Artificial intelligence, algorithmic pricing and collusion
Code description

Emilio Calvano* Giacomo Calzolari † Vincenzo Denicolò‡ Sergio Pastorello§

July 11, 2020

1 Introduction

This document illustrates how to use the code for Calvano et al. (2019). It provides an overview of the software, instructions on how to install it and a guide on how to use it.

The code and all associated files are available at the dedicated OPENICPSR workspace. For future updates of the code post-publication, please refer to `https://sites.google.com/view/giacomo-calzolari/research/algorithmsaer2020`.

All the computations involving simulation experiments in Calvano et al. (2019) were completed using FORTRAN programs, with output that are text files containing summary statistics or detailed information on the simulation experiments. These text files are then fed to R scripts which must be executed within RStudio. This generates the plots and tables in the paper and the online appendix.[1]

The code-base contains the source code to generate three executable files. The first executable, `baseline.exe`, performs the computations for all experiments with two exceptions: the stochastic demand case (section 6.3 in Calvano et al., 2019) and the variable market structure case (section 6.4). These two environments are handled by two separate executables, `stochasