, payB5,
      b,l,
      y5[ii]
    );

    ll[ii,] = ll_i;
  }

  }

}


model {

  // hierarchical structure with noncentered parameterization ------------------
  to_vector(z) ~ std_normal();

  // priors--------------------------------------------------------------------
  MU ~ normal(prior_MU[1],prior_MU[2]);
  TAU ~ cauchy(0.0,prior_TAU);
  L_Omega ~ lkj_corr_cholesky(prior_Omega);
  mix_narrow ~ beta(prior_mix[1],prior_mix[2]);


  for (ii in 1:N) {
    target+= log_sum_exp(ll[ii,]);
  }

}

generated quantities {
```

```
  vector[N] postprob_narrow;
  for (ii in 1:N) {
    postprob_narrow[ii] = softmax(ll[ii,]')[1];
  }

}
```

```
d<-"Code/EF2024/social_mixture.csv" |>
  read.csv() |>
  filter(grepl("postprob",par))|>
  mutate(
    cumul = rank(mean)/n()-0.5/n()
  )


(
  ggplot(d,aes(x=mean))
  +stat_ecdf()
  +theme_bw()
  +geom_errorbar(aes(xmin = X2.5.,xmax = X97.5., y=cumul),alpha=0.5,color="red")
  +xlim(c(0,1))
  +xlab(expression(alpha~"(extent of narrow bracketing)"))
  +ylab("posterior probability of bracketing narrowly")
)
```



20.5 What do we get out of the structural models, and what could we miss?

I have analyzed these data in a *really* different way to Ellis and Freeman (2024). There is no "Bayesian alternative to a maximum likelihood implementation" for me to do here like I can do for (say) strategy frequency estimation or structurally estimating risk preferences. Ellis and Freeman (2024) use nonparametric measures of mistakes to classify participants based on the number of mistakes they make relative to (say) Weak Axiom of Revealed Preference (WARP) predictions. Loosely speaking, this kind of analysis can put participants into *three* bins: (i) narrow bracketing, (ii) broad bracketing, and (iii) something else. This "something else" bin is *really* important, because it could be that participants are doing something other than narrowly or broadly bracketing. Ignoring this bin means that we will mis-classify these "something else" participants into one of the bracketing bins. This mis-classification may not be random, and so the resulting fractions may over- or under-state the prevalence of each bracketing behavior in the sample.

On the other hand, my approach forces every participant to be a partial bracketer (with different extents of bracketing, α), or in the case of the mixture model, forces everybody to be either a broad or narrow bracketer. Out of this we get an easily interpretable measure of bracketing behavior for each participant, but we cannot say anything about the "something else". This is not to say that we couldn't also include something else in the structural model, but it would have to be a *specific* "something else" in the form of another parametric model.

21 Application: Ranked choices and the Thurstonian model

Up to this point, I have shown you models for choosing *one thing* out of a choice set. This is often how we present things to our participants. In these situations, revealed preference tells us that the chosen item must be the item yielding the greatest utility of all items in the choice set. From a probabilistic choice perspective, we know that the utility of this item plus an error was the greatest. However in other experiments we might ask our participants to *rank* the items in a choice set. In principle, this gives a *a lot* more information, because not only do we know the payoff-maximizing item (i.e. the item ranked first), but we also know how all of the other items compare, too.[94]

While it is in principle possible to write down a (say) logit choice model where each option is one way to rank the items in the choice set, this will become computationally infeasible very quickly. Consider for example the choice data shown in Table 50, which is the ranking of nine different distributions of earnings made by the first participant in the first game of Alempaki et al. (2022). There are $9! = 362,880$ possible ways to rank these payoffs! In practice we will need a more tractable model for how people make decisions in environments like these.

```
ranking<-"Data/ACKLP2022_ranking.dta" |>
  read_dta() |>
  mutate(uid = paste("id",id) |> as.factor() |> as.numeric()) |>
  arrange(uid,game)

ranking |>
  filter(uid==1 & game==1) |>
  dplyr::select(ranking,own,other) |>
  arrange(ranking) |>
  kbl(caption = "Ranking of payoffs in game 1 made by participant 1,") |>
  kable_classic(full_width=FALSE)
```

[94] This of course comes with the caveat that a ranking task may be more confusing to participants, and so decisions in a ranking task may be more noisy. I do not want to weigh in on the methodology here, instead we will just proceed with the understanding that we have ranked data.

Table 50: Ranking of payoffs in game 1 made by participant 1,

ranking	own	other
1	8	2
2	8	0
3	6	6
4	6	8
5	4	4
6	2	2
7	0	2
8	0	8
9	0	10

21.1 The Thurstonian model

One way to tractably handle ordered choice data is with a *Thurstonian model*. This model assumes that each item k in the choice set j has a latent variable $z_{j,k}$. The ranking we observe in the data is the ranking of the realization of the vector z_j. We then assume a distribution for this latent vector, and estimate the parameters in this distribution. For example, we might assume a normal distribution:

$$z_{j,k} \sim \text{ind } N(\mu_{j,k}, \sigma^2)$$

From a structural perspective, we can interpret $z_{j,k}$ as the utility of item k in choice set j. We can let the mean of this latent variable $\mu_{j,k}$ be a function of the data and our model's parameters, which means that we can use this to estimate a utility function! That is, suppose we have a utility function $U(x;\theta)$ with arguments x and parameters θ, then we can set $\mu_{j,k} = U(x_{j,k};\theta)$.

To see how this model relates to models of choosing a single thing from a choice set, consider the special case where the choice set has just two items. In this case, the participant will rank item $k = 1$ over $k = 2$ if and only if $z_{j,1} > z_{j,2}$, which occurs with probability:

$$\begin{aligned}
\Pr(z_{j,1} > z_{j,2}) &= \Pr\left(U(x_{j,1};\theta) - U(x_{j,2};\theta) + \sigma(\xi_1 - \xi_2) > 0\right) \qquad \xi_1, \xi_2 \sim iidN(0,1) \\
&= \Pr\left(\frac{U(x_{j,1};\theta) - U(x_{j,2};\theta)}{\sigma} > \xi_2 - \xi_1\right) \\
&= \Phi\left(\frac{U(x_{j,1};\theta) - U(x_{j,2};\theta)}{\sqrt{2}\sigma}\right)
\end{aligned}$$

where $\Phi(\cdot)$ is the standard normal cdf, so it reduces to a model of probit choice!

As I have described the model up to this point, it is not identifiable if each $\mu_{j,k}$ is a free parameter. This is because adding a constant to the mean vector μ_j will generate the same predictions. A simple solution to this is to normalize the mean utility of one of the items in the choice set to zero. However in the example I present below, we do not need to do this, because our utility function does not have a constant in it. In fact, this should be the case for all of our utility functions, becuase we can always represent the same preferences by adding a constant to all of our utilities.

The trick to estimating this model in *Stan* is to sort the data by the ranking, then take advantage of the `ordered vector` data type. That is, I pass the data about each choice set to *Stan* so that it is ordered from worst to best. That way, I know that the latent utilities must be ordered like this:

$$z_{j,K} > z_{j,K-1} > z_{j,K-2} > \ldots > z_{j,1}$$

I then use data augmentation to draw these z_js. This part is really easy once you organize the data in the right way (outcomes ranked from worst to best). You will see in the *Stan* files below that all you really need to do to augment the data here is to just declare the distribution (i.e. `z ~ normal(U,sigma);`). The `ordered` vector takes care of the constraints.

21.2 Computational issues

All of this data augmentation means that there will be a lot of numbers *Stan* needs to keep track of: every $z_{j,k}$ needs to be augmented. For example, suppose you have N participants each do J ranking tasks, each with K things to rank in each task. This means that your model needs to augment NJK pieces of data on top of anything else that you are doing. This is a lot more than you would normally do with a hierarchical model that just needs to augment the individual-level parameters. In this case if you have p individual-level parameters per participant, you would just be augmenting Np things.[95] All of this is to say, beware of how much memory this augmentation is chewing up, and consider not saving these augmented data from your estimation, because there will be thousands of realizations for each of these variables. Think about whether you would actually need them.

21.3 Example dataset and model

Alempaki et al. (2022) elicit their participants' preferences over the outcomes in eight games by asking them to rank these outcomes. An example of this ranking for the first participant in the first game is shown in Table 50. Here "own" refers to the participant's own monetary payoff, and "other" refers to their opponent's monetary payoff. In order to incentivize truth-telling, if this ranking task was selected for payment, two of the outcomes were randomly selected. The outcome that the participant ranked higher was then implemented. For example based on the choices in Table 50, if outcomes $(8, 0)$ and $(2, 2)$ were randomly selected, $(8, 0)$ would be implemented because this participant ranked this outcome higher.

As Alempaki et al. (2022) were concerned about other-regarding preferences in their games, we will estimate a simple 1-parameter utility model for other-regarding preferences:

$$U(x; \alpha) = x^{\text{own}} + \alpha x^{\text{other}}$$

where x^{own} is the decisioon-maker's own earnings, and x^{other} is the other participant's earnings. Here I am trying to keep things a simple as possible (while still hopefully being interesting) becuase I was having trouble estimating the full Fehr and Schmidt (1999) model of inequality-aversion. Maybe when I understand more the pathologies of this model I will extend this chapter to include this.[96]

21.4 A representative agent model

While at this point you know how I feel about the value of reporting representative agent models, this one was really useful in helping me to understand the Thurstoninal model, so I present it here as a Thurstonian *only* model without the hierarchical stuff that should really go with it. But don't worry, we'll put all of that back in before the end of the chapter.

This model has two parameters, α and σ, so we willset thesir priors to:

$$\begin{aligned}\alpha &\sim N(0, 1)\\ \sigma &\sim \text{Cauchy}^{+}(0, 1)\end{aligned}$$

These are fairly diffuse. Probably too diffuse if I actually calibrated them properly (especially for α), but they will pin down the model enough.

Without further ado, here is the *Stan* program I came up with to implement the model.

[95] I struggle to think of a tractable model/experiment combination where $p > JK$.

[96] Specifically, *Stan* was giving me some BFMI warnings.

```
data {

  int nparticipants;

  int<lower=0> N9; // total number of decisions with 9 unique payoffs
  int<lower=0> N8; // total number of decisions with 8 unique payoffs
  int<lower=0> N7; // total number of decisions with 7 unique payoffs

  int id9[N9];
  int id8[N8];
  int id7[N7];

  // payoffs from the game, ranked in ordered choice, worst to best
    // for the games with 9 uninque payoffs
    vector[9] own9[N9];
    vector[9] other9[N9];
    // for the games with 8 uninque payoffs
    vector[8] own8[N8];
    vector[8] other8[N8];
    // for the games with 9 uninque payoffs
    vector[7] own7[N7];
    vector[7] other7[N7];

    vector[2] prior_alpha;
    real prior_sigma;
}

parameters {

  real alpha; // coefficient on other's payoff
  real<lower=0> sigma; // sd of latent variable

  // augmented data
  ordered[9] z9[N9];
  ordered[8] z8[N8];
  ordered[7] z7[N7];


}

transformed parameters {


}

model {



  for (ii in 1:N9) {

    // utility
    vector[9] U = own9[ii]+alpha*other9[ii];
```

parameter	mean	se_mean	sd	X2.5.	X25.	X50.	X75.	X97.5.	n_eff	Rhat
alpha	0.15	0.00	0.01	0.14	0.15	0.15	0.16	0.17	4433.08	1
sigma	1.52	0.00	0.02	1.48	1.50	1.52	1.53	1.56	2430.91	1
lp__	-7161.59	2.42	75.89	-7311.48	-7213.40	-7161.84	-7110.53	-7010.85	986.81	1

```
    z9[ii] ~ normal(U,sigma);

  }

  for (ii in 1:N8) {
    // utility
    vector[8] U = own8[ii]+alpha*other8[ii];


    z8[ii] ~ normal(U,sigma);
  }

  for (ii in 1:N7) {
    // utility
    vector[7] U = own7[ii]+alpha*other7[ii];

    z7[ii] ~ normal(U,sigma);
  }



  alpha ~ normal(prior_alpha[1],prior_alpha[2]);
  sigma ~ cauchy(0.0,prior_sigma);

}
```

As you can see, it is really an exercise in data-augmentation, and a bit of thought put in to how to pass the data to *Stan*. With regard to the latter, one of the quirks of the dataset is that there ranking lists are of different length. This is because some of the Alempaki et al. (2022) games have duplicate payoffs. So while there were nine outcomes of each game, in some games there were only seven or eight unique payoff profiles. I decided to handle these just as separate datasets, basically just splitting the full dataset into one that contained lists of length nine, one with lists of length eight, and one with lists of length seven. For the data-augmentation part, you can see this in the `model` block, for instance for the length-9 lists:

```
    vector[9] U = own9[ii]+alpha*other9[ii];

    z9[ii] ~ normal(U,sigma);
```

Here is *Stan*'s summary of the posterior simulation

```
summary("Code/ACKLP2022/estimates_FSranking_homogeneous.rds" |>
          readRDS())$summary |>
  data.frame() |>
  rownames_to_column(var = "parameter") |>
  kbl(digits=2) |>
  kable_classic(full_width=FALSE)
```

21.5 A hierarchical model

While going to a hierarchical version of this model in principle should be fairly straightforward, this proved to be a long hot walk in the sun. In particular, for many of the specifications I tried, I kept getting BFMI warnings from *Stan*. In the end, the model I got to run successfully without warnings did the following relative to what I usually do with hierarchical models:

1. I set the correlation matrix equal to the idenitiy matrix. That is, I assumed no correlation between α_i and σ_i
2. I used a half-normal prior for the standard deviation parameters rather than a half-Cauchy. This puts less mass in the tail of the distribution
3. I used much more informative priors than I usually would, especially for the standard deviation parameters.

Specifically, the hierarchical model specification is:

$$
\begin{aligned}
\alpha_i &\sim N(\mu_\alpha, \tau_\alpha^2) \\
\log \sigma_i &\sim N(\mu_\sigma, \tau_\sigma^2) \\
\text{hyper-priors: } \mu_\alpha &\sim N(0, 0.25) \\
\mu_\sigma &\sim N(0, 1) \\
\tau_\mu &\sim N^+(0, 0.01) \\
\tau_\sigma &\sim N^+(0, 0.01)
\end{aligned}
$$

Here is the *Stan* program that implements it:

```
data {

  int nparticipants;

  int<lower=0> N9; // total number of decisions with 9 unique payoffs
  int<lower=0> N8; // total number of decisions with 8 unique payoffs
  int<lower=0> N7; // total number of decisions with 7 unique payoffs

  int id9[N9];
  int id8[N8];
  int id7[N7];

  // payoffs from the game, ranked in ordered choice, worst to best
    // for the games with 9 uninque payoffs
    vector[9] own9[N9];
    vector[9] other9[N9];
    // for the games with 8 uninque payoffs
    vector[8] own8[N8];
    vector[8] other8[N8];
    // for the games with 9 uninque payoffs
    vector[7] own7[N7];
    vector[7] other7[N7];

    vector[2] prior_MU[2];
    vector[2] prior_TAU;
    real prior_OMEGA;
}
```

```
parameters {

  vector[2] MU;
  vector<lower=0>[2] TAU;
  //cholesky_factor_corr[2] L_OMEGA;

  matrix[2,nparticipants] z_pars;

  ordered[9] z9[N9];
  ordered[8] z8[N8];
  ordered[7] z7[N7];


}

transformed parameters {

  vector[nparticipants] alpha;
  vector<lower=0>[nparticipants] sigma;

  {

    matrix[2,nparticipants] pars = rep_matrix(MU,nparticipants)
      + //diag_pre_multiply(TAU, L_OMEGA) * z_pars;
      diag_matrix(TAU)*z_pars;

    alpha = pars[1,]';
    sigma = exp(pars[2,]');

  }

}

model {



  for (ii in 1:N9) {

    // utility
    vector[9] U = own9[ii]+alpha[id9[ii]]*other9[ii];

    z9[ii] ~ normal(U,sigma[id9[ii]]);

  }

  for (ii in 1:N8) {
    // utility
    vector[8] U = own8[ii]+alpha[id8[ii]]*other8[ii];


    z8[ii] ~ normal(U,sigma[id8[ii]]);
  }
```

par	mean	se_mean	sd	X2.5.	X25.	X50.	X75.	X97.5.	n_eff	Rhat
μ_α	0.15	0	0.01	0.12	0.14	0.15	0.15	0.17	2208.84	1
μ_σ	0.00	0	0.03	-0.05	-0.01	0.01	0.02	0.05	1612.34	1
τ_α	0.09	0	0.00	0.08	0.08	0.09	0.09	0.09	4569.19	1
τ_σ	0.16	0	0.01	0.15	0.16	0.16	0.17	0.17	2321.31	1

```
  for (ii in 1:N7) {
    // utility
    vector[7] U = own7[ii]+alpha[id7[ii]]*other7[ii];

    z7[ii] ~ normal(U,sigma[id7[ii]]);
  }


  // hierarchical structure

  to_vector(z_pars) ~ std_normal();

  // priors

  for (pp in 1:2) {

    MU[pp] ~ normal(prior_MU[pp][1],prior_MU[pp][2]);
    TAU ~ normal(0.0,prior_TAU[pp]);
  }

  //L_OMEGA ~ lkj_corr_cholesky(prior_OMEGA);
}
```

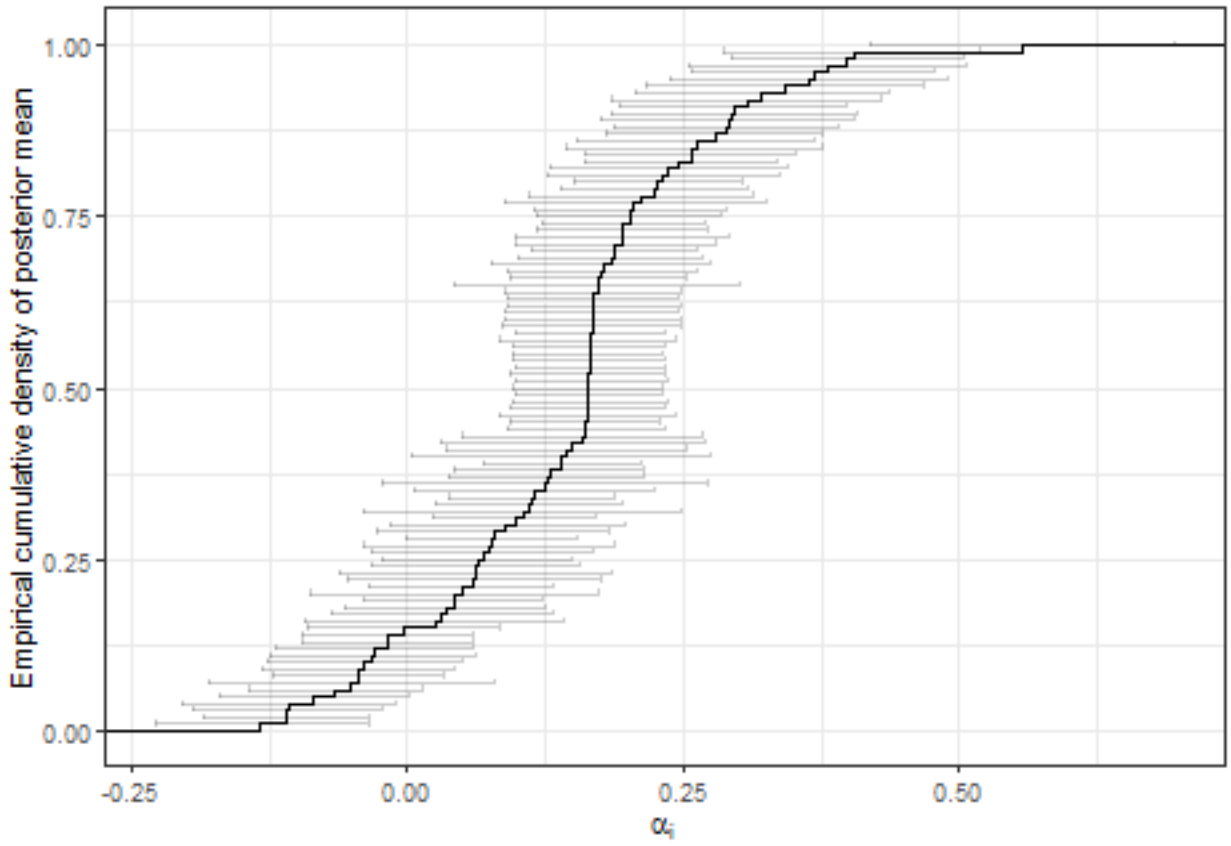
And here are the population-level estimates from it:

```
Fit<-summary("Code/ACKLP2022/estimates_linearranking_hierarchical.rds" |>
  readRDS())$summary |>
  data.frame() |>
  rownames_to_column(var = "par")

population<-Fit |>
  filter(grepl("MU",par) | grepl("TAU",par)) |>
  mutate(
    par = c("$\\mu_\\alpha$","$\\mu_\\sigma$","$\\tau_\\alpha$","$\\tau_\\sigma$")
  )

population |>
  kbl(digits = 2) |>
  kable_classic(full_width=FALSE)
```

and here are the shrinkage estimates for α

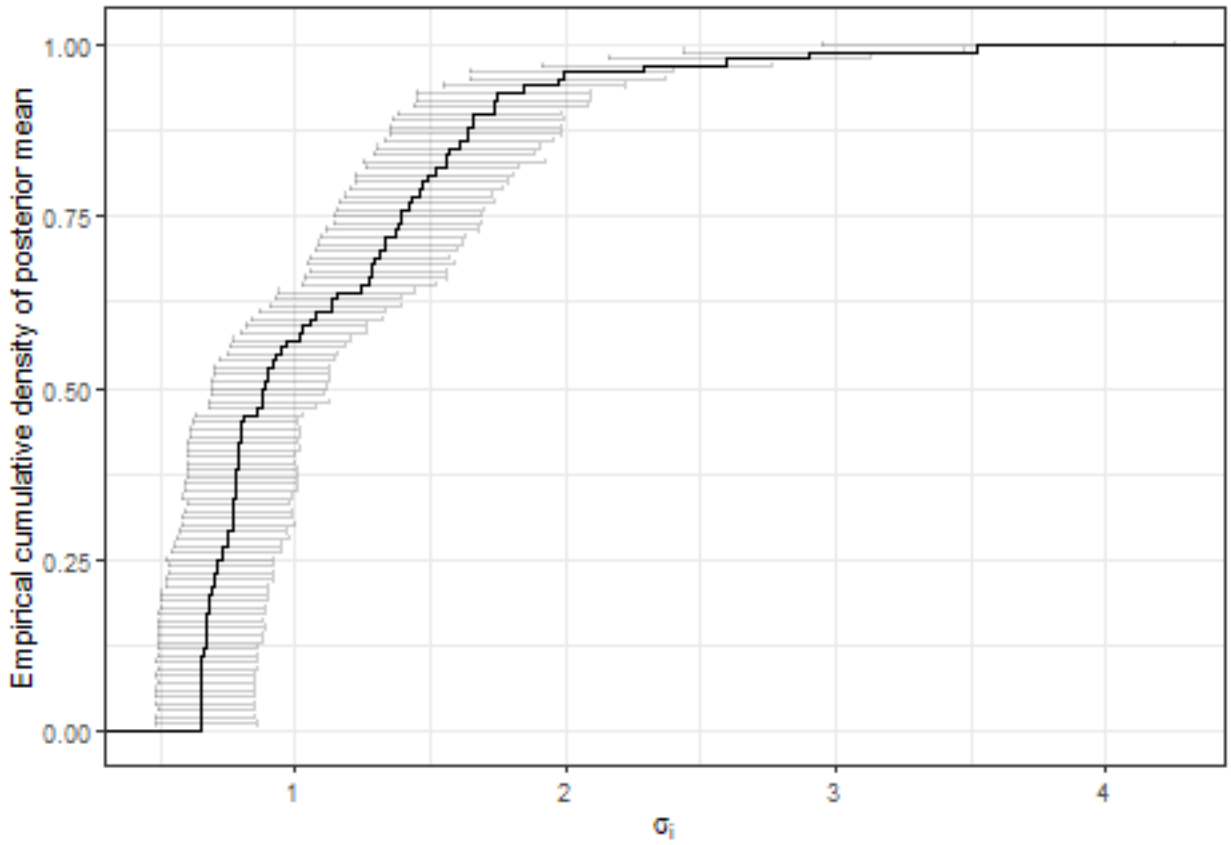
```
FitAlpha<-Fit |>
  filter(grepl("alpha",par)) |>
  mutate(
    cumul = rank(mean)/n()
  )
```

```
(
  ggplot(FitAlpha,aes(x=mean))
  +stat_ecdf()
  +theme_bw()
  +geom_errorbar(aes(y=cumul,xmin = X2.5.,xmax = X97.5.),alpha = 0.2)
  +xlab(expression(alpha[i]))
  +ylab("Empirical cumulative density of posterior mean")
)
```



and for σ:

```
FitAlpha<-Fit |>
  filter(grepl("sigma",par)) |>
  mutate(
    cumul = rank(mean)/n()
  )

(
  ggplot(FitAlpha,aes(x=mean))
  +stat_ecdf()
  +theme_bw()
  +geom_errorbar(aes(y=cumul,xmin = X2.5.,xmax = X97.5.),alpha = 0.2)
  +xlab(expression(sigma[i]))
  +ylab("Empirical cumulative density of posterior mean")
)
```



For perspective, if $\sigma_i = 1$ then a selfish person ranking two options with a dollar difference between them will rank them with the highest payoff first with probability:

$$\Phi\left(\frac{1}{\sqrt{2}}\right) \approx 0.76$$

21.6 *R* code used to run this

```
library(tidyverse)
library(haven)
library(rstan)
  rstan_options(auto_write = TRUE)
  options(mc.cores = parallel::detectCores())



ranking<-"Data/ACKLP2022_ranking.dta" |>
  read_dta() |>
  mutate(uid = paste("id",id) |> as.factor() |> as.numeric()) |>
  arrange(id,game,-ranking)
```

```
d<-ranking |>
  group_by(id,game) |>
  mutate(
    nuniquepayoffs = n(),
    idg = paste0(id,"-",game),
    k = 1:n()
  )

own9<-list()
other9<-list()

own8<-list()
other8<-list()

own7<-list()
other7<-list()

for (gg in (d$idg |> unique())) {


  dg<-d |> filter(idg==gg)

  if (dim(dg)[1]==9) {

    own9[[paste(gg)]]<-dg$own |> unlist() |> as.vector()
    other9[[paste(gg)]]<-dg$other |> unlist() |> as.vector()

  } else if (dim(dg)[1]==8) {
    own8[[paste(gg)]]<-dg$own |> unlist() |> as.vector()
    other8[[paste(gg)]]<-dg$other |> unlist() |> as.vector()
  } else {
    own7[[paste(gg)]]<-dg$own |> unlist() |> as.vector()
    other7[[paste(gg)]]<-dg$other |> unlist() |> as.vector()
  }
}


#------------------------------------------------------------------------------
# Pooled model
#------------------------------------------------------------------------------

model<-"Code/ACKLP2022/linearranking_homogeneous.stan" |>
  stan_model()



dStan<-list(

  nparticipants = d$uid |> unique() |> length(),

  N9 = length(own9),
```

```
  N8 = length(own8),
  N7 = length(own7),

  id9 = (d |> filter(nuniquepayoffs==9 & k==1))$uid,
  id8 = (d |> filter(nuniquepayoffs==8 & k==1))$uid,
  id7 = (d |> filter(nuniquepayoffs==7 & k==1))$uid,

  own9 = own9,
  other9=other9,
  own8 = own8,
  other8=other8,
  own7 = own7,
  other7=other7,

  prior_alpha = c(0,1),
  prior_sigma = 1
)

Fit <- model |>
  sampling(
    data=dStan,
    #chains=4,iter=10000,
    seed=43,
    pars = c("z9","z8","z7"),include=FALSE
  )

Fit |>
  saveRDS("Code/ACKLP2022/estimates_FSranking_homogeneous.rds")



################################################################################
# Hierarchical model
################################################################################

model<-"Code/ACKLP2022/linearranking_hierarchical.stan" |>
  stan_model()

dStan<-list(

  nparticipants = d$uid |> unique() |> length(),

  N9 = length(own9),
  N8 = length(own8),
  N7 = length(own7),

  id9 = (d |> filter(nuniquepayoffs==9 & k==1))$uid,
  id8 = (d |> filter(nuniquepayoffs==8 & k==1))$uid,
  id7 = (d |> filter(nuniquepayoffs==7 & k==1))$uid,

  own9 = own9,
  other9=other9,
  own8 = own8,
```

```
  other8=other8,
  own7 = own7,
  other7=other7,

  prior_MU = list(c(0,0.25),
                #  c(1,0.25),
                  c(0,1)
                  ),
  prior_TAU = c(0.01,0.01),
  prior_OMEGA = 4
)


Fit <- model |>
  sampling(
    data=dStan,
    chains=8,iter=2000,
    seed=42,
    pars = c("z_pars","z9","z8","z7"),include=FALSE
  )

