EKW lectures

Prof. Dr. Philipp Eisenhauer

Oct 07, 2021

CONTENTS

This repository contains several lectures that use the `respy` package to solve, simulate, and estimate Eckstein-Keane-Wolpin models. These support our educational activities around our group's research code. We provide an accompanying handout and presentation for this class of models here.

CHAPTER

ONE

OVERVIEW

[1]:
```
import sys
import respy as rp
import numpy as np
sys.path.insert(0, "python")
from auxiliary import plot_observed_choices  # noqa: E402
from auxiliary import plot_time_preference  # noqa: E402
from auxiliary import plot_policy_forecast  # noqa: E402
```

1.1 Structural microeconometrics

Computational modeling in economics

- provide learning opportunities
- assess importance of competing mechanisms
- predict the effects of public policies

Eckstein-Keane-Wolpin (EKW) models}

- understanding individual decisions
  - human capital investment
  - savings and retirement
- predicting effects of policies
  - welfare programs
  - tax schedules

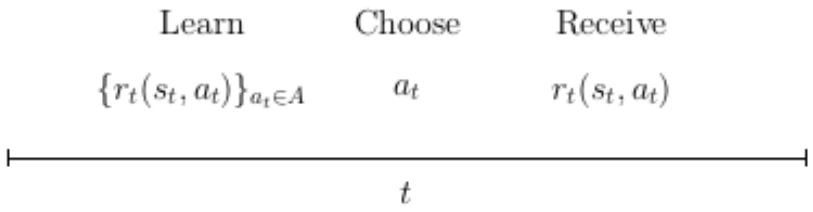
Components

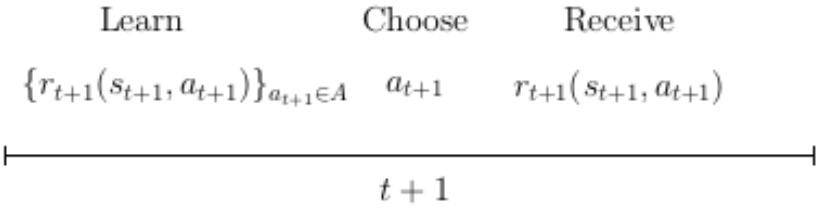
- economic model
- mathematical formulation
- calibration procedure

1.1.1 Economic model

Decision problem

$t = 1, .., T$	decision period
$s_t \in S$	state
$a_t \in A$	action
$a_t(s_t)$	decision rule
$r_t(s_t, a_t)$	immediate reward





$\pi = (a_1^\pi(s_1), .., a_T^\pi(s_T))$	policy
δ	discount factor
$p_t(s_t, a_t)$	conditional distribution

Individual's objective

$$\max_{\pi \in \Pi} E_{s_1}^\pi \left[\sum_{t=1}^{T} \delta^{t-1} r_t(s_t, a_t^\pi(s_t)) \,\middle|\, \mathcal{I}_1 \right]$$

1.1.2 Mathematical formulation

Policy evaluation

$$v_t^\pi(s_t) \equiv E_{s_t}^\pi \left[\sum_{j=0}^{T-t} \delta^j \, r_{t+j}(s_{t+j}, a_{t+j}^\pi(s_{t+j})) \,\middle|\, \mathcal{I}_t \right]$$

Inductive scheme

$$v_t^\pi(s_t) = r_t(s_t, a_t^\pi(s_t)) + \delta \, E_{s_t}^\pi \left[v_{t+1}^\pi(s_{t+1}) \,\middle|\, \mathcal{I}_t \right]$$

Optimality equations

$$v_t^{\pi^*}(s_t) = \max_{a_t \in A} \left\{ r_t(s_t, a_t) + \delta \, E_{s_t}^{\pi^*} \left[v_{t+1}^{\pi^*}(s_{t+1}) \,\middle|\, \mathcal{I}_t \right] \right\}$$

Algorithm 1 Backward induction procedure

for $t = T, \ldots, 1$ **do**

  if t == T **then**

    $$v_T^{\pi^*}(s_T) = \max_{a_T \in A} \left\{ r_T(s_T, a_T) \right\} \qquad \forall\, s_T \in S$$

  else

    Compute $v_t^{\pi^*}(s_t)$ for each $s_t \in S$ by

    $$v_t^{\pi^*}(s_t) = \max_{a_t \in A} \left\{ r_t(s_t, a_t) + \delta\, \mathrm{E}_{s_t}^{\pi} \left[v_{t+1}^{\pi^*}(s_{t+1}) \middle| \mathcal{I}_t \right] \right\}$$

    and set

    $$a_t^{\pi^*}(s_t) = \arg\max_{a_t \in A} \left\{ r_t(s_t, a_t) + \delta\, \mathrm{E}_{s_t}^{\pi} \left[v_{t+1}^{\pi^*}(s_{t+1}) \middle| \mathcal{I}_t \right] \right\}.$$

  end if

end for

1.1.3 Calibration procedure

Data

**

- $s_t = (x_t, \epsilon_t)$
  - x_t observed
  - ϵ_t unobserved

Procedures

- likelihood-based

$$\hat{\theta} \equiv argmax_{\theta \in \Theta} \prod_{i=1}^{N} \prod_{t=1}^{T_i} p_{it}(a_{it}, r_{it} \mid x_{it}, \theta)$$

- simulation-based

$$\hat{\theta} \equiv argmin_{\theta \in \Theta} (M_D - M_S(\theta))' W (M_D - M_S(\theta))$$

1.1.4 Example

Seminal paper

- Keane, M. P., & Wolpin, K. I. (1994). The solution and estimation of discrete choice dynamic programming models by simulation and interpolation: Monte Carlo evidence. Review of Economics and Statistics, 76 (4), 648–672.

Model of occupational choice

- 1,000 individuals starting at age 16
- life cycle histories
  - school attendance
  - occupation-specific work status
- wages

Labor market

$$r_t(s_t, 1) = w_{1t} = \exp\{ \underbrace{\alpha_{10}}_{\text{endowment}} + \underbrace{\alpha_{11} g_t}_{\text{schooling}} + \underbrace{\alpha_{12} e_{1t} + \alpha_{13} e_{1t}^2}_{\text{own experience}} + \underbrace{\alpha_{14} e_{2t} + \alpha_{15} e_{2t}^2}_{\text{other experience}} + \underbrace{\epsilon_{1t}}_{\text{shock}} \}$$

The same setup applies to the second occupation.

Schooling

$$r_t(s_t, 3) = \underbrace{\beta_0}_{\text{taste}} - \underbrace{\beta_1 I[\, g_t \geq 12 \,]}_{\text{direct cost}} - \underbrace{\beta_2 I[\, a_{t-1} \neq 3 \,]}_{\text{reenrollment effort}} + \underbrace{\epsilon_{3t}}_{\text{shock}}$$

Home

$$r_t(s_t, 4) = \underbrace{\gamma_0}_{\text{taste}} + \underbrace{\epsilon_{4t}}_{\text{shock}}$$

State space

$$s_t = \{g_t, e_{1t}, e_{2t}, a_{t-1}, \epsilon_{1t}, \epsilon_{2t}, \epsilon_{3t}, \epsilon_{4t}\}$$

Transitions

- observed state variables

$$\begin{aligned} e_{1,t+1} &= e_{1t} + I[\, a_t = 1 \,] \\ e_{2,t+1} &= e_{2t} + I[\, a_t = 2 \,] \\ g_{t+1} &= g_t \ + I[\, a_t = 3 \,] \end{aligned}$$

- unobserved state variables

$$\{\epsilon_{1t}, \epsilon_{2t}, \epsilon_{3t}, \epsilon_{4t}\} \sim N(0, \Sigma)$$

[2]:
```
params, options = rp.get_example_model("kw_94_two", with_data=False)
```

How is the economy parametrized?

[3]:
```
params.head()
```

[3]:
```
                         value                                        comment
category name
delta    delta          0.9500                                discount factor
wage_a   constant       9.2100                           log of rental price
         exp_edu        0.0400    return to an additional year of schooling
         exp_a          0.0330              return to same sector experience
         exp_a_square  -0.0005  return to same sector, quadratic experience
```

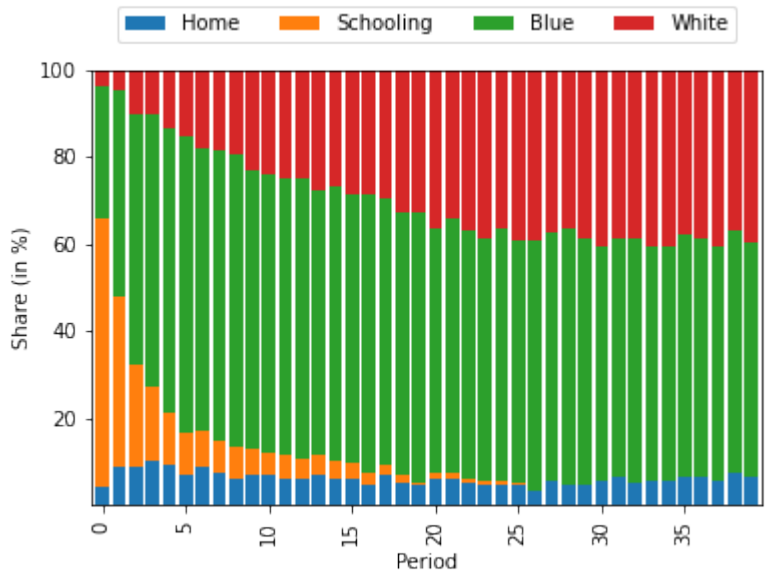
How are the options set?

[4]:
```
options
```

[4]:
```
{'estimation_draws': 200,
 'estimation_seed': 500,
 'estimation_tau': 500,
 'interpolation_points': -1,
 'n_periods': 40,
 'simulation_agents': 1000,
 'simulation_seed': 132,
 'solution_draws': 500,
 'solution_seed': 1,
 'monte_carlo_sequence': 'random',
 'core_state_space_filters': ["period > 0 and exp_{choices_w_exp} == period and lagged_
↪choice_1 != '{choices_w_exp}'",
  "period > 0 and exp_a + exp_b + exp_edu == period and lagged_choice_1 == '{choices_wo_
↪exp}'",
  "period > 0 and lagged_choice_1 == 'edu' and exp_edu == 0",
  "lagged_choice_1 == '{choices_w_wage}' and exp_{choices_w_wage} == 0",
  "period == 0 and lagged_choice_1 == '{choices_w_wage}'"],
 'covariates': {'constant': '1',
  'exp_a_square': 'exp_a ** 2',
  'exp_b_square': 'exp_b ** 2',
  'at_least_twelve_exp_edu': 'exp_edu >= 12',
  'not_edu_last_period': "lagged_choice_1 != 'edu'"}}
```

We can now simulate a dataset and look at the individual decisions.

[5]:
```
simulate_func = rp.get_simulate_func(params, options)
plot_observed_choices(simulate_func(params))
```

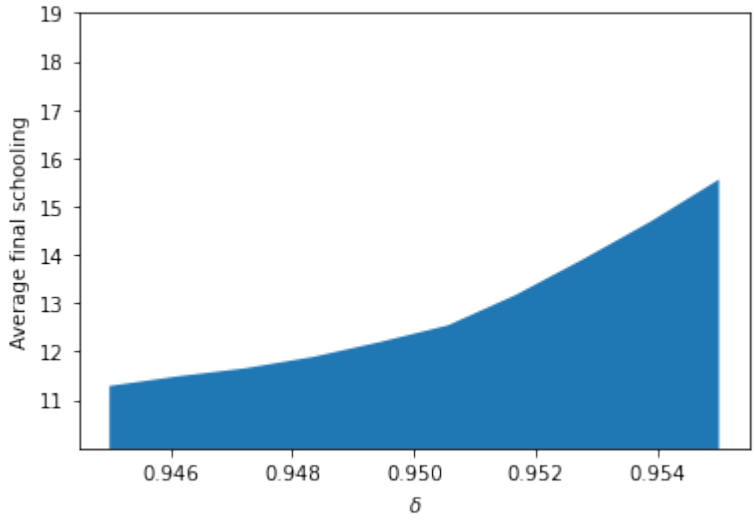


Mechanisms

[6]:
```
def time_preference_wrapper_kw_94(simulate_func, params, value):
    policy_params = params.copy()
    policy_params.loc[("delta", "delta"), "value"] = value
    policy_df = simulate_func(policy_params)

    edu = policy_df.groupby("Identifier")["Experience_Edu"].max().mean()

    return edu
```

Now we can iterate over a grid of discount factors.

[7]:
```
deltas = np.linspace(0.945, 0.955, 10)
edu_level = list()

for i, delta in enumerate(deltas):
    stat = time_preference_wrapper_kw_94(simulate_func, params, delta)
    edu_level.append(stat)

plot_time_preference(deltas, edu_level)
```

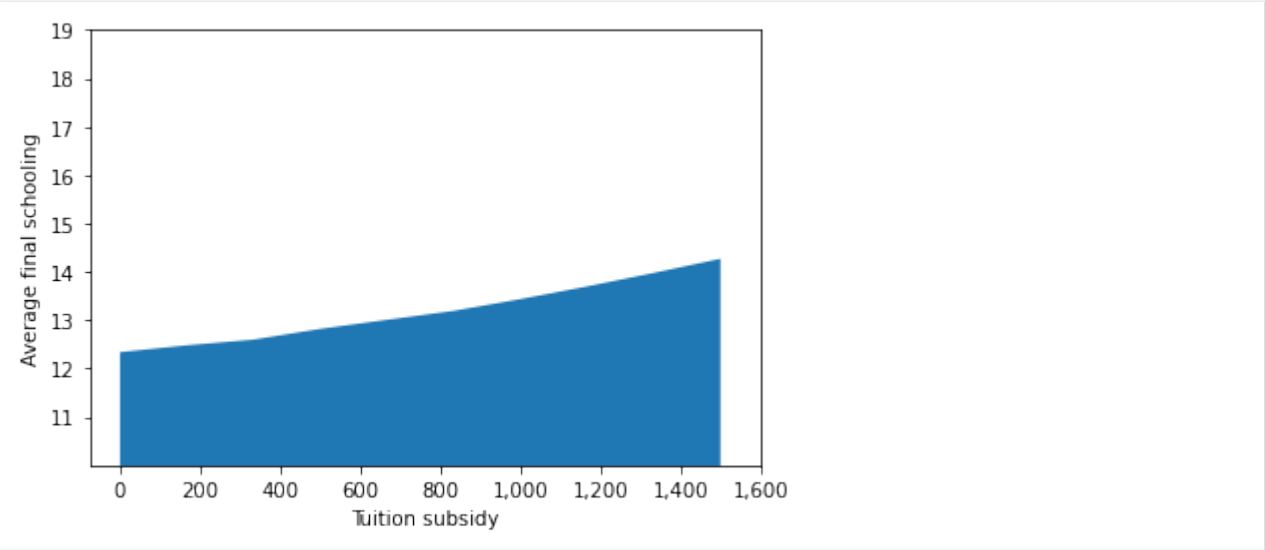


Policy forecast

[8]:
```
def tuition_policy_wrapper_kw_94(simulate_func, params, tuition_subsidy):
    policy_params = params.copy()
    policy_params.loc[("nonpec_edu", "at_least_twelve_exp_edu"), "value"] += tuition_
↪subsidy
    policy_df = simulate_func(policy_params)

    edu = policy_df.groupby("Identifier")["Experience_Edu"].max().mean()

    return edu
```

Now we can iterate over a grid of tuition subsidies.

[9]:
```
subsidies = np.linspace(0, 1500, num=10, dtype=int, endpoint=True)
edu_level = list()

for i, subsidy in enumerate(subsidies):
    stat = tuition_policy_wrapper_kw_94(simulate_func, params, subsidy)
    edu_level.append(stat)

plot_policy_forecast(subsidies, edu_level)
```



We discuss the basic motivation behind structural econometric modeling. We quickly focus on the class of Eckstein-Keane-Wolpin models that are often used in labor economics to study human capital accumulation and discuss their economic model, mathematical formulation, and alternative calibration procedures. We explore the seminal work outlined in Keane & Wolpin (1994) as an example.

CHAPTER

TWO

CALIBRATION

2.1 Maximum likelihood

[1]:
```
from timeit import default_timer as timer
from functools import partial
import yaml
import sys

from estimagic import maximize
from scipy.optimize import root_scalar
from scipy.stats import chi2
import numdifftools as nd
import pandas as pd
import respy as rp
import numpy as np

sys.path.insert(0, "python")
from auxiliary import plot_bootstrap_distribution  # noqa: E402
from auxiliary import plot_computational_budget  # noqa: E402
from auxiliary import plot_smoothing_parameter  # noqa: E402
from auxiliary import plot_score_distribution  # noqa: E402
from auxiliary import plot_score_function  # noqa: E402
from auxiliary import plot_likelihood  # noqa: E402
```

2.1.1 Maximum likelihood estimation

Introduction

EKW models are calibrated to data on observed individual decisions and experiences under the hypothesis that the individual's behavior is generated from the solution to the model. The goal is to back out information on reward functions, preference parameters, and transition probabilities. This requires the full parameterization θ of the model.

Economists have access to information for $i = 1, ..., N$ individuals in each time period t. For every observation (i, t) in the data, we observe action a_{it}, reward r_{it}, and a subset x_{it} of the state s_{it}. Therefore, from an economist's point of view, we need to distinguish between two types of state variables $s_{it} = (x_{it}, \epsilon_{it})$. At time t, the economist and individual both observe x_{it} while ϵ_{it} is only observed by the individual. In summary, the data $\mathcal{D}$ has the following structure:

$$\mathcal{D} = \{a_{it}, x_{it}, r_{it} : i = 1, ..., N; t = 1, ..., T_i\},$$

where T_i is the number of observations for which we observe individual i.

Likelihood-based calibration seeks to find the parameterization $\hat{\theta}$ that maximizes the likelihood function $\mathcal{L}(\theta \mid \mathcal{D})$, i.e. the probability of observing the given data as a function of θ. As we only observe a subset x_t of the state, we can determine the probability $p_{it}(a_{it}, r_{it} \mid x_{it}, \theta)$ of individual i at time t in x_{it} choosing a_{it} and receiving r_{it} given parametric assumptions about the distribution of ϵ_{it}. The objective function takes the following form:

$$\hat{\theta} \equiv \text{argmax}\theta \in \Theta \underbrace{\prod_{i=1}^{N} \prod_{t=1}^{T_i} p_{it}(a_{it}, r_{it} \mid x_{it}, \theta)}_{\mathcal{L}(\theta|\mathcal{D})} .$$

We will explore the following issues:

- likelihood function
- score function and statistic
  - asymptotic distribution
  - linearity
- confidence intervals
  - Wald
  - likelihood - based
  - Bootstrap
- numerical approximations
  - smoothing of choice probabilities
  - grid search

Most of the material is from the following two references:

- Pawitan, Y. (2001). In all likelihood: Statistical modelling and inference using likelihood. Clarendon Press, Oxford.
- Casella, G., & Berger, R. L. (2002). Statistical inference. Duxbury, Belmont, CA.

Let's get started!

[2]:
```
options_base = yaml.safe_load(open("../../configurations/robinson/robinson.yaml", "r"))

params_base = pd.read_csv(open("../../configurations/robinson/robinson.csv", "r"))
params_base.set_index(["category", "name"], inplace=True)

simulate = rp.get_simulate_func(params_base, options_base)
df = simulate(params_base)
```

Let us briefly inspect the parameterization.

[3]:
```
params_base
```

[3]:

		value
category	name	
delta	delta	0.950
wage_fishing	exp_fishing	0.070
nonpec_fishing	constant	-0.100
nonpec_hammock	constant	1.046
shocks_sdcorr	sd_fishing	0.010

(continues on next page)

(continued from previous page)

```
                sd_hammock            0.010
                corr_hammock_fishing  0.000
```

Several options need to be specified as well.

[4]:
```
options_base
```

[4]:
```
{'estimation_draws': 100,
 'estimation_seed': 100,
 'estimation_tau': 0.001,
 'interpolation_points': -1,
 'n_periods': 5,
 'simulation_agents': 1000,
 'simulation_seed': 132,
 'solution_draws': 100,
 'solution_seed': 456,
 'covariates': {'constant': '1'}}
```

We can now look at the simulated dataset.

[5]:
```
df.head()
```

[5]:
```
                    Experience_Fishing  Shock_Reward_Fishing  \
Identifier Period
0          0                         0              0.999650
           1                         1              1.000743
           2                         2              0.996461
           3                         3              0.998907
           4                         4              0.989419

                    Meas_Error_Wage_Fishing  Shock_Reward_Hammock  \
Identifier Period
0          0                             1              0.000410
           1                             1              0.015065
           2                             1              0.011853
           3                             1             -0.007859
           4                             1              0.012452

                    Meas_Error_Wage_Hammock  Dense_Key  Core_Index   Choice  \
Identifier Period
0          0                             1          0           0  fishing
           1                             1          1           1  fishing
           2                             1          2           2  fishing
           3                             1          3           3  fishing
           4                             1          4           4  fishing

                        Wage  Discount_Rate  ...  Nonpecuniary_Reward_Fishing  \
Identifier Period                            ...
0          0       0.999650           0.95  ...                         -0.1
           1       1.073305           0.95  ...                         -0.1
           2       1.146203           0.95  ...                         -0.1
           3       1.232329           0.95  ...                         -0.1
           4       1.309130           0.95  ...                         -0.1
```

(continues on next page)

(continued from previous page)

```
                    Wage_Fishing  Flow_Utility_Fishing  Value_Function_Fishing  \
Identifier Period
0          0            0.999650              0.899650                4.739374
           1            1.073305              0.973305                4.042189
           2            1.146203              1.046203                3.224430
           3            1.232329              1.132329                2.292998
           4            1.309130              1.209130                1.209130

                    Continuation_Value_Fishing  Nonpecuniary_Reward_Hammock  \
Identifier Period
0          0                          4.041815                        1.046
           1                          3.230405                        1.046
           2                          2.292871                        1.046
           3                          1.221756                        1.046
           4                          0.000000                        1.046

                    Wage_Hammock  Flow_Utility_Hammock  Value_Function_Hammock  \
Identifier Period
0          0                 NaN              1.046410                4.732282
           1                 NaN              1.061065                3.905840
           2                 NaN              1.057853                3.076295
           3                 NaN              1.038141                2.113919
           4                 NaN              1.058452                1.058452

                    Continuation_Value_Hammock
Identifier Period
0          0                          3.879866
           1                          2.994500
           2                          2.124675
           3                          1.132398
           4                          0.000000

[5 rows x 21 columns]
```

Likelihood function

We can now start exploring the likelihood function that provides an order of preference on θ. The likelihood function is a measure of information about the potentially unknown parameters of the model. The information will usually be incomplete and the likelihood function also expresses the degree of incompleteness

We will usually work with the sum of the individual log-likelihoods throughout as the likelihood cannot be represented without raising problems of numerical overflow. Note that the criterion function of the `respy` package returns to the average log-likelihood across the sample. Thus, we need to be careful with scaling it up when computing some of the test statistics later in the notebook.

We will first trace out the likelihood over reasonable parameter values.

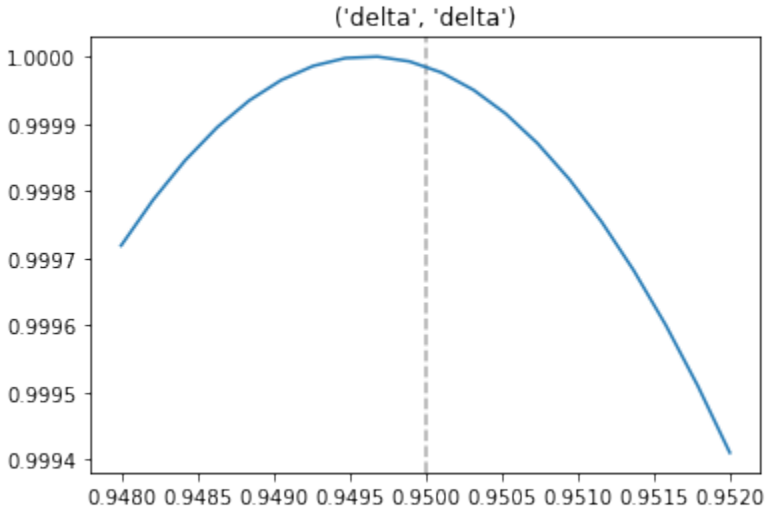
[6]:
```
params_base["lower"] = [0.948, 0.0695, -0.11, 1.04, 0.0030, 0.005, -0.10]
params_base["upper"] = [0.952, 0.0705, -0.09, 1.05, 0.1000, 0.015, +0.10]
```

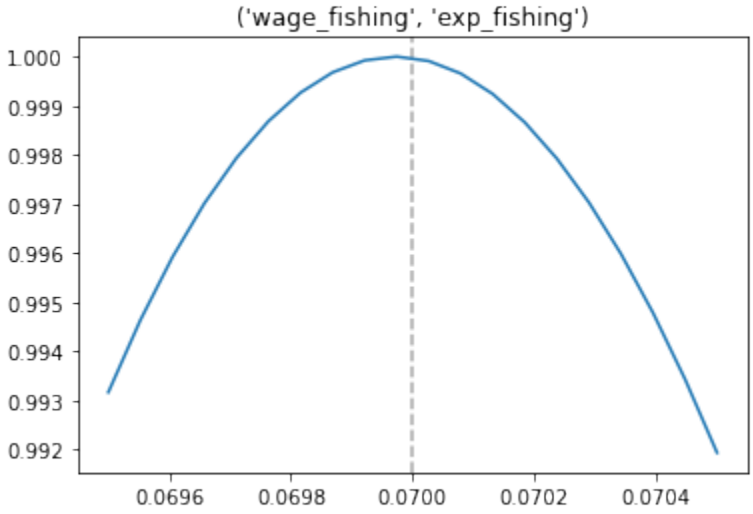
We plot the normalized likelihood, i.e. set the maximum of the likelihood function to one by dividing it by its maximum.

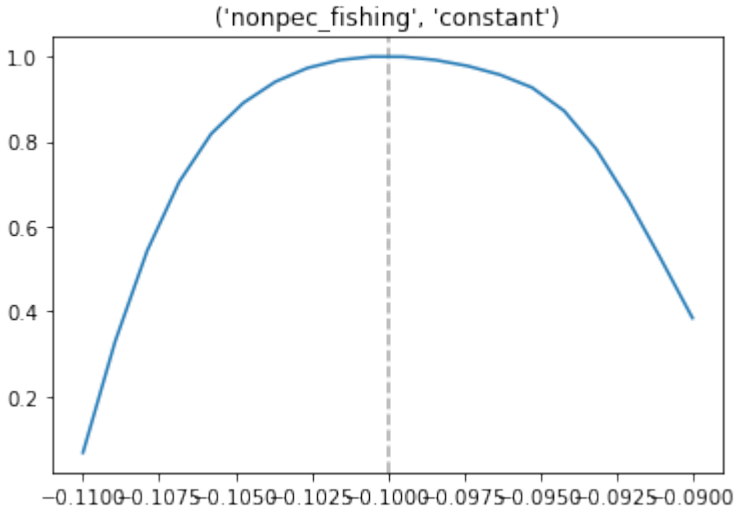
[7]:
```
crit_func = rp.get_log_like_func(params_base, options_base, df)

rslts = dict()
for index in params_base.index:

    upper, lower = params_base.loc[index][["upper", "lower"]]
    grid = np.linspace(lower, upper, 20)

    fvals = list()
    for value in grid:
        params = params_base.copy()
        params.loc[index, "value"] = value
        fval = options_base["simulation_agents"] * crit_func(params)
        fvals.append(fval)

    rslts[index] = fvals
```

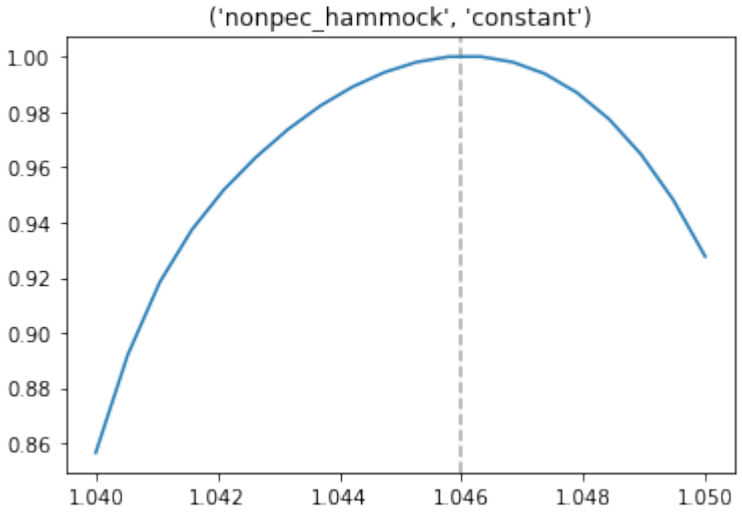
Let's visualize the results.

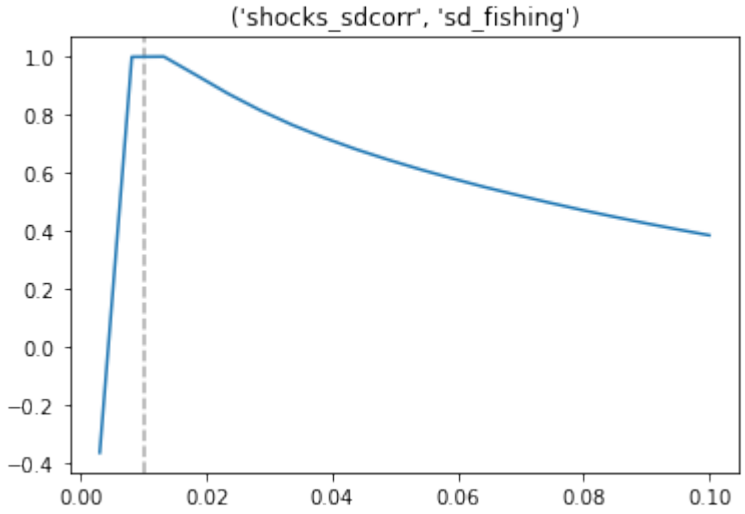
[8]:
```
plot_likelihood(rslts, params_base)
```

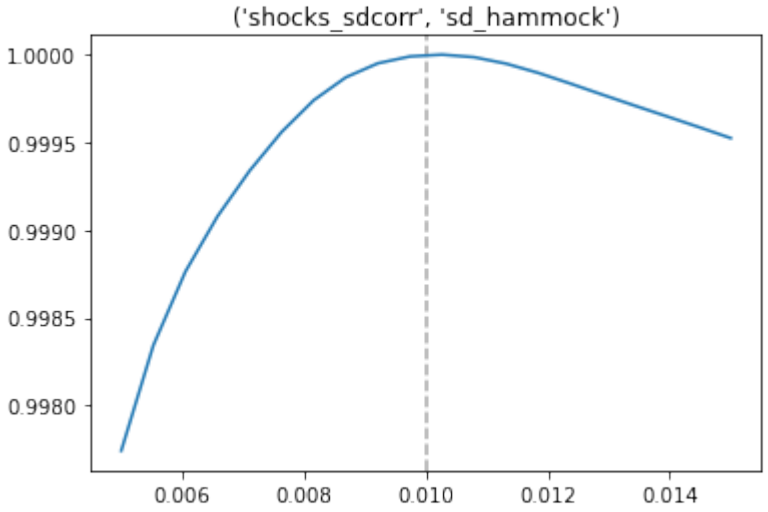


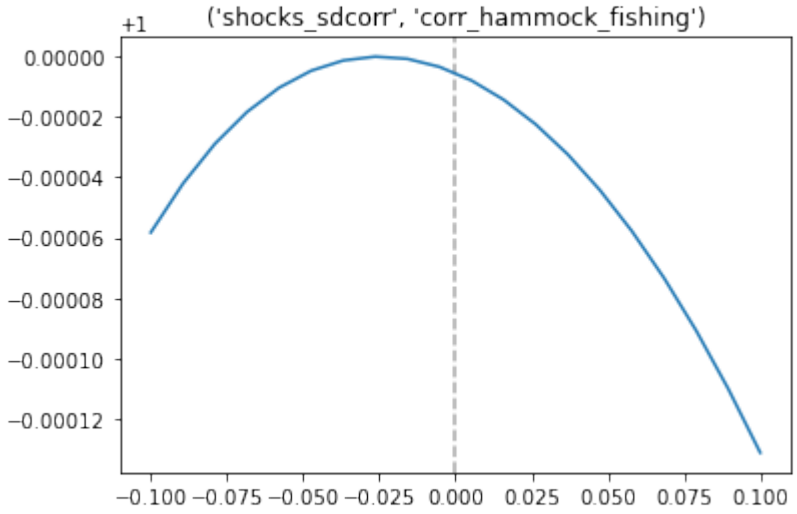

('wage_fishing', 'exp_fishing')
1.000
0.999
0.998
0.997
0.996
0.995
0.994
0.993
0.992
0.0696
0.0698
0.0700
0.0702
0.0704


('nonpec_fishing', 'constant')
1.0
0.8
0.6
0.4
0.2
−0.1100
−0.1075
−0.1050
−0.1025
−0.1000
−0.0975
−0.0950
−0.0925
−0.0900


('nonpec_hammock', 'constant')
1.00
0.98
0.96
0.94
0.92
0.90
0.88
0.86
1.040
1.042
1.044
1.046
1.048
1.050


('shocks_sdcorr', 'sd_fishing')
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
0.00
0.02
0.04
0.06
0.08
0.10


('shocks_sdcorr', 'sd_hammock')
1.0000
0.9995
0.9990
0.9985
0.9980
0.006
0.008
0.010
0.012
0.014


('shocks_sdcorr', 'corr_hammock_fishing')
+1
0.00000
−0.00002
−0.00004
−0.00006
−0.00008
−0.00010
−0.00012
−0.100
−0.075
−0.050
−0.025
0.000
0.025
0.050
0.075
0.100

Maximum likelihood estimate

So far, we looked at the likelihood function in its entirety. Going forward, we will take a narrower view and just focus on the maximum likelihood estimate. We restrict our attention to the discount factor δ and treat it as the only unknown parameter. We will use estimagic for all our estimations.

[9]:
```
crit_func = rp.get_log_like_func(params_base, options_base, df)
```

However, we will make our life even easier and fix all parameters but the discount factor δ.

[10]:
```
constr_base = [
    {"loc": "shocks_sdcorr", "type": "fixed"},
    {"loc": "wage_fishing", "type": "fixed"},
    {"loc": "nonpec_fishing", "type": "fixed"},
    {"loc": "nonpec_hammock", "type": "fixed"},
]
```

We will start the estimation with a perturbation of the true value.

[11]:
```
params_start = params_base.copy()
params_start.loc[("delta", "delta"), "value"] = 0.91
```

Now we are ready to deal with the selection and specification of the optimization algorithm.

[15]:
```
algo_options = {"stopping_max_criterion_evaluations ": 100}
algo_name = "nag_pybobyqa"

results = maximize(
    criterion=crit_func,
    params=params_base,
    algorithm=algo_name,
    algo_options=algo_options,
    constraints=constr_base,
)
```

```
C:\Users\Annica\anaconda3\envs\ekw-lectures\lib\site-packages\estimagic\optimization\
↪optimize.py:851: UserWarning: The following algo_options were ignored because they are␣
↪not compatible with nag_pybobyqa:

 {'stopping_max_criterion_evaluations ': 100}
  warnings.warn(
```

Let's look at the results.

[28]:
```
params_rslt = results["solution_params"]
params_rslt
```

[28]:

		lower	lower_bound	upper	upper_bound
category	name				
delta	delta	0.9480	-inf	0.9520	inf
wage_fishing	exp_fishing	0.0695	-inf	0.0705	inf
nonpec_fishing	constant	-0.1100	-inf	-0.0900	inf
nonpec_hammock	constant	1.0400	-inf	1.0500	inf
shocks_sdcorr	sd_fishing	0.0030	-inf	0.1000	inf
	sd_hammock	0.0050	-inf	0.0150	inf

(continues on next page)

(continued from previous page)

```
                corr_hammock_fishing -0.1000         -inf  0.1000          inf

                                        value
category       name
delta          delta                 0.949629
wage_fishing   exp_fishing           0.070000
nonpec_fishing constant             -0.100000
nonpec_hammock constant              1.046000
shocks_sdcorr  sd_fishing            0.010000
               sd_hammock            0.010000
               corr_hammock_fishing  0.000000
```

[25]:
```
fval = results["solution_criterion"] * options_base["simulation_agents"]
print(f"criterion function at optimum {fval:5.3f}")
```

```
criterion function at optimum -10049.384
```

We need to set up a proper interface to use some other Python functionality going forward.

[29]:
```
def wrapper_crit_func(crit_func, options_base, params_base, value):

    params = params_base.copy()
    params.loc["delta", "value"] = value

    return options_base["simulation_agents"] * crit_func(params)


p_wrapper_crit_func = partial(wrapper_crit_func, crit_func, options_base, params_base)
```

We need to use the MLE repeatedly going forward.

[30]:
```
delta_hat = params_rslt.loc[("delta", "delta"), "value"]
```

At the maximum, the second derivative of the log-likelihood is negative and we define the observed Fisher information as follows

$$I(\hat{\theta}) \equiv -\frac{\partial^2 \log L(\hat{\theta})}{\partial^2 \theta}$$

A larger curvature is associated with a strong peak, thus indicating less uncertainty about θ.

[31]:
```
delta_fisher = -nd.Derivative(p_wrapper_crit_func, n=2)([delta_hat])
```

[32]:
```
delta_fisher
```

[32]:
```
2101705.2240765863
```

Score statistic and Score function

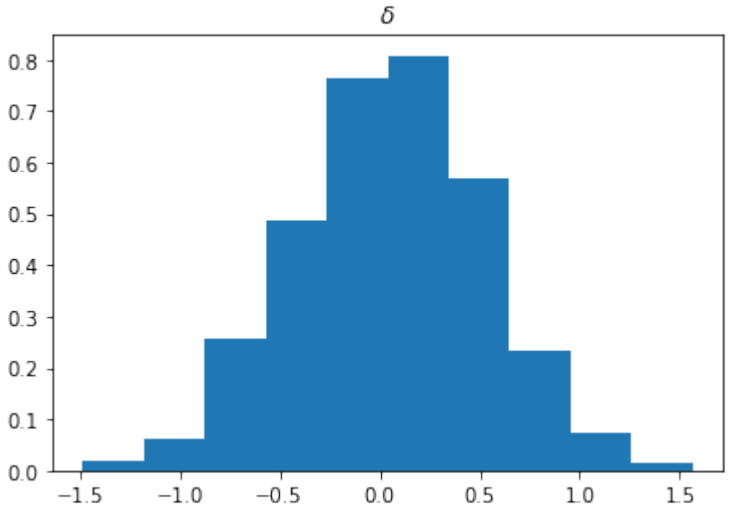
The Score function is the first-derivative of the log-likelihood.

$$S(\theta) \equiv \frac{\partial \log L(\theta)}{\partial \theta}$$

Distribution

The asymptotic normality of the score statistic is of key importance in deriving the asymptotic normality of the maximum likelihood estimator. Here we simulate $1,000$ samples of $10,000$ individuals and compute the score function at the true values. I had to increase the number of simulated individuals as convergence to the asymptotic distribution just took way to long.

[33]:
```
plot_score_distribution()
```



Linearity

We seek linearity of the score function around the true value so that the log-likelihood is reasonably well approximated by a second order Taylor-polynomial.

$$\log L(\theta) \approx \log L(\hat{\theta}) + S(\hat{\theta})(\theta - \hat{\theta}) - \tfrac{1}{2} I(\hat{\theta}))(\theta - \hat{\theta})^2$$

Since $S(\hat{\theta}) = 0$, we get:

$$\log \left(\frac{L(\theta)}{L(\hat{\theta})} \right) \approx -\tfrac{1}{2} I(\hat{\theta})(\theta - \hat{\theta})^2$$

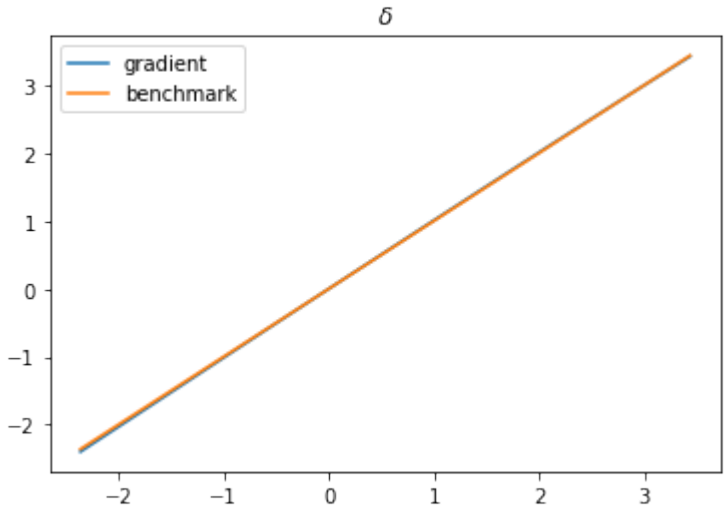
Taking the derivative to work with the score function, the following relationship is approximately true if the usual regularity conditions hold:

$$-I^{-1/2}(\hat{\theta}) S(\theta) \approx I^{1/2}(\hat{\theta})(\theta - \hat{\theta})$$

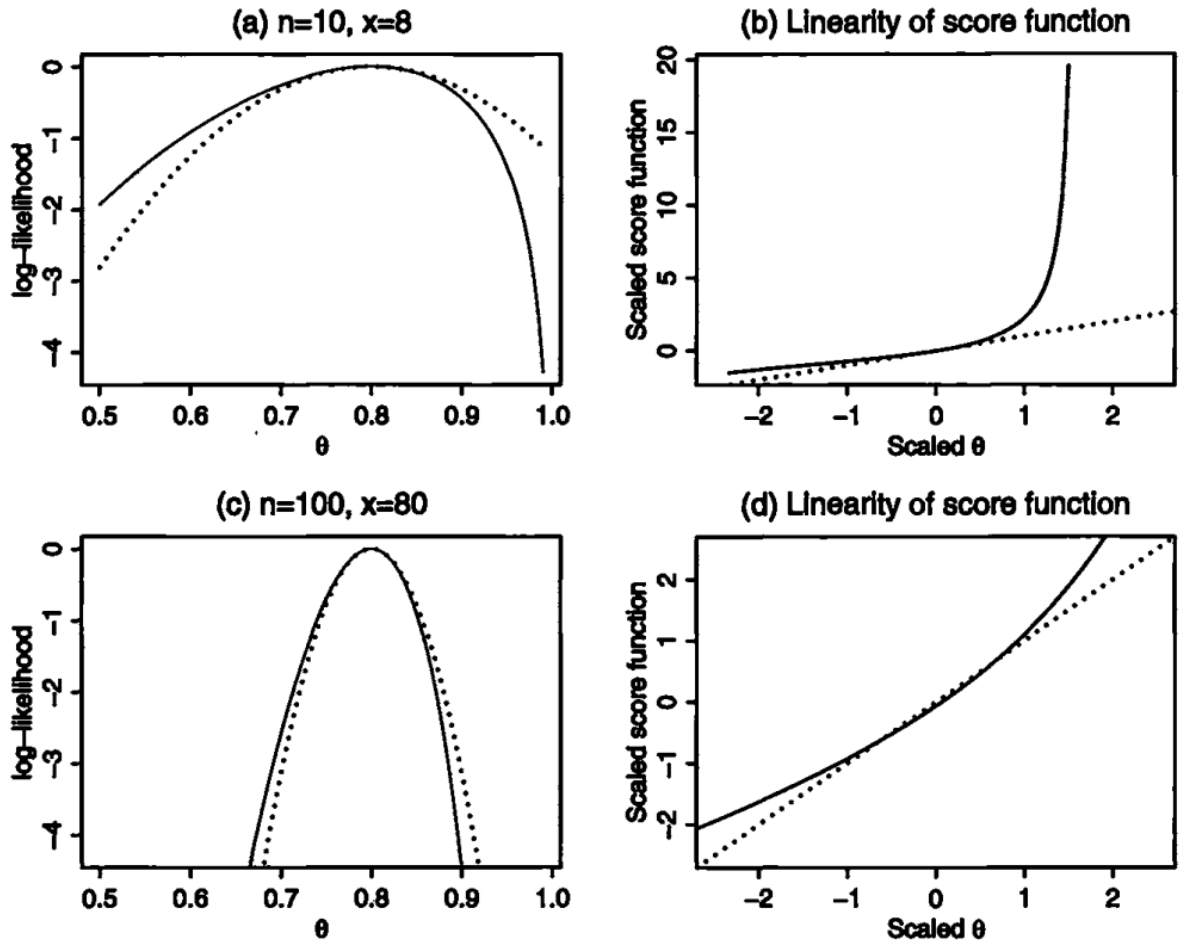
[34]:
```
num_points, index = 10, ("delta", "delta")

upper, lower = params_base.loc[index, ["upper", "lower"]]
grid = np.linspace(lower, upper, num_points)

fds = np.tile(np.nan, num_points)
for i, point in enumerate(grid):
    fds[i] = nd.Derivative(p_wrapper_crit_func, n=1)([point])

norm_fds = fds * -(1 / np.sqrt(delta_fisher))
norm_grid = (grid - delta_hat) * (np.sqrt(delta_fisher))
```

In the best case we see a standard normal distribution of $I^{1/2}(\hat{\theta})(\theta - \hat{\theta})$ and so it is common practice to evaluate the linearity over -2 and 2.

[35]:
```
plot_score_function(norm_grid, norm_fds)
```



Alternative shapes are possible.



Confidence intervals

How do we communicate the statistical evidence using the likelihood? Several notions exist that have different demands on the score function. Wile the Wald intervals rely on the asymptotic normality and linearity, likelihood-based intervals only require asymptotic normality. In well-behaved problems, both measures of uncertainty agree.

Wald intervals

[36]:
```
rslt = list()
rslt.append(delta_hat - 1.96 * 1 / np.sqrt(delta_fisher))
rslt.append(delta_hat + 1.96 * 1 / np.sqrt(delta_fisher))
"{:5.3f} / {:5.3f}".format(*rslt)
```

[36]:
```
'0.948 / 0.951'
```

Likelihood-based intervals

[37]:
```
def root_wrapper(delta, options_base, alpha, index):

    crit_val = -0.5 * chi2.ppf(1 - alpha, 1)

    params_eval = params_base.copy()
    params_eval.loc[("delta", "delta"), "value"] = delta
    likl_ratio = options_base["simulation_agents"] * (
        crit_func(params_eval) - crit_func(params_base)
    )

    return likl_ratio - crit_val
```

[38]:
```
brackets = [[0.75, 0.95], [0.95, 1.10]]

rslt = list()
for bracket in brackets:
    root = root_scalar(
        root_wrapper,
        method="bisect",
        bracket=bracket,
        args=(options_base, 0.05, index),
    ).root
    rslt.append(root)
print("{:5.3f} / {:5.3f}".format(*rslt))
```

```
0.948 / 0.951
```

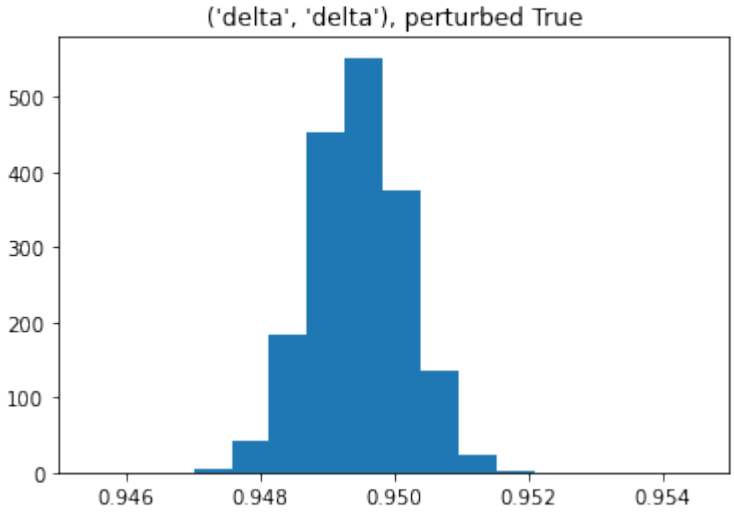
Bootstrap

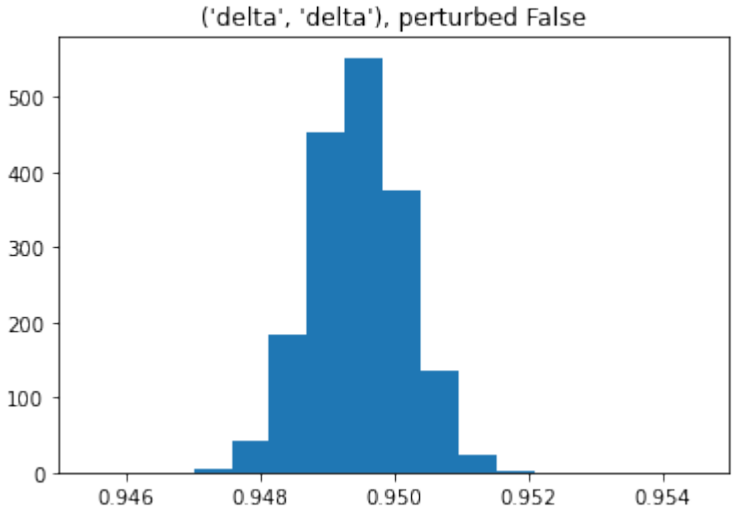
We can now run a simple bootstrap to see how the asymptotic standard errors line up.

Here are some useful resources on the topic:

- Davison, A., & Hinkley, D. (1997). Bootstrap methods and their application. Cambridge University Press, Cambridge.
- Hesterberg, T. C. (2015). What teachers should know about the bootstrap: Resampling in the undergraduate statistics curriculum, *The American Statistician, 69*(4), 371-386.
- Horowitz, J. L. (2001). Chapter 52. The bootstrap. In Heckman, J.J., & Leamer, E.E., editors, *Handbook of Econometrics, 5*, 3159-3228. Elsevier Science B.V.

[39]:
```
plot_bootstrap_distribution()
```





We can now construct the bootstrap confidence interval.

[40]:
```
fname = "material/bootstrap.delta_perturb_true.pkl"
boot_params = pd.read_pickle(fname)

rslt = list()
for quantile in [0.025, 0.975]:
    rslt.append(boot_params.loc[("delta", "delta"), :].quantile(quantile))
print("{:5.3f} / {:5.3f}".format(*rslt))
```

```
0.948 / 0.951
```

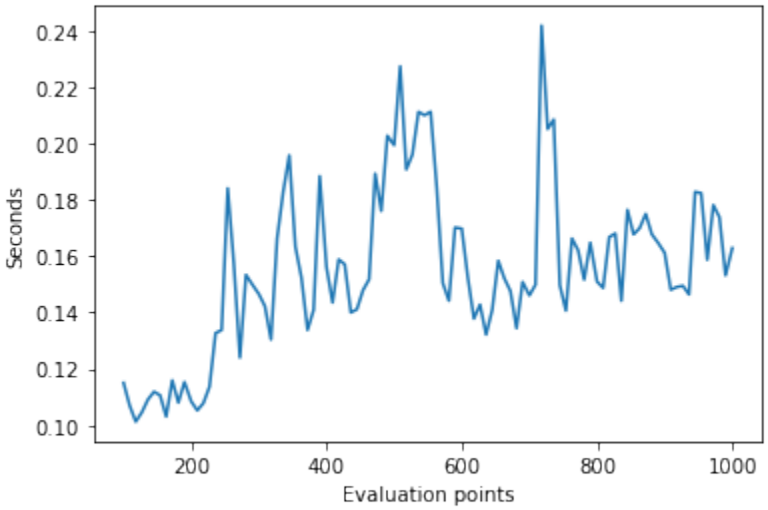
Numerical aspects

The shape and properties of the likelihood function are determined by different numerical tuning parameters such as quality of numerical integration, smoothing of choice probabilities. We would simply choose all components to be the "best", but that comes at the cost of increasing the time to solution.

[41]:
```
grid = np.linspace(100, 1000, 100, dtype=int)

rslts = list()
for num_draws in grid:
    options = options_base.copy()

    options["estimation_draws"] = num_draws
    options["solution_draws"] = num_draws

    start = timer()
    rp.get_solve_func(params_base, options)
    finish = timer()

    rslts.append(finish - start)
```

We are ready to see how time to solution increases as we improve the quality of the numerical integration by increasing the number of Monte Carlo draws.

[42]:
```
plot_computational_budget(grid, rslts)
```



We need to learn where to invest a limited computational budget. We focus on the following going forward:

- smoothing parameter for logit accept-reject simulator
- grid search across core parameters

Smoothing parameter

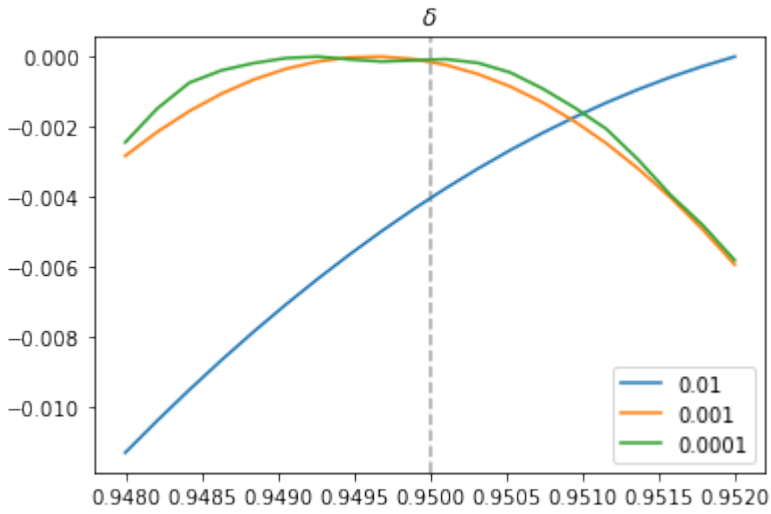
We now show the shape of the likelihood function for alternative choices of the smoothing parameter τ. There exists no closed-form solution for the choice probabilities, so these are simulated. Application of a basic accept-reject (AR) simulator poses the two challenges. First, there is the ocurrance of zero probability simulation for low probability events which causes problems for the evaluation of the log-likelihood. Second, the choice probabilities are not smooth in the parameters and instead are a step function. This is why McFadden (1989) introduces a class of smoothed AR simulators. The logit-smoothed AR simulator is the most popular one and also implemented in `respy`. The implementation requires to specify the smoothing parameter τ. As $\tau \to 0$ the logit smoother approaches the original indicator function.

- McFadden, D. (1989). A method of simulated moments for estimation of discrete response models without numerical integration. *Econometrica, 57*(5), 995-1026.
- Train, K. (2009). Discrete choice methods with simulation. Cambridge University Press, Cambridge.

[43]:
```
rslts = dict()

for tau in [0.01, 0.001, 0.0001]:

    index = ("delta", "delta")

    options = options_base.copy()
    options["estimation_tau"] = tau

    crit_func = rp.get_log_like_func(params_base, options, df)
    grid = np.linspace(0.948, 0.952, 20)

    fvals = list()
    for value in grid:
        params = params_base.copy()
        params.loc[index, "value"] = value
        fvals.append(crit_func(params))

    rslts[tau] = fvals - np.max(fvals)
```

Now we are ready to inspect the shape of the likelihood function.

[44]:
```
plot_smoothing_parameter(rslts, params_base, grid)
```



Grid search

We can look at the interplay of several major numerical tuning parameters. We combine choices for `simulation_agents`, `solution_draws`, `estimation_draws`, and `tau` to see how the maximum of the likelihood function changes.

[45]:
```
df = pd.read_pickle("material/tuning.delta.pkl")
df.loc[((10000), slice(None)), :]
```

[45]:

			delta
agents	draws	tau	
10000	100	1.000000e-02	0.952
		1.000000e-03	0.949684
		1.000000e-04	0.949684
		1.000000e-05	0.949684
		1.000000e-06	0.949684
		1.000000e-07	0.949684
	1000	1.000000e-02	0.952
		1.000000e-03	0.950316
		1.000000e-04	0.950105
		1.000000e-05	0.950105
		1.000000e-06	0.950105
		1.000000e-07	0.950105
	10000	1.000000e-02	0.952
		1.000000e-03	0.950316
		1.000000e-04	0.950105
		1.000000e-05	0.950105
		1.000000e-06	0.950105
		1.000000e-07	0.950105

[ ]:
```
```

We estimate the Robinson Crusoe model using the method of maximum likelihood. We study the likelihood function, the distribution of the score statistic, linearity of the score function, and compare alternative methods to compute confidence intervals. Finally, we study the sensitivity of the likelihood function to numerical parameters.

2.2 Method of simulated moments

[1]:
```
import numpy as np
import pandas as pd
import respy as rp
from python.auxiliary_setup import load_model_specs
from python.auxiliary_plots import plot_chatter_numagents_sim
from python.auxiliary_plots import plot_chatter_numagents_both
from python.auxiliary_plots import plot_criterion_params
from python.auxiliary_plots import plot_moments_choices
from python.auxiliary_plots import plot_moments_wage
from python.auxiliary_plots import plot_chatter
```

2.2.1 Method of Simulated Moments Estimation (MSM)

This lecture heavily relies on respy's method of simulated moments (MSM) interface. A guide to using the interface can be found here.

Table of Contents

Introduction

The method of simulated moments approach to estimating model parameters is to minimize a certain distance between observed moments and simulated moments with respect to the parameters that generate the simulated model.

Denote X our observed data and $m(X)$ the vector of observed moments. To construct the criterion function, we use the parameter vector θ to simulate data from the model $\hat{X}$. We can then calculate the simulated moments $m(\hat{X}|\theta)$.

The criterion function is then given by

$$\psi(\theta) = (m(X) - m(\hat{X}|\theta))'\Omega(m(X) - m(\hat{X}|\theta)) \tag{2.1}$$

where the difference between observed and simulated moments $(m(X)-m(\hat{X}|\theta))$ constitutes a vector of the dimension $1 \times M$ with $1, .., M$ denoting the number of moments. The $M \times M$ weighting matrix is given by Ω.

The MSM estimator is defined as the solution to

$$\hat{\theta}(\Omega) = \underset{\theta}{\operatorname{argmin}} \psi(\theta). \tag{2.2}$$

The criterion function is thus a strictly positive scalar and the estimator depends on the choice of moments m and the weighting matrix Ω. The weighting matrix applies some scaling for discrepancies between the observed and simulated moments. If we use the identity matrix, each moment is given equal weight and the criterion function reduces to the sum of squared moment deviations.

Aside from the choice of moments and weighting matrix, some other important choices that influence the the estimation are the simulator itself and the algorithm and specifications for the optimization procedure. Many explanations for simulated method of moments estimation also feature the number of simulations as a factor that is to be determined for estimation. We can ignore this factor for now since we are working with a large simulated dataset.

In the following we will set up the data, moments and weighting matrix needed to construct the criterion function and subsequently try to estimating the parameters of the model using this MSM setup.

Observed Data

We generate our model and data using respy and a simple Robinson Crusoe model. In this model, the agent, Robinson Crusoe, in each time period decides between two choice options: working (i.e. going fishing) or spending time in the hammock.

We can use respy to simulate the data for this exercise.

[2]:
```
params_true, options = load_model_specs()
```

[3]:
```
# Generate observed data from model.
simulate = rp.get_simulate_func(params_true, options)
data_obs = simulate(params_true)
```

Let's take a look at the model specifications.

[4]:
```
params_true
```

[4]:

		value
category	name	
delta	delta	0.950
wage_fishing	exp_fishing	0.070
nonpec_fishing	constant	-0.100
nonpec_hammock	constant	1.046

(continues on next page)

(continued from previous page)

```
shocks_sdcorr  sd_fishing            0.010
               sd_hammock            0.010
               corr_hammock_fishing  0.000
```

[5]:
```
options
```

[5]:
```
{'estimation_draws': 100,
 'estimation_seed': 100,
 'estimation_tau': 0.001,
 'interpolation_points': -1,
 'n_periods': 5,
 'simulation_agents': 1000,
 'simulation_seed': 132,
 'solution_draws': 100,
 'solution_seed': 456,
 'covariates': {'constant': '1'}}
```

[6]:
```
data_obs.head(10)
```

[6]:
```
                    Experience_Fishing  Shock_Reward_Fishing  \
Identifier Period
0          0                        0              0.999650
           1                        1              1.000743
           2                        2              0.996461
           3                        3              0.998907
           4                        4              0.989419
1          0                        0              0.992894
           1                        0              0.995369
           2                        0              1.000668
           3                        0              0.992542
           4                        0              0.998695

                    Meas_Error_Wage_Fishing  Shock_Reward_Hammock  \
Identifier Period
0          0                             1              0.000410
           1                             1              0.015065
           2                             1              0.011853
           3                             1             -0.007859
           4                             1              0.012452
1          0                             1              0.014190
           1                             1             -0.003848
           2                             1             -0.008441
           3                             1             -0.013488
           4                             1             -0.005397

                    Meas_Error_Wage_Hammock  Dense_Key  Core_Index   Choice  \
Identifier Period
0          0                             1          0           0  fishing
           1                             1          1           1  fishing
           2                             1          2           2  fishing
           3                             1          3           3  fishing
           4                             1          4           4  fishing
```

(continues on next page)

(continued from previous page)

```
1          0                          1          0          0  hammock
           1                          1          1          0  hammock
           2                          1          2          0  hammock
           3                          1          3          0  hammock
           4                          1          4          0  hammock

                       Wage  Discount_Rate  ...  Nonpecuniary_Reward_Fishing  \
Identifier Period                           ...
0          0       0.999650           0.95  ...                         -0.1
           1       1.073305           0.95  ...                         -0.1
           2       1.146203           0.95  ...                         -0.1
           3       1.232329           0.95  ...                         -0.1
           4       1.309130           0.95  ...                         -0.1
1          0            NaN           0.95  ...                         -0.1
           1            NaN           0.95  ...                         -0.1
           2            NaN           0.95  ...                         -0.1
           3            NaN           0.95  ...                         -0.1
           4            NaN           0.95  ...                         -0.1

                     Wage_Fishing  Flow_Utility_Fishing  Value_Function_Fishing  \
Identifier Period
0          0           0.999650              0.899650                4.739374
           1           1.073305              0.973305                4.042189
           2           1.146203              1.046203                3.224430
           3           1.232329              1.132329                2.292998
           4           1.309130              1.209130                1.209130
1          0           0.992894              0.892894                4.732618
           1           0.995369              0.895369                3.740144
           2           1.000668              0.900668                2.837176
           3           0.992542              0.892542                1.885496
           4           0.998695              0.898695                0.898695

                     Continuation_Value_Fishing  Nonpecuniary_Reward_Hammock  \
Identifier Period
0          0                           4.041815                        1.046
           1                           3.230405                        1.046
           2                           2.292871                        1.046
           3                           1.221756                        1.046
           4                           0.000000                        1.046
1          0                           4.041815                        1.046
           1                           2.994500                        1.046
           2                           2.038430                        1.046
           3                           1.045215                        1.046
           4                           0.000000                        1.046

                     Wage_Hammock  Flow_Utility_Hammock  Value_Function_Hammock  \
Identifier Period
0          0                 NaN              1.046410                4.732282
           1                 NaN              1.061065                3.905840
           2                 NaN              1.057853                3.076295
           3                 NaN              1.038141                2.113919
           4                 NaN              1.058452                1.058452
```

(continues on next page)

(continued from previous page)

```
1          0                NaN              1.060190              4.746062
           1                NaN              1.042152              3.876343
           2                NaN              1.037559              2.974068
           3                NaN              1.032512              2.025466
           4                NaN              1.040603              1.040603

                    Continuation_Value_Hammock
Identifier Period
0          0                          3.879866
           1                          2.994500
           2                          2.124675
           3                          1.132398
           4                          0.000000
1          0                          3.879866
           1                          2.983359
           2                          2.038430
           3                          1.045215
           4                          0.000000

[10 rows x 21 columns]
```

Data Moments

For the setup of the estimation we first have to choose a set of moments that we will use to match the observed data and the simulated model. For this model we include two sets of moments:

1. The first set are Robinson's **choice frequencies** (choice frequencies here refers to the share of agents that have chosen a specific option) for each period.
2. The second set are moments that characterize the **wage distribution** for each period, i.e. the mean of the wage of all agents that have chosen fishing in a given period and the standard deviation of the wages.

We need a functions that compute the set of moments on the observed and simulated data.

[7]:
```
def calc_choice_frequencies(df):
    """Calculate choice frequencies"""
    return df.groupby("Period").Choice.value_counts(normalize=True).unstack()
```

[8]:
```
def calc_wage_distribution(df):
    """Calculate wage distribution."""
    return df.groupby(["Period"])["Wage"].describe()[["mean", "std"]]
```

[9]:
```
# Save functions in a dictionary to pass to the criterion function later on.
calc_moments = {
    "Choice Frequencies": calc_choice_frequencies,
    "Wage Distribution": calc_wage_distribution,
}
```

Respy's MSM interface additionally requires us to specify, how missings in the computed moments should be handled. Missing moments for instance can arise, when certain choice options aren't picked at all for some periods. We just specify a function that replaces missings with 0.

[10]:
```
def replace_nans(df):
    """Replace missing values in data."""
    return df.fillna(0)
```

Now we are ready to calculate the moments.

[11]:
```
moments_obs = {
    "Choice Frequencies": replace_nans(calc_moments["Choice Frequencies"](data_obs)),
    "Wage Distribution": replace_nans(calc_moments["Wage Distribution"](data_obs)),
}
```

[12]:
```
print("Choice Frequencies")
print(moments_obs["Choice Frequencies"])
print("\n Wage Distribution")
print(moments_obs["Wage Distribution"])
```

```
Choice Frequencies
        fishing  hammock
Period
0         0.698    0.302
1         0.698    0.302
2         0.698    0.302
3         0.698    0.302
4         0.698    0.302

 Wage Distribution
            mean       std
Period
0       1.003189  0.008640
1       1.072815  0.010737
2       1.150106  0.012316
3       1.234041  0.012406
4       1.322711  0.013473
```

Weighting Matrix

Next we specify a weighting matrix. It needs to be a square matrix with the same number of diagonal elements as there are moments. One option would be to use the identity matrix, but we use a weighting matrix that adjusts for the variance of each moment to improve the estimation process. The variances for the moments are constructed using a bootstrapping procedure.

[13]:
```
def get_weighting_matrix(data, calc_moments, num_boots, num_agents_msm):
    """ Compute weighting matrix for estimation with MSM."""
    # Seed for reproducibility.
    np.random.seed(123)

    index_base = data.index.get_level_values("Identifier").unique()

    # Create bootstrapped moments.
    moments_sample = list()
    for _ in range(num_boots):
        ids_boot = np.random.choice(index_base, num_agents_msm, replace=False)
```

(continues on next page)

(continued from previous page)

```
        moments_boot = [calc_moments[key](data.loc[ids_boot, :]) for key in calc_moments.
↪keys()]

        moments_boot = rp.get_flat_moments(moments_boot)

        moments_sample.append(moments_boot)

    # Compute variance for each moment and construct diagonal weighting matrix.
    moments_var = np.array(moments_sample).var(axis=0)
    weighting_matrix = np.diag(moments_var ** (-1))

    return np.nan_to_num(weighting_matrix)
```

[14]:
```
W = get_weighting_matrix(data_obs, calc_moments, 300, 500)
```

[15]:
```
pd.DataFrame(W)
```

[15]:

	0	1	2	3	4
0	4795.115376	0.000000	0.000000	0.000000	0.000000
1	0.000000	4795.115376	0.000000	0.000000	0.000000
2	0.000000	0.000000	4795.115376	0.000000	0.000000
3	0.000000	0.000000	0.000000	4795.115376	0.000000
4	0.000000	0.000000	0.000000	0.000000	4795.115376
5	0.000000	0.000000	0.000000	0.000000	0.000000
6	0.000000	0.000000	0.000000	0.000000	0.000000
7	0.000000	0.000000	0.000000	0.000000	0.000000
8	0.000000	0.000000	0.000000	0.000000	0.000000
9	0.000000	0.000000	0.000000	0.000000	0.000000
10	0.000000	0.000000	0.000000	0.000000	0.000000
11	0.000000	0.000000	0.000000	0.000000	0.000000
12	0.000000	0.000000	0.000000	0.000000	0.000000
13	0.000000	0.000000	0.000000	0.000000	0.000000
14	0.000000	0.000000	0.000000	0.000000	0.000000
15	0.000000	0.000000	0.000000	0.000000	0.000000
16	0.000000	0.000000	0.000000	0.000000	0.000000
17	0.000000	0.000000	0.000000	0.000000	0.000000
18	0.000000	0.000000	0.000000	0.000000	0.000000
19	0.000000	0.000000	0.000000	0.000000	0.000000

	5	6	7	8	9
0	0.000000	0.000000	0.000000	0.000000	0.000000
1	0.000000	0.000000	0.000000	0.000000	0.000000
2	0.000000	0.000000	0.000000	0.000000	0.000000
3	0.000000	0.000000	0.000000	0.000000	0.000000
4	0.000000	0.000000	0.000000	0.000000	0.000000
5	4795.115376	0.000000	0.000000	0.000000	0.000000
6	0.000000	4795.115376	0.000000	0.000000	0.000000
7	0.000000	0.000000	4795.115376	0.000000	0.000000
8	0.000000	0.000000	0.000000	4795.115376	0.000000
9	0.000000	0.000000	0.000000	0.000000	4795.115376
10	0.000000	0.000000	0.000000	0.000000	0.000000
11	0.000000	0.000000	0.000000	0.000000	0.000000

(continues on next page)

(continued from previous page)

```
12     0.000000     0.000000     0.000000     0.000000     0.000000
13     0.000000     0.000000     0.000000     0.000000     0.000000
14     0.000000     0.000000     0.000000     0.000000     0.000000
15     0.000000     0.000000     0.000000     0.000000     0.000000
16     0.000000     0.000000     0.000000     0.000000     0.000000
17     0.000000     0.000000     0.000000     0.000000     0.000000
18     0.000000     0.000000     0.000000     0.000000     0.000000
19     0.000000     0.000000     0.000000     0.000000     0.000000

                10            11            12            13            14  \
0   0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
1   0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
2   0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
3   0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
4   0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
5   0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
6   0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
7   0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
8   0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
9   0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
10  9.466832e+06  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
11  0.000000e+00  6.036166e+06  0.000000e+00  0.000000e+00  0.000000e+00
12  0.000000e+00  0.000000e+00  4.698312e+06  0.000000e+00  0.000000e+00
13  0.000000e+00  0.000000e+00  0.000000e+00  4.248616e+06  0.000000e+00
14  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  3.566318e+06
15  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
16  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
17  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
18  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
19  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00

                15            16            17            18            19
0   0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
1   0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
2   0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
3   0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
4   0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
5   0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
6   0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
7   0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
8   0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
9   0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
10  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
11  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
12  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
13  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
14  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
15  1.779242e+07  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
16  0.000000e+00  1.258966e+07  0.000000e+00  0.000000e+00  0.000000e+00
17  0.000000e+00  0.000000e+00  9.317930e+06  0.000000e+00  0.000000e+00
18  0.000000e+00  0.000000e+00  0.000000e+00  8.904094e+06  0.000000e+00
19  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  1.067252e+07
```

Criterion Function

We have collected the observed data for our model, chosen the set of moments we want to use for estimation and defined a weighting matrix based on these moments. We can now set up the criterion function to use for estimation.

As already discussed above, the criterion function is given by the weighted square product of the difference between observed moments $m(X)$ and simulated moments $m(\hat{X}|\theta)$. Trivially, if we have that $m(X) = m(\hat{X}|\theta)$, the criterion function returns a value of 0. Thus, the closer θ is to the real parameter vector, the smaller should be the value for the criterion function.

[16]:
```
criterion_msm = rp.get_moment_errors_func(
    params_true, options, calc_moments, replace_nans, moments_obs, W
)
```

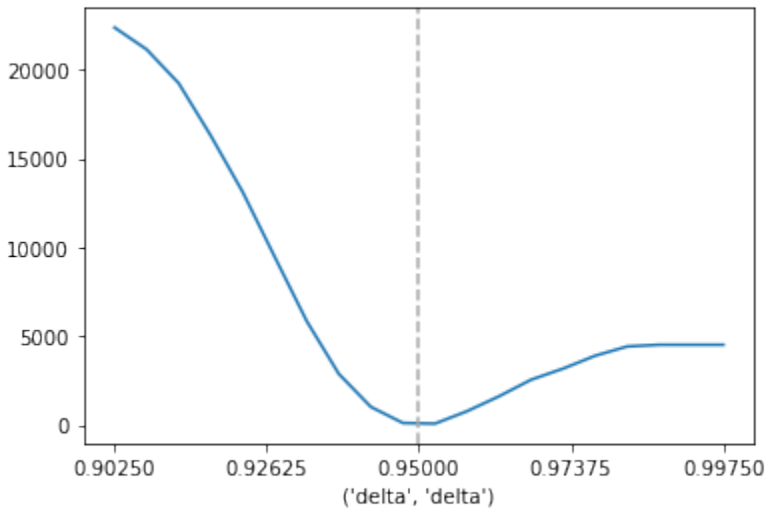
Criterion function at the true parameter vector:

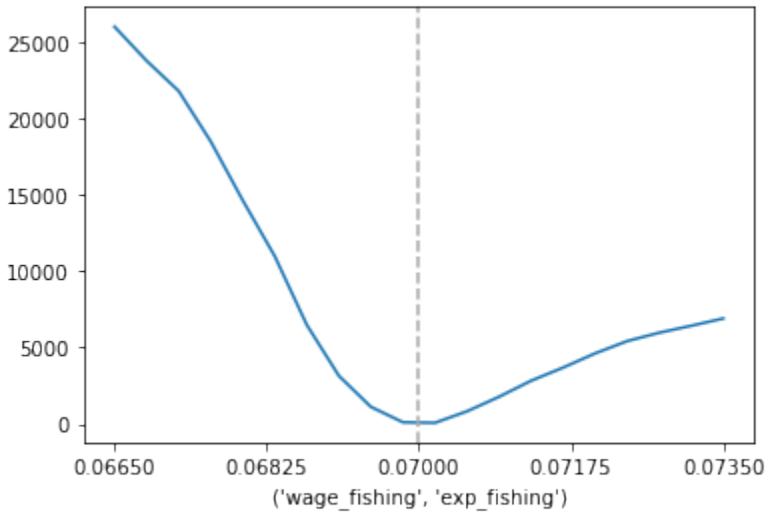
[17]:
```
fval = criterion_msm(params_true)
fval
```

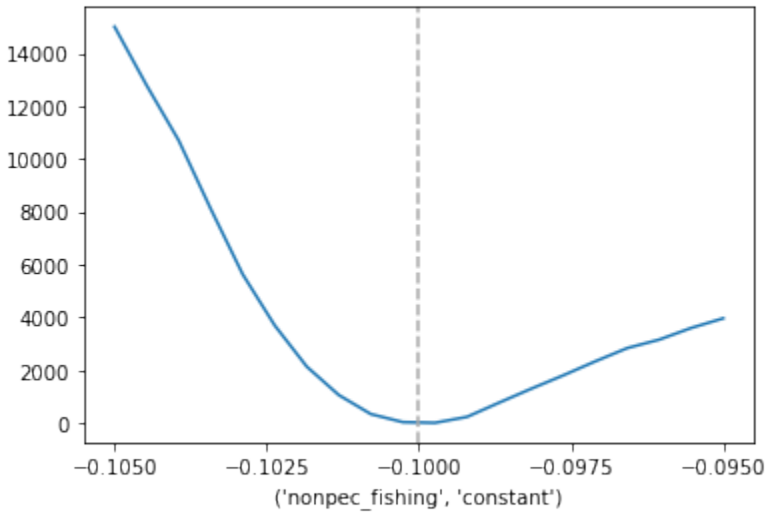
[17]:
```
0.0
```

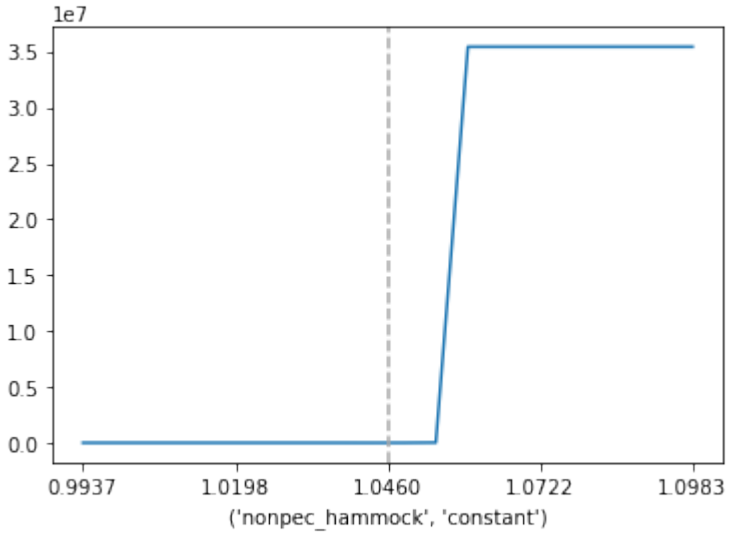
We can plot the criterion function to examine its behavior around the minimum in more detail. The plots below show the criterion function at varying values of all parameters in the the paramter vector.

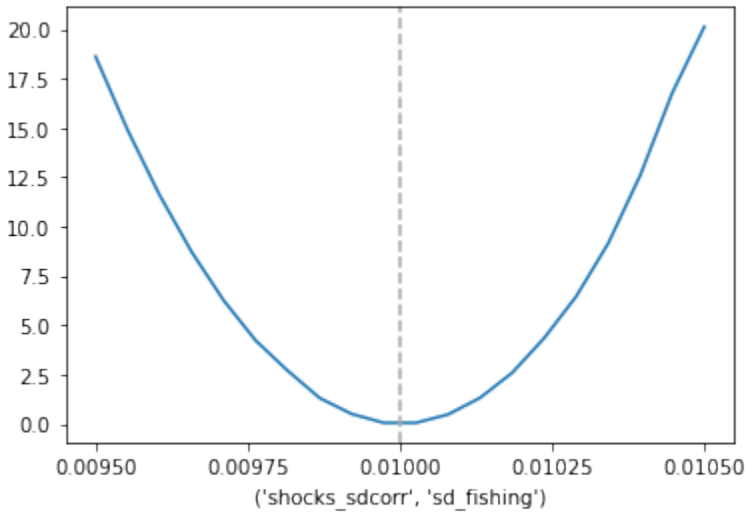
[18]:
```
plot_criterion_params(params_true, list(params_true.index), criterion_msm, 0.05)
```

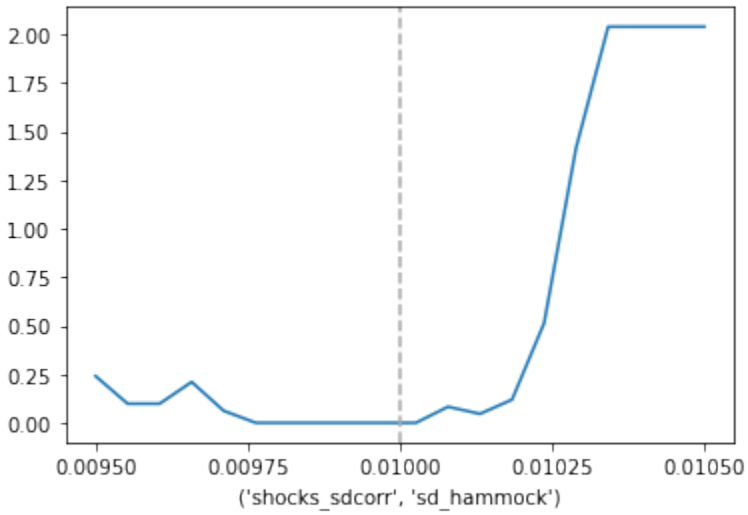


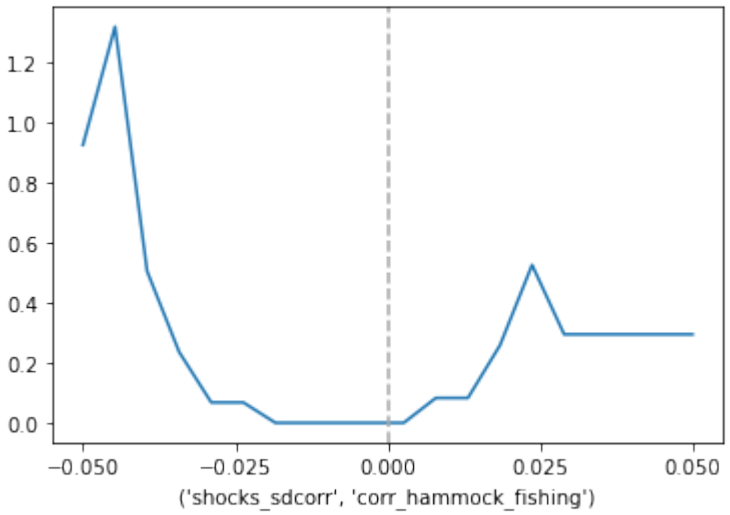

25000
20000
15000
10000
5000
0
0.06650
0.06825
0.07000
0.07175
0.07350
('wage_fishing', 'exp_fishing')


14000
12000
10000
8000
6000
4000
2000
0
−0.1050
−0.1025
−0.1000
−0.0975
−0.0950
('nonpec_fishing', 'constant')


1e7
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0.9937
1.0198
1.0460
1.0722
1.0983
('nonpec_hammock', 'constant')



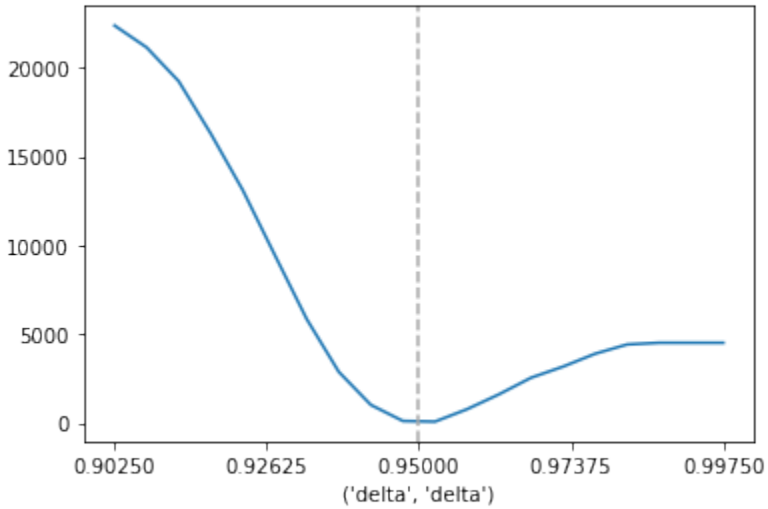


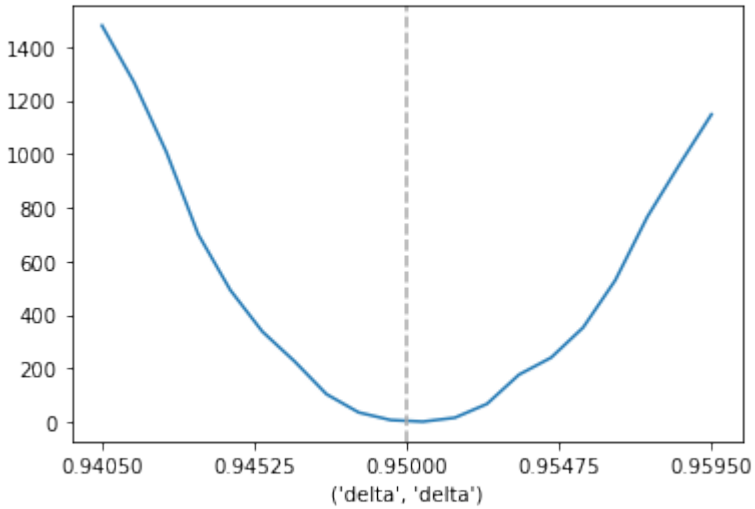


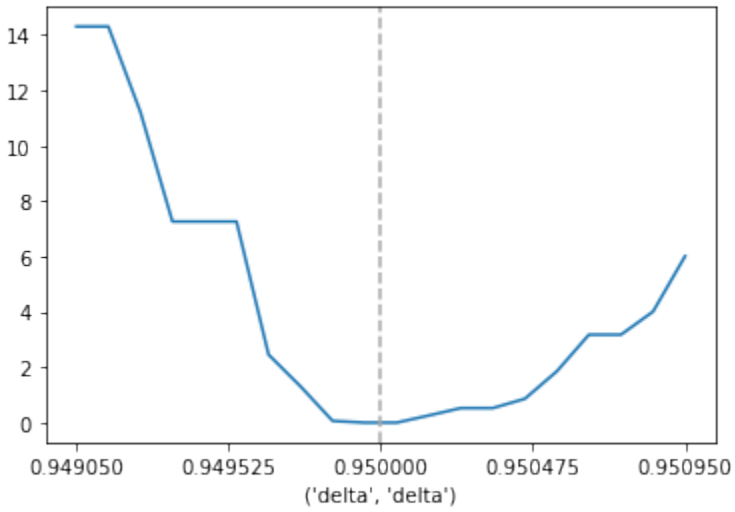
This depiction for most parameters conceals the fact that the criterion function is not a smooth function of our parameter values. We can reveal this property if we 'zoom in' on the function far enough. The plots below show the criterion

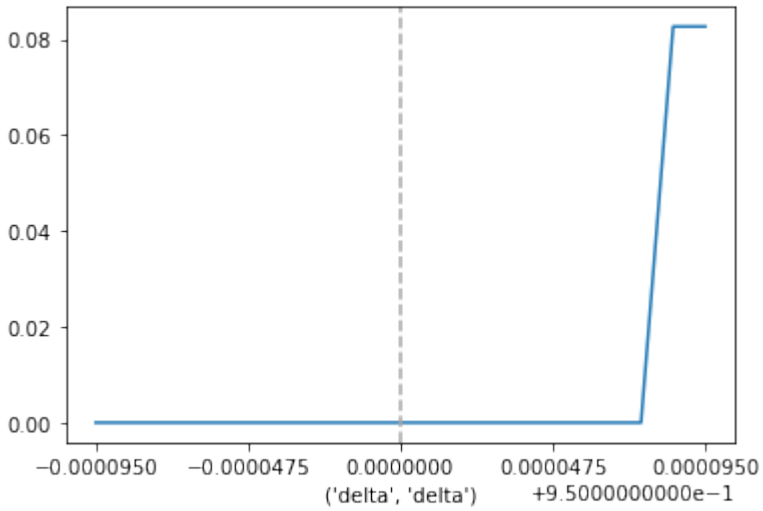
function for varying values of *delta* around the true minimum value of 0.95. We can see that the function exhibits small plateaus and is thus not completely smooth.

[19]:
```
for radius in [0.05, 0.01, 0.001, 0.0001]:
    plot_criterion_params(params_true, list(params_true.index)[0:1], criterion_msm,␣
↪radius)
```









Estimation Exercise

In the following we will conduct a simulation exercise to estimate the parameter vector using our criterion function and weighting matrix. We will begin by simulating data using the new parameter vector and examine how the simulated moments differ from the observed ones. We will then use an optimizer to minimize the criterion function in order to retrieve the true parameter vector. Additionally, we will explore how the criterion function behaves if we change the simulation seed, move away from the true values and test the optimization for a derivative-based optimization algorithm.

Estimation for one parameter

For now, our candidate parameter vector will just differ in *delta* from the true parameters.

[20]:
```
params_cand = params_true.copy()
params_cand.loc["delta", "value"] = 0.93
```

Simulated Moments

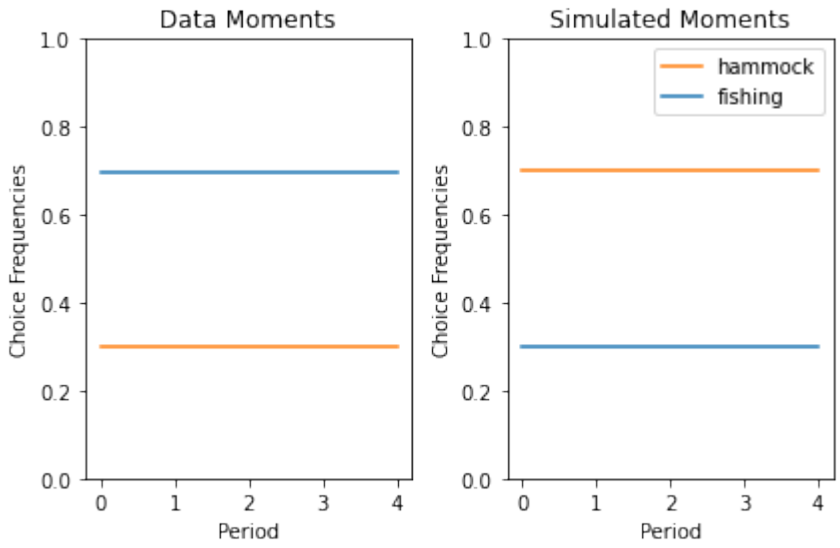
We can now use our model to simulate data using the candidate parameter vector. We can see that the choice frequencies and wage distribution differ from the moments of the observed dataset.

[21]:
```
params = params_cand.copy()
simulate = rp.get_simulate_func(params, options)
df_sim = simulate(params)
```

[22]:
```
moments_sim = {
    "Choice Frequencies": replace_nans(calc_moments["Choice Frequencies"](df_sim)),
    "Wage Distribution": replace_nans(calc_moments["Wage Distribution"](df_sim)),
}
```

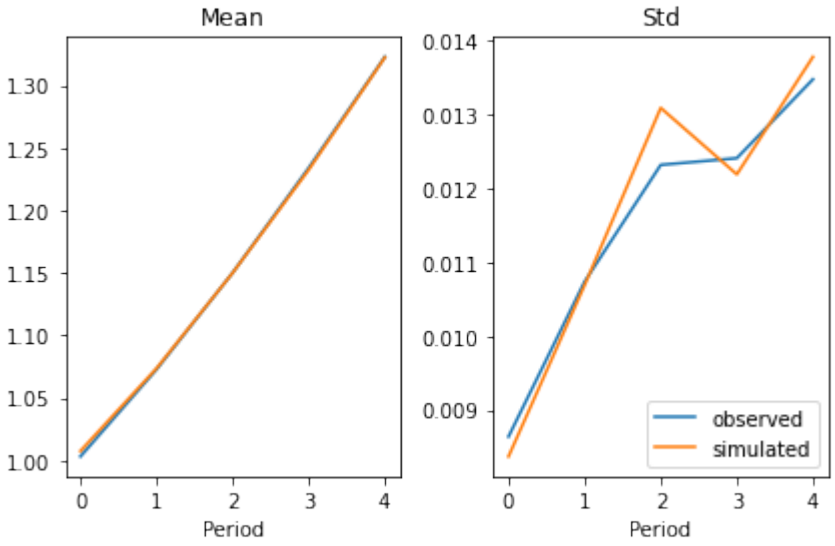
We can plot the moments to compare the choice frequencies for each period.

[23]:
```
plot_moments_choices(moments_obs, moments_sim)
```



The plots below show the mean and the standard deviation in the wages for each period.

[24]:
```
plot_moments_wage(moments_obs, moments_sim)
```



The criterion function value for the candidate parameter vector is not zero.

[25]:
```
fval = criterion_msm(params_cand)
fval
```

[25]:
```
7838.131228444257
```

Optimization Procedure

We will now use an optimization procedure to retrieve the true parameter vector. For the optimization we can use estimagic. In order to minimize the criterion function we need estimagic's `minimize` function.

[26]:
```
from estimagic import minimize  # noqa: E402
```

We have verified above that the criterion function gives a value of 0 for the true parameter vector. Before we try different parameter specifications, we can check whether an optimizer recognizes the true vector as the minimum of our criterion function.

As the code below shows, the optimization algorithm recognizes the true parameter vector as the minimum of the criterion function as it returns a function value of 0 and the true parameter values.

[27]:
```
rslt = minimize(
    criterion=criterion_msm,
    params=params_true,
    algorithm="nag_pybobyqa",
)
rslt["solution_criterion"]
```

[27]:
```
0.0
```

Upper and Lower Bounds

We can help the optimizer by specifying bounds for the parameters. Since we know the true parameters in the case of this model, we can just pick upper and lower bounds that are fairly close to the true values of the parameters to aid the optimizer in the search for the optimum. By default, the upper and lower bounds are set to ∞ and $-\infty$, so specifying upper and lower bounds substantially reduces the range of parameter values that the optimizer can potentially cover.

For optimization with estimagic, we can specify bounds by adding the columns *'lower'* and *'upper'* to the dataframe that contains the parameter values.

[28]:
```
params_cand["lower_bound"] = [0.69, 0.066, -0.2, 1, 0, 0, 0]
params_cand["upper_bound"] = [1, 0.078, -0.09, 1.1, 0.5, 0.5, 0.5]
params_cand
```

[28]:

		value	lower_bound	upper_bound
category	name			
delta	delta	0.930	0.690	1.000
wage_fishing	exp_fishing	0.070	0.066	0.078
nonpec_fishing	constant	-0.100	-0.200	-0.090
nonpec_hammock	constant	1.046	1.000	1.100
shocks_sdcorr	sd_fishing	0.010	0.000	0.500
	sd_hammock	0.010	0.000	0.500
	corr_hammock_fishing	0.000	0.000	0.500

Constraints

Additionally we hold all other parameters fixed for now to aid the optimizer in finding the optimal value for *delta*.

[29]:
```
# Define base constraints to use for the rest of the notebook.
constr_base = [
    {"loc": "shocks_sdcorr", "type": "sdcorr"},
    {"loc": "delta", "type": "fixed"},
    {"loc": "wage_fishing", "type": "fixed"},
    {"loc": "nonpec_fishing", "type": "fixed"},
    {"loc": "nonpec_hammock", "type": "fixed"},
    {"loc": "shocks_sdcorr", "type": "fixed"},
]
```

[30]:
```
# Remove constraint for delta.
constr = constr_base.copy()
constr.remove({"loc": "delta", "type": "fixed"})
```

Optimize

We can now optimize the criterion function with respect to the parameter vector. The optimizer manages to reach a function value of 0 and finds an approximately correct *delta* for our model.

This exercise again reveals that we are dealing with a non-smooth criterion function. The optimizer does not return the exact value of 0.95 for *delta* because of the little plateaus we saw when zooming into the criterion function. As the plot shows, there is a small area around the true value for *delta* that also returns a function value of 0 and might thus be picked by the optimizer.

[31]:
```
rslt = minimize(
    criterion=criterion_msm,
    params=params_cand,
    algorithm="nag_pybobyqa",
    constraints=constr,
)
rslt["solution_criterion"]
```

[31]:
```
0.5248952065747418
```

Chatter in the Criterion Function

In this exercise we explore the sensitivity of the criterion function to the choice of simulation seed and number of agents.

Changing the Simulation Seed

As shown above, the optimizer manages to find a function value of exactly 0 for the true parameter vector. This is the case because respy controls the realization of random elements in the simulation process via a simulation seed. The model thus always simulates the exact same dataset for a given parameter vector. Our criterion function becomes exactly 0 at the true parameter vector because for this vector, the observed and simulated data are identical.

Changing the simulation seed results in simulated moments that, even for the true parameter vector, are never completely equal to the observed moments.

Let's estimate the model with a different simulation seed.

[32]:
```
options_new_seed = options.copy()
options_new_seed["simulation_seed"] = 400
```

We can see that the criterion function is not 0 anymore for the true parameter vector.

[33]:
```
criterion_msm = rp.get_moment_errors_func(
    params_true, options_new_seed, calc_moments, replace_nans, moments_obs, W
)
criterion_msm(params_true)
```

[33]:
```
38.97135021944551
```

Our optimizer thus also does not return a function value of 0 for the true parameter vector.

[34]:
```
rslt_new_seed = minimize(
    criterion=criterion_msm,
    params=params_true,
    algorithm="nag_pybobyqa",
    constraints=constr,
)
rslt_new_seed["solution_criterion"]
```

[34]:
```
17.369549302805428
```

Since the optimizer does not even recognize the true parameter vector, it is also not able to reach a criterion function value of 0 for a different parameter vector.

[35]:
```
rslt_new_seed = minimize(
    criterion=criterion_msm,
    params=params_cand,
    algorithm="nag_pybobyqa",
    constraints=constr,
)

rslt_new_seed["solution_criterion"]
```

[35]:
```
17.369549302805428
```

Multiple Simulation Seeds

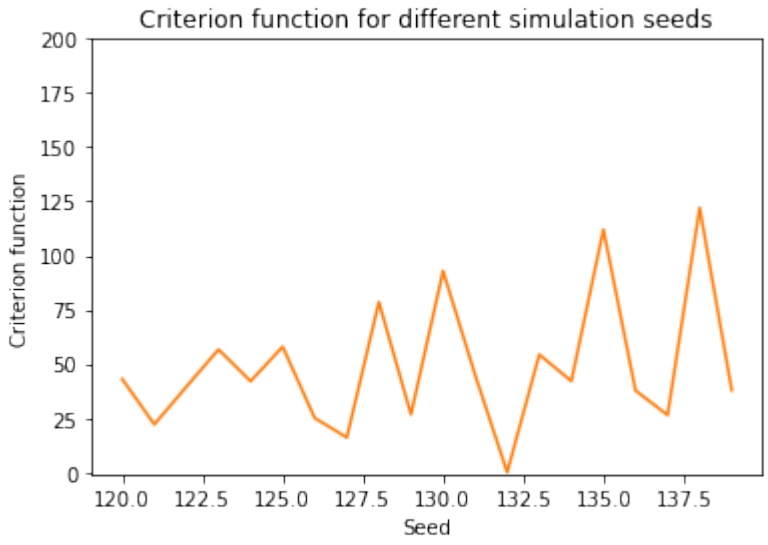
The section above shows what happens to the criterion function if we change the seed for simulating the data. In the sections below we will repeat this exercise for multiple seeds and explore what happens if we increase the number of simulated as well as observed agents in our model.

[36]:
```
# List of seeds, includes true simulation seed of 132.
seeds = list(range(120, 140, 1))
```

[37]:
```
# Define other inputs for criterion function
criterion_kwargs = dict()
criterion_kwargs["params"] = params_true.copy()
criterion_kwargs["options"] = options
criterion_kwargs["calc_moments"] = calc_moments
criterion_kwargs["replace_nans"] = replace_nans
criterion_kwargs["moments_obs"] = moments_obs
criterion_kwargs["weighting_matrix"] = W
```

The plot below shows the criterion function for different simulation seeds, including the true one. We can seed that different seeds lead to different fits between the simulated and observed data, with some seeds performing worse than others. Only the true seed leads to a function value of 0.

[38]:
```
plot_chatter(seeds, criterion_kwargs)
```

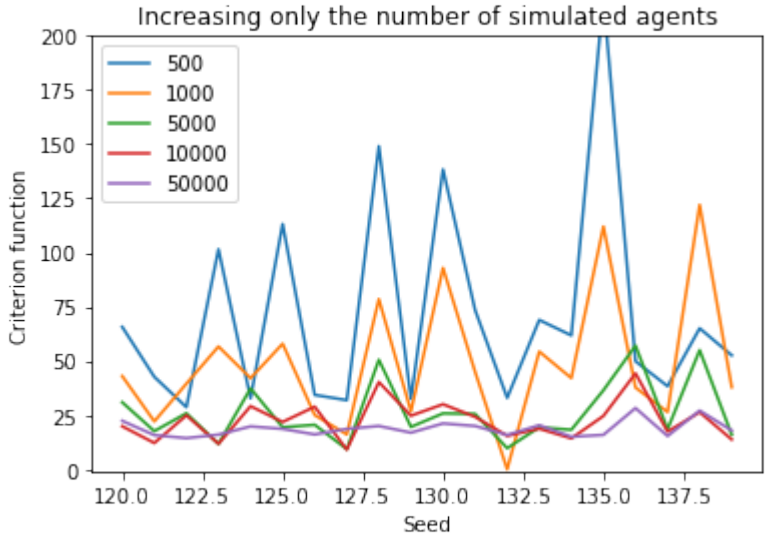


Changing the Simulation Seed for increasing Sample Size of Simulated Agents

We now repeat this exercise for an increasing number of simulated agents. As the plot below shows, increasing the number of simulated agents reduces the chatter considerably but the criterion function remains consistently above zero. Only the simulated sample of 1000 agents at the true simulation seed reaches a function of zero, since, as discussed before, the simulated sample is identical to the observed one for this calibration.

[39]:
```
# Changing the number of agents.
num_agents = [500, 1000, 5000, 10000, 50000]
```

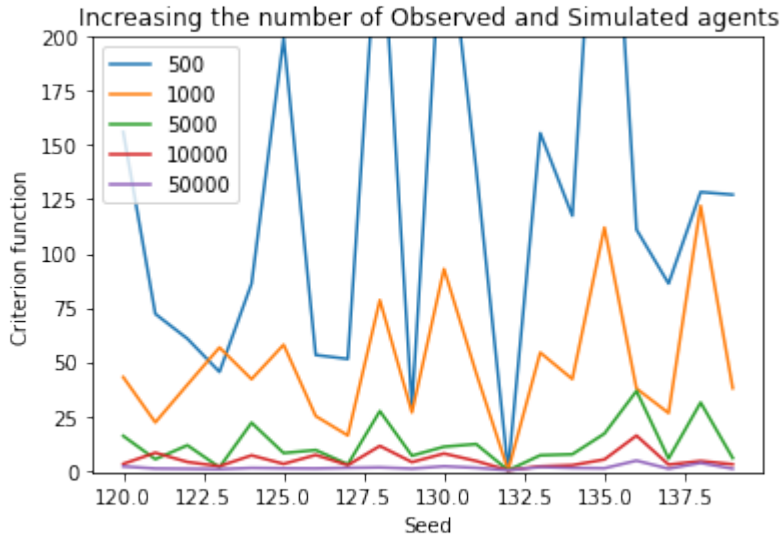
[40]:
```
plot_chatter_numagents_sim(seeds, num_agents, criterion_kwargs)
```



Changing the Simulation Seed for increasing Sample Size of Simulated *and* Observed Agents

We now increase not only the number of simulated agents but also the number of observed agents in our sample. Doing so decreases the chatter in the criterion function as before but also levels out the function around 0 for all simulation seeds. For large enough samples of observed and simulated agents the choice of simulation seed is thus irrelevant.

[41]:
```
plot_chatter_numagents_both(seeds, num_agents, calc_moments, replace_nans, criterion_
↪kwargs)
```



Note: In contrast to the previous plot, we can see that the function reaches a value of 0 at the true simulation seed for all quantities of agents. This is the case because we now simultaneously increase the number of both simulated and observed agents while the observed sample remained untouched in the previous plot. Increasing the number of agents simultaneously for both groups creates identical samples for the true simulation seed at each quantity of agents.

Moving away from the Optimum

For the next exercise we explore what happens to the criterion function if we move away from the optimum. We do this in a controlled fashion by changing the parameter values for *delta* and *wage_fishing* and fixing the latter in the constraints. Doing so should lead *delta* to move away from its optimal value of 0.95 during optimization.

[42]:
```
params_cand.loc["wage_fishing", "value"] = 0.072
params_cand
```

[42]:

		value	lower_bound	upper_bound
category	name			
delta	delta	0.930	0.690	1.000
wage_fishing	exp_fishing	0.072	0.066	0.078
nonpec_fishing	constant	-0.100	-0.200	-0.090
nonpec_hammock	constant	1.046	1.000	1.100
shocks_sdcorr	sd_fishing	0.010	0.000	0.500
	sd_hammock	0.010	0.000	0.500
	corr_hammock_fishing	0.000	0.000	0.500

The optimizer cannot retrieve the true parameter and does not reach a value of 0.

[43]:
```
criterion_msm = rp.get_moment_errors_func(
    params_true, options_new_seed, calc_moments, replace_nans, moments_obs, W
)
rslt_wrong_fix = minimize(
    criterion=criterion_msm,
    params=params_cand,
    algorithm="nag_pybobyqa",
    constraints=constr,
```

(continues on next page)

(continued from previous page)

```
)
rslt_wrong_fix["solution_criterion"]
```

[43]:
```
813.7988993578208
```

Retrieving the true Parameter Vector

We now repeat the estimation with the new parameter vector and free up *wage_fishing* to retrieve the optimal values for both parameters.

The parameter for *wage_fishing* is still 0.072 since we fixed it for the prior estimation:

[44]:
```
params_cand.loc[:, "value"] = rslt_wrong_fix["solution_params"][["value"]]
params_cand
```

[44]:

		value	lower_bound	upper_bound
category	name			
delta	delta	0.923022	0.690	1.000
wage_fishing	exp_fishing	0.072000	0.066	0.078
nonpec_fishing	constant	-0.100000	-0.200	-0.090
nonpec_hammock	constant	1.046000	1.000	1.100
shocks_sdcorr	sd_fishing	0.010000	0.000	0.500
	sd_hammock	0.010000	0.000	0.500
	corr_hammock_fishing	0.000000	0.000	0.500

We now free up *wage_fishing* in the constraints in addition to *delta*.

[45]:
```
# Adjust constraints to free up both delta and wage_fishing.
constr_u = constr_base.copy()
constr_u.remove({"loc": "delta", "type": "fixed"})
constr_u.remove({"loc": "wage_fishing", "type": "fixed"})
```

Freeing up the non-optimal *wage_fishing* improves the estimates. The criterion function value is much closer to 0 and the optimizer manages to retrieve the true parameter values quite closely.

[62]:
```
rslt_unfix = minimize(
    criterion=criterion_msm,
    params=params_cand[["value"]],
    algorithm="nag_pybobyqa",
    constraints=constr_u,
)
rslt_unfix["solution_criterion"]
```

[62]:
```
710.9165575145821
```

For easier comparison, we can compute the difference between the true and estimated value:

[63]:
```
deviation = params_true["value"] - rslt_unfix["solution_params"]["value"]
deviation
```

[63]:

category	name	
delta	delta	0.026703
wage_fishing	exp_fishing	-0.001791
nonpec_fishing	constant	0.000000

(continues on next page)

(continued from previous page)

```
nonpec_hammock  constant                0.000000
shocks_sdcorr   sd_fishing              0.000000
                sd_hammock              0.000000
                corr_hammock_fishing    0.000000
Name: value, dtype: float64
```

Derivative-Based Optimization Algorithm

So far we have used only one optimization algorithm to estimate the parameter vector. The algorithm we used, BOBYQA (Bound Optimization by Quadratic Approximation), is a derivative-free optimization algorithm which works fairly well on our criterion function. As discussed above, our criterion function is a step function that contains plateaus for certain ranges of parameter values. An important implication of this property is that we cannot calculate proper derivatives and thus derivative-based optimization algorithms will not work to estimate the parameters.

To demonstrate this problem, we will now try to estimate the parameters using an optimization algorithm that does use derivatives during optimization.

[64]:
```
# Define candidate parameter vector and constraints.
params_cand = params_true.copy()
params_cand.loc["delta", "value"] = 0.93

params_cand["lower_bound"] = [0.79, 0.066, -0.11, 1.04, 0, 0, 0]
params_cand["upper_bound"] = [1, 0.072, -0.095, 1.055, 0.5, 0.5, 0.5]

constr = constr_base.copy()
constr.remove({"loc": "delta", "type": "fixed"})
```

We try the L-BFGS-B (Limited-Memory-Broyden-Fletcher-Goldfarb-Shanno with box constraints) algorithm. As shown below, L-BFGS-B fails and the optimizer returns the same parameter vector that we used as an input.

[65]:
```
criterion_msm = rp.get_moment_errors_func(
    params_true, options_new_seed, calc_moments, replace_nans, moments_obs, W
)
rslt = minimize(
    criterion=criterion_msm,
    params=params_cand,
    algorithm="scipy_lbfgsb",
    constraints=constr,
)
rslt
```

[65]:
```
{'solution_x': array([0.93]),
 'solution_criterion': 7204.902739425344,
 'solution_derivative': array([0.]),
 'solution_hessian': None,
 'n_criterion_evaluations': 1,
 'n_derivative_evaluations': None,
 'n_iterations': 0,
 'success': True,
 'reached_convergence_criterion': None,
 'message': 'CONVERGENCE: NORM_OF_PROJECTED_GRADIENT_<=_PGTOL',
 'solution_params':                                      lower_bound  upper_bound  value
```

(continues on next page)

(continued from previous page)

```
category       name
delta          delta                  0.790        1.000  0.930
wage_fishing   exp_fishing            0.066        0.072  0.070
nonpec_fishing constant              -0.110       -0.095 -0.100
nonpec_hammock constant               1.040        1.055  1.046
shocks_sdcorr  sd_fishing             0.000        0.500  0.010
               sd_hammock             0.000        0.500  0.010
               corr_hammock_fishing   0.000        0.500  0.000}
```

References

- Adda, J., & Cooper, R. W. (2003). *Dynamic Economics: Quantitative Methods and Applications*. MIT press.
- Adda, J., Dustmann, C., & Stevens, K. (2017). The Career Costs of Children. *Journal of Political Economy*, 125(2), 293-337.
- Andrews, I., Gentzkow, M., & Shapiro, J. M. (2017). Measuring the Sensitivity of Parameter Estimates to Estimation Moments. *The Quarterly Journal of Economics*, 132(4), 1553-1592.
- Bruins, M., Duffy, J. A., Keane, M. P., & Smith Jr, A. A. (2018). Generalized Indirect Inference for Discrete Choice Models. *Journal of econometrics*, 205(1), 177-203.
- Davidson, R., & MacKinnon, J. G. (2004). *Econometric Theory and Methods (Vol. 5)*. New York: Oxford University Press.
- Evans, R. W. (2018, July 5). Simulated Method of Moments (SMM) Estimation. Retrieved November 30, 2019, from https://notes.quantecon.org/submission/5b3db2ceb9eab00015b89f93.
- Frazier, D. T., Oka, T., & Zhu, D. (2019). Indirect Inference with a Non-Smooth Criterion Function. *Journal of Econometrics*, 212(2), 623-645.
- Gourieroux, M., & Monfort, D. A. (1996). *Simulation-based econometric methods*. Oxford university press.
- McFadden, D. (1989). A Method of Simulated Moments for Estimation of Discrete Response Models without Numerical Integration. *Econometrica: Journal of the Econometric Society*, 995-1026.

We estimate the Robinson Crusoe model using the method of simulated moments. We revisit the basic ideas behind this approach and then prototype a criterion function and a weighting matrix. We conclude with a Monte Carlo exploration of selected challenges in the estimation procedure.

CHAPTER

THREE

EXTENSIONS

We showcase several extensions to the baseline model and its standard analysis.

Hyperbolic discounting

A central component of dynamic behavioral models are time preferences that characterize the intertemporal decision problem agents face. However, time preference parameters are usually underidentified in structural models which can have far-reaching effects on the counterfactual prediction capacities of a model.

In this lecture, we examine the identification of time preference parameters in a discrete choice dynamic model of occupational choice. We extend the traditional exponential discounting framework by introducing (quasi-)hyperbolic discounting and naïve agents.

Uncertainty quantification

… work in progress

CHAPTER

FOUR

MISCELLANEOUS

[1]:
```
import yaml
import sys
import os

import pandas as pd
import respy as rp

sys.path.insert(0, "python")
from auxiliary import plot_average_wages_over_time  # noqa: E402
from auxiliary import plot_choice_probabilities  # noqa: E402
from auxiliary import plot_distibution_wages  # noqa: E402
```

4.1 Robinson Crusoe economy

The `respy` package provides several example models that can be used to explore selected issues in the structural microeconometrics. We focus on the `robinson` example, which allows for quick iterations. All example models share the same interface so they can be easily swapped in and out of any analysis.

[2]:
```
options_base = yaml.safe_load(open(os.environ["ROBINSON_SPEC"] + "/robinson.yaml", "r"))

params_base = pd.read_csv(open(os.environ["ROBINSON_SPEC"] + "/robinson.csv", "r"))
params_base.set_index(["category", "name"], inplace=True)
```

Let's look at the parameterization of the model, the specified options and a simulated dataset.

[3]:
```
params_base
```

[3]:

		value
category	name	
delta	delta	0.950
wage_fishing	exp_fishing	0.070
nonpec_fishing	constant	-0.100
nonpec_hammock	constant	1.046
shocks_sdcorr	sd_fishing	0.010
	sd_hammock	0.010
	corr_hammock_fishing	0.000

[4]:
```
options_base
```

[4]:
```
{'estimation_draws': 100,
 'estimation_seed': 100,
 'estimation_tau': 0.001,
 'interpolation_points': -1,
 'n_periods': 5,
 'simulation_agents': 1000,
 'simulation_seed': 132,
 'solution_draws': 100,
 'solution_seed': 456,
 'covariates': {'constant': '1'}}
```

Now we are ready to simulate a sample.

[5]:
```
simulate = rp.get_simulate_func(params_base, options_base)
df = simulate(params_base)
```

We first look at the general structure of the data.

[6]:
```
df.head()
```

[6]:

		Experience_Fishing	Shock_Reward_Fishing	Meas_Error_Wage_Fishing	Shock_Reward_Hammock	Meas_Error_Wage_Hammock	Choice	Wage	Discount_Rate	Present_Bias	Nonpecuniary_Reward_Fishing	Wage_Fishing	Flow_Utility_Fishing	Value_Function_Fishing
Identifier	Period													
0	0	0	1.007197	1	0.010305	1	fishing	1.007197	0.95	1	-0.1	1.007197		
	1	1	0.981015	1	0.010596	1	fishing	1.052147	0.95	1	-0.1	1.052147		
	2	2	0.998980	1	-0.003797	1	fishing	1.149101	0.95	1	-0.1	1.149101		
	3	3	0.989253	1	-0.011702	1	fishing	1.220419	0.95	1	-0.1	1.220419		
	4	4	1.010431	1	-0.002176	1	fishing	1.336932	0.95	1	-0.1	1.336932		

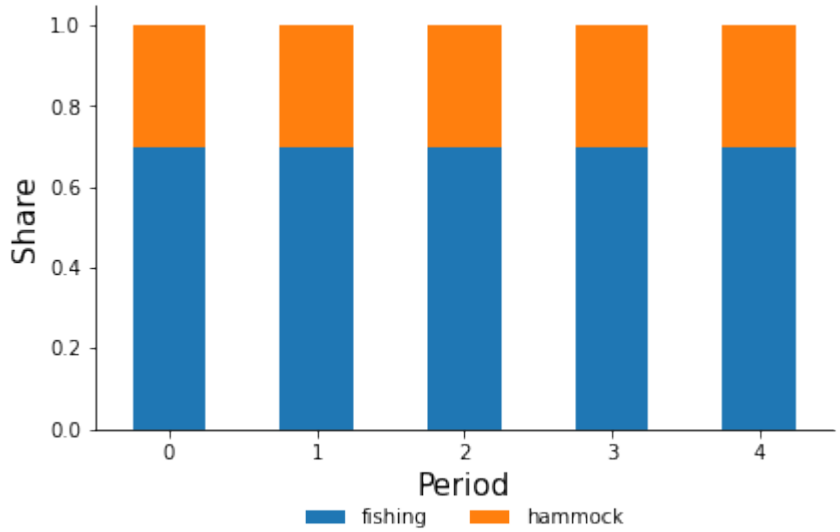
(continues on next page)

(continued from previous page)

```
0           0                    0.907197                4.747606
            1                    0.952147                4.022193
            2                    1.049101                3.230306
            3                    1.120419                2.282845
            4                    1.236932                1.236932

                    Continuation_Value_Fishing  Nonpecuniary_Reward_Hammock  \
Identifier Period
0           0                         4.042536                        1.046
            1                         3.231627                        1.046
            2                         2.296006                        1.046
            3                         1.223607                        1.046
            4                         0.000000                        1.046

                    Wage_Hammock  Flow_Utility_Hammock  Value_Function_Hammock  \
Identifier Period
0           0                NaN              1.056305                4.742931
            1                NaN              1.056596                3.903085
            2                NaN              1.042203                3.063422
            3                NaN              1.034298                2.111715
            4                NaN              1.043824                1.043824

                    Continuation_Value_Hammock
Identifier Period
0           0                         3.880659
            1                         2.996304
            2                         2.127598
            3                         1.134123
            4                         0.000000
```

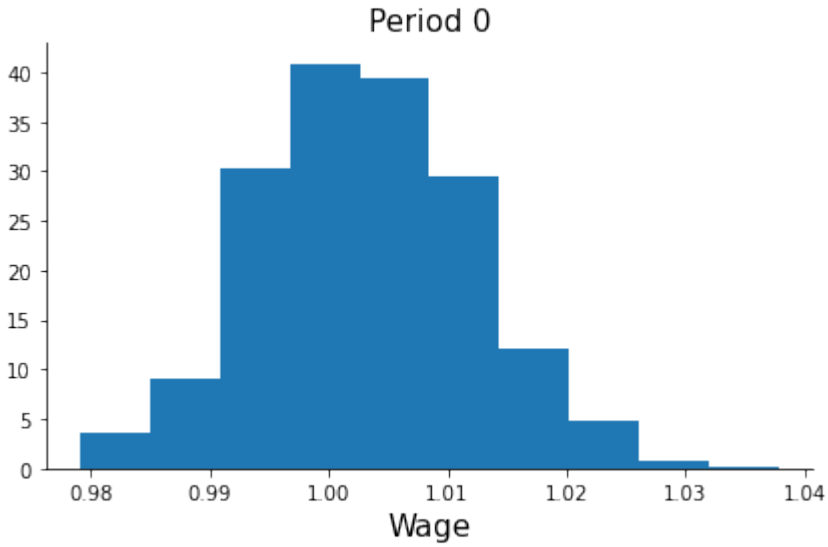
We can study the resulting choice patterns.

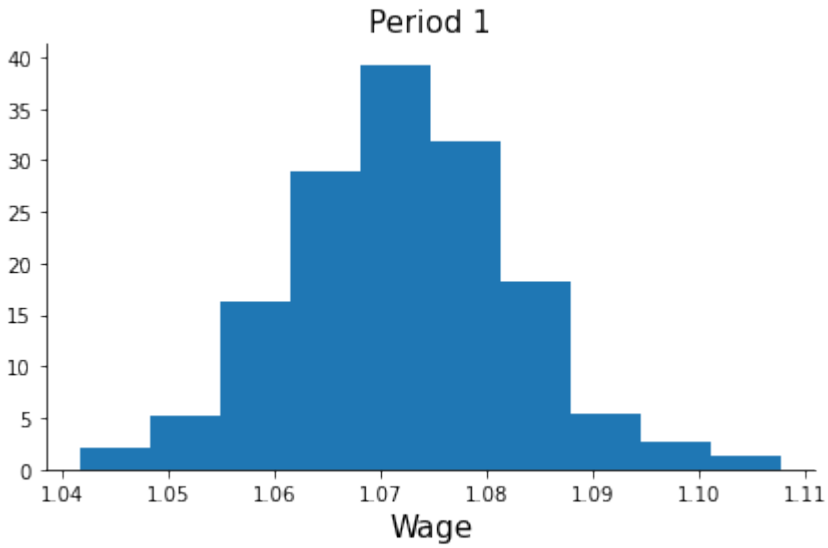
[7]:
```
plot_choice_probabilities(df)
```

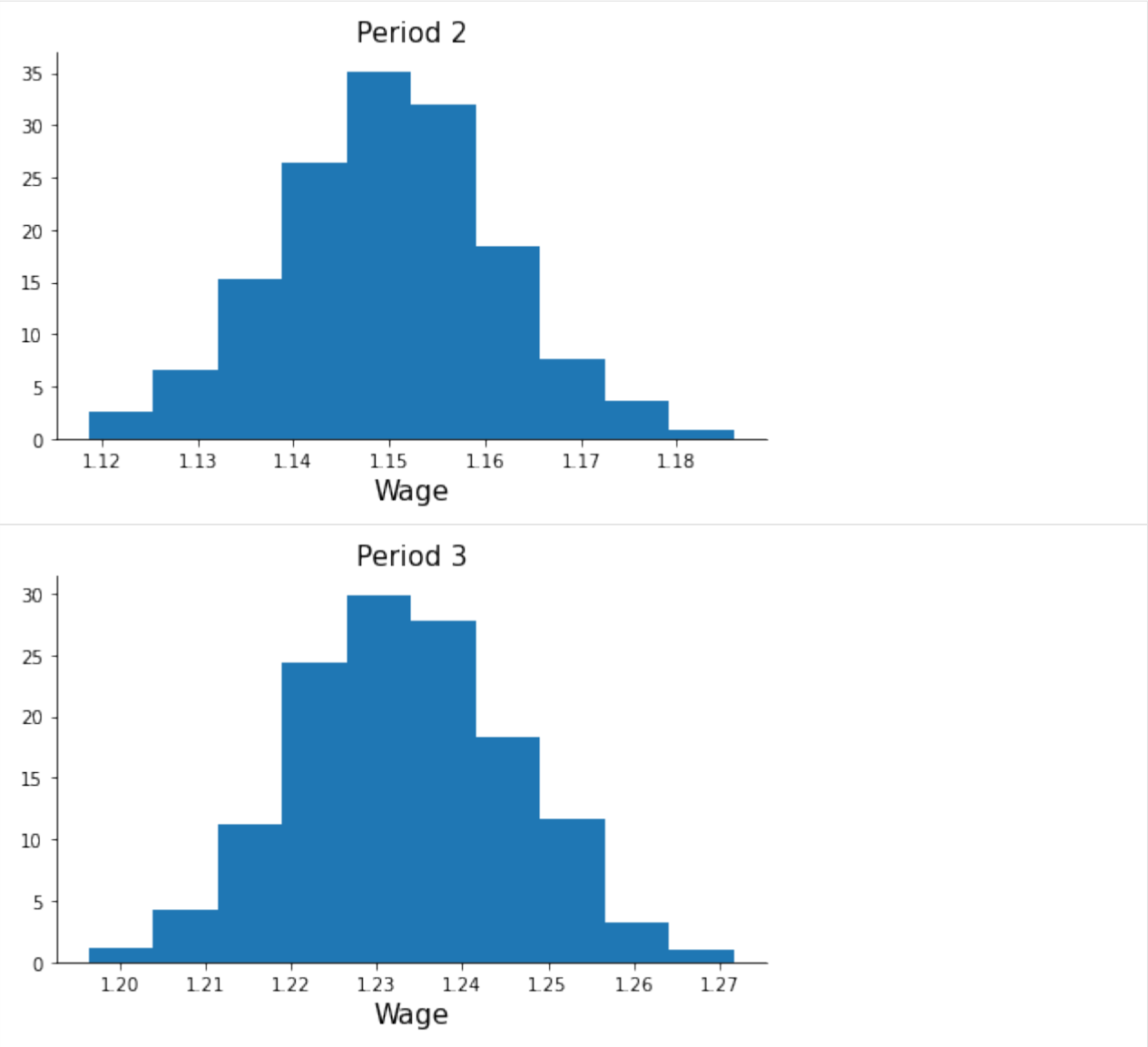


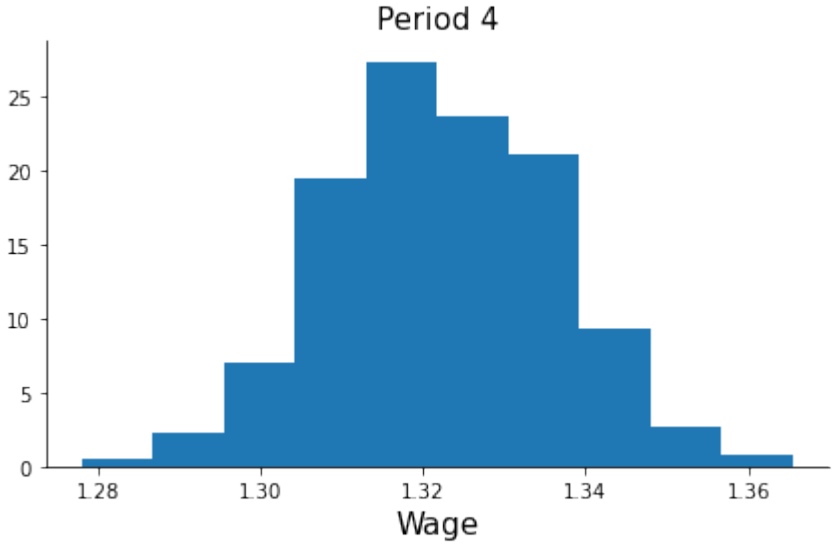
We can look at the distirbution of wages.

[8]:
```
plot_distibution_wages(df)
```



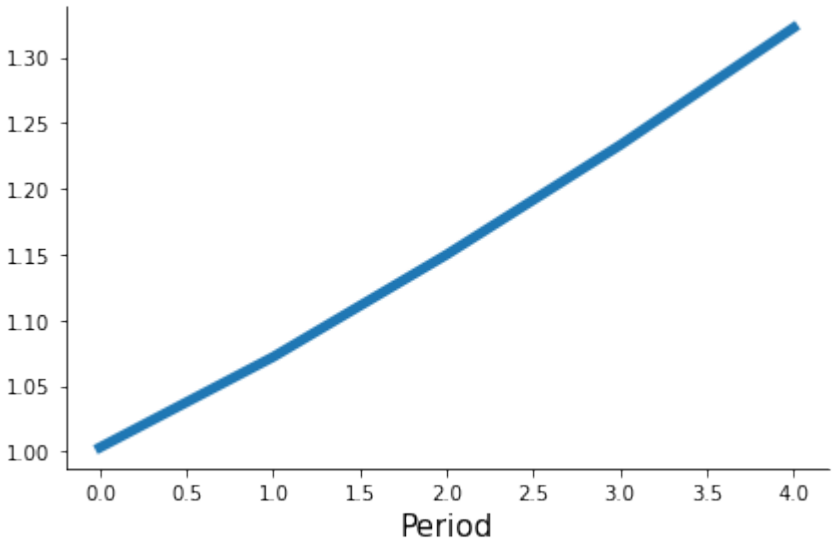



Period 2
35
30
25
20
15
10
5
0
1.12
1.13
1.14
1.15
1.16
1.17
1.18
Wage
Period 3
30
25
20
15
10
5
0
1.20
1.21
1.22
1.23
1.24
1.25
1.26
1.27
Wage



We can look at average wages over time.

[9]:
```
plot_average_wages_over_time(df)
```



We outline the model that serves as the running example throughout the lectures.

CHAPTER

FIVE

REPLICATIONS

Please see `respy`'s online documentation for several replications of seminal papers in the literature.

CHAPTER

SIX

REVIEWS

- **Blundell, R. (2017)**. What have we learned from structural models?, *American Economic Review*, 107(5): 287-92.
- **Galiani, S., Pantano, J. (2021)**. Structural models: Inception and frontier, *(National Bureau of Economic Research Working Papers No. 28698)*.
- **Keane, M., Todd, P., and Wolpin, K. I. (2011)**. The structural estimation of behavioral models: Discrete choice dynamic programming methods and applications, In *Ashenfelter, O. and Card, D. eds.* (pp. 331-461), Elsevier.
- **Low, H., Meghir, C. (2017)**. The use of structural models in econometrics, *Journal of Economic Perspectives*, 31(2): 33-58.

CHAPTER

SEVEN

POWERED BY