Fit |>
  saveRDS("Code/ACKLP2022/estimates_linearranking_hierarchical.rds")
```

Links to data

Links to the data that are freely available from other sources:

- Anwar and Georgalos (2024): https://osf.io/br5uy/
- Alempaki et al. (2022): https://osf.io/zhufe/
- Bland (2019a): https://www.sciencedirect.com/science/article/pii/S2214804318303616#ec-research-data
- Bochet and Magnani (2024): https://www.aeaweb.org/articles?id=10.1257/mic.20210344
- Bruhin, Fehr, and Schunk (2019): https://academic.oup.com/jeea/article/17/4/1025/5001317?guestAccessKey=005c6e7e-70c9-4946-bc13-095f35424669&login=false#supplementary-data
- Costa-Gomes and Crawford (2006): https://www.aeaweb.org/articles?id=10.1257/aer.96.5.1737
- Dal Bó and Fréchette (2011): https://www.aeaweb.org/articles?id=10.1257/aer.101.1.411
- Ellis and Freeman (2024): https://www.aeaweb.org/articles?id=10.1257/aer.20210877
- Halevy, Persitz, and Zrill (2018): https://github.com/persitzd/RP-Toolkit/tree/master/Data%20Files
  - Use `Data_HPD_2018.csv`
- Harrison and Swarthout (2023):
  - Data available on Glenn Harrison's website here: http://cear.gsu.edu/gwh/
  - I am using `GSUData.dta` from the "house money" folder
- METARET:
  - https://paolocrosetto.shinyapps.io/METARET_APP/
  - Click on the "Download" tab, then "Download all task data"

Links to data that are available in this book's GitHub data repository

- Cason, Friedman, and Hopkins (2010): https://github.com/JamesBlandEcon/BayesForExperimentsData/tree/main/CFH2010TASP

References

Alempaki, Despoina, Andrew M Colman, Felix Kölle, Graham Loomes, and Briony D Pulford. 2022. "Investigating the Failure to Best Respond in Experimental Games." *Experimental Economics*, 1–24.

Andreoni, James, and John Miller. 2002. "Giving According to GARP: An Experimental Test of the Consistency of Preferences for Altruism." *Econometrica* 70 (2): 737–53. https://www.jstor.org/stable/2692289.

Andreoni, James, and Lise Vesterlund. 2001. "Which Is the Fair Sex? Gender Differences in Altruism." *The Quarterly Journal of Economics* 116 (1): 293–312. https://doi.org/10.1162/003355301556419.

Anwar, Chowdhury Mohammad Sakib, and Konstantinos Georgalos. 2024. "Position Uncertainty in a Sequential Public Goods Game: An Experiment." *Experimental Economics*, 1–34.

Barberis, Nicholas, and Ming Huang. 2009. "Preferences with Frames: A New Utility Specification That Allows for the Framing of Risks." *Journal of Economic Dynamics and Control* 33 (8): 1555–76.

Barberis, Nicholas, Ming Huang, and Richard H Thaler. 2006. "Individual Preferences, Monetary Gambles, and Stock Market Participation: A Case for Narrow Framing." *American Economic Review* 96 (4): 1069–90.

Betancourt, Michael. 2020. "Towards a Principled Bayesian Workflow." https://betanalpha.github.io/assets/case_studies/principled_bayesian_workflow.html.

Betancourt, Michael, and Mark Girolami. 2015. "Hamiltonian Monte Carlo for Hierarchical Models." *Current Trends in Bayesian Methodology with Applications* 79 (30): 2–4. https://doi.org/10.48550/arXiv.1312.0906.

Bland, James R. 2019a. "How Many Games Are We Playing? An Experimental Analysis of Choice Bracketing in Games." *Journal of Behavioral and Experimental Economics* 80: 80–91. https://doi.org/10.1016/j.socec.2019.03.011.

———. 2019b. "Measuring and Comparing Two Kinds of Rationalizable Opportunity Cost in Mixture Models." *Games* 11 (1): 1.

———. 2020. "Heterogeneous Trembles and Model Selection in the Strategy Frequency Estimation Method." *Journal of the Economic Science Association* 6 (2): 113–24.

———. 2022. "Bayesian Inference for Quantal Response Equilibrium in Normal-Form Games." *Available at SSRN 3748586*. https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3748586.

———. 2023a. "Bayesian Inference for Quantal Response Equilibrium in Normal-Form Games." *Games and Economic Behavior*. https://doi.org/https://doi.org/10.1016/j.geb.2023.05.005.

———. 2023b. "Bayesian Model Selection and Prior Calibration for Models of Behavior in Economic Experiments." https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4334267.

———. 2023c. "Computing Quantal Response Equilibrium in Some Bayesian Games Using Marginal Strategy Profiles." *Available at SSRN 4551341*.

Bland, James R, and Yaroslav Rosokha. 2019. "Learning Under Uncertainty with Multiple Priors: Experimental Investigation." *Available at SSRN 3419020*.

Bland, James R, and Theodore L Turocy. 2025. "Quantal Response Equilibrium as a Structural Model for Estimation: The Missing Manual." *Games and Economic Behavior*.

Blavatskyy, Pavlo, Valentyn Panchenko, and Andreas Ortmann. 2023. "How Common Is the Common-Ratio Effect?" *Experimental Economics* 26 (2): 253–72.

Bochet, Olivier, and Jacopo Magnani. 2024. "Limited Strategic Thinking and the Cursed Match." *American Economic Journal: Microeconomics* 16 (3): 321–44. https://doi.org/10.1257/mic.20210344.

Bruhin, Adrian, Ernst Fehr, and Daniel Schunk. 2019. "The Many Faces of Human Sociality: Uncovering the Distribution and Stability of Social Preferences." *Journal of the European Economic Association* 17 (4): 1025–69.

Camerer, Colin, and Teck Hua Ho. 1999. "Experience-Weighted Attraction Learning in Normal Form Games." *Econometrica* 67 (4): 827–74.

Cason, Timothy N., Daniel Friedman, and Ed Hopkins. 2010. "Testing the TASP: An Experimental

Investigation of Learning in Games with Unstable Equilibria." *Journal of Economic Theory* 145 (6): 2309–31. https://doi.org/http://dx.doi.org/10.1016/j.jet.2010.08.011.

Conte, Anna, John D Hey, and Peter G Moffatt. 2011. "Mixture Models of Choice Under Risk." *Journal of Econometrics* 162 (1): 79–88.

Costa-Gomes, Miguel A, and Vincent P Crawford. 2006. "Cognition and Behavior in Two-Person Guessing Games: An Experimental Study." *American Economic Review* 96 (5): 1737–68.

Dal Bó, Pedro, and Guillaume R Fréchette. 2011. "The Evolution of Cooperation in Infinitely Repeated Games: Experimental Evidence." *American Economic Review* 101 (1): 411–29.

———. 2019. "Strategy Choice in the Infinitely Repeated Prisoner's Dilemma." *American Economic Review* 109 (11): 3929–52.

Diekmann, Andreas. 1985. "Volunteer's Dilemma." *Journal of Conflict Resolution* 29 (4): 605–10.

Ellis, Andrew, and David J. Freeman. 2024. "Revealing Choice Bracketing." *American Economic Review* 114 (9): 2668–2700. https://doi.org/10.1257/aer.20210877.

Eyster, Erik, and Matthew Rabin. 2005. "Cursed Equilibrium." *Econometrica* 73 (5): 1623–72.

Fehr, Ernst, and Klaus M Schmidt. 1999. "A Theory of Fairness, Competition, and Cooperation." *The Quarterly Journal of Economics* 114 (3): 817–68.

Fudenberg, Drew, David G Rand, and Anna Dreber. 2012. "Slow to Anger and Fast to Forgive: Cooperation in an Uncertain World." *American Economic Review* 102 (2): 720–49.

Gallice, Andrea, and Ignacio Monzón. 2019. "Co-Operation in Social Dilemmas Through Position Uncertainty." *The Economic Journal* 129 (621): 2137–54.

Gao, Xiaoxue Sherry, Glenn W Harrison, and Rusty Tchernis. 2022. "Behavioral Welfare Economics and Risk Preferences: A Bayesian Approach." *Experimental Economics*, 1–31. https://doi.org/10.1007/s10683-022-09751-0.

Goeree, Jacob K, Charles A Holt, and Susan K Laury. 2002. "Private Costs and Public Benefits: Unraveling the Effects of Altruism and Noisy Behavior." *Journal of Public Economics* 83 (2): 255–76.

Goeree, Jacob K, Charles A Holt, and Angela M Smith. 2017. "An Experimental Examination of the Volunteer's Dilemma." *Games and Economic Behavior* 102: 303–15.

Gronau, Quentin F., Alexandra Sarafoglou, Dora Matzke, Alexander Ly, Udo Boehm, Maarten Marsman, David S. Leslie, Jonathan J. Forster, Eric-Jan Wagenmakers, and Helen Steingroever. 2017. "A Tutorial on Bridge Sampling." *Journal of Mathematical Psychology* 81: 80–97. https://doi.org/https://doi.org/10.1016/j.jmp.2017.09.005.

Gronau, Quentin F., Henrik Singmann, and Eric-Jan Wagenmakers. 2020. "Bridgesampling: An r Package for Estimating Normalizing Constants." *Journal of Statistical Software* 92: 1–29.

Halevy, Yoram, Dotan Persitz, and Lanny Zrill. 2018. "Parametric Recoverability of Preferences." *Journal of Political Economy* 126 (4): 1558–93. https://doi.org/10.1086/697741.

Harrison, Glenn W, and Jia Min Ng. 2016. "Evaluating the Expected Welfare Gain from Insurance." *Journal of Risk and Insurance* 83 (1): 91–120. https://doi.org/10.1111/jori.12142.

Harrison, Glenn W, and E Elisabet Rutström. 2009. "Expected Utility Theory and Prospect Theory: One Wedding and a Decent Funeral." *Experimental Economics* 12 (2): 133–58.

Harrison, Glenn W, and Todd Swarthout. 2023. "Cumulative Prospect Theory in the Laboratory: A Reconsideration." In *Research in Experimental Economics: Models of Risk Preferences: Descriptive and Normative Challenges*, edited by G. W. Harrison and D. Ross. Bingley, UK: Emerald.

Hey, John D. 2001. "Does Repetition Improve Consistency?" *Experimental Economics* 4 (1): 5–54.

Hey, John D, and Daniela Di Cagno. 1990. "Circles and Triangles an Experimental Estimation of Indifference Lines in the Marschak-Machina Triangle." *Journal of Behavioral Decision Making* 3 (4): 279–305.

Hey, John D, and Chris Orme. 1994. "Investigating Generalizations of Expected Utility Theory Using Experimental Data." *Econometrica: Journal of the Econometric Society*, 1291–1326.

Holt, Charles A, and Susan K Laury. 2002. "Risk Aversion and Incentive Effects." *American Economic Review* 92 (5): 1644–55.

Holt, Charles A, Ricky Sahu, and Angela M Smith. 2022. "An Experimental Analysis of Risk Effects in Attacker-Defender Games." *Southern Economic Journal* 89 (1): 185–215.

McKelvey, Richard D, Andrew M McLennan, and Theodore L Turocy. 2022. "Gambit: Software Tools for Game Theory." http://www.gambit-project.org/.

McKelvey, Richard D, and Thomas R Palfrey. 1995. "Quantal Response Equilibria for Normal Form Games."

Games and Economic Behavior 10 (1): 6–38. https://doi.org/10.1006/game.1995.1023.

Meng, Xiao-Li, and Wing Hung Wong. 1996. "Simulating Ratios of Normalizing Constants via a Simple Identity: A Theoretical Exploration." *Statistica Sinica*, 831–60.

Moffatt, Peter G. 2015. *Experimetrics: Econometrics for Experimental Economics*. Palgrave Macmillan.

Monroe, Brian Albert. 2020. "The Statistical Power of Individual-Level Risk Preference Estimation." *Journal of the Economic Science Association* 6 (2): 168–88.

Ochs, Jack. 1995. "Games with Unique, Mixed Strategy Equilibria: An Experimental Study." *Games and Economic Behavior* 10 (1): 202–17.

Prelec, Drazen. 1998. "The Probability Weighting Function." *Econometrica*, 497–527.

Quiggin, John. 1982. "A Theory of Anticipated Utility." *Journal of Economic Behavior & Organization* 3 (4): 323–43.

Rabin, Matthew, and Georg Weizsäcker. 2009. "Narrow Bracketing and Dominated Choices." *The American Economic Review* 99 (4): 1508–43.

Romero, Julian, and Yaroslav Rosokha. 2018. "Constructing Strategies in the Indefinitely Repeated Prisoner's Dilemma Game." *European Economic Review* 104: 185–219.

Stahl, Dale O. 2018. "Assessing the Forecast Performance of Models of Choice." *Journal of Behavioral and Experimental Economics* 73: 86–92.

Stahl, Dale O, and Paul W Wilson. 1994. "Experimental Evidence on Players' Models of Other Players." *Journal of Economic Behavior & Organization* 25 (3): 309–27.

Stan Development Team. 2022. "Stan User's Guide, Version 2.31."

Turocy, Theodore L. 2005. "A Dynamic Homotopy Interpretation of the Logistic Quantal Response Equilibrium Correspondence." *Games and Economic Behavior* 51 (2): 243–63.

———. 2010. "Computing Sequential Equilibria Using Agent Quantal Response Equilibria." *Economic Theory* 42: 255–69.

Vehtari, Aki, Andrew Gelman, and Jonah Gabry. 2017. "Practical Bayesian Model Evaluation Using Leave-One-Out Cross-Validation and WAIC." *Statistics and Computing* 27 (5): 1413–32.

Wilcox, Nathaniel T. 2006. "Theories of Learning in Games and Heterogeneity Bias." *Econometrica* 74 (5): 1271–92.

———. 2011. "'Stochastically More Risk Averse:'a Contextual Theory of Stochastic Discrete Choice Under Risk." *Journal of Econometrics* 162 (1): 89–104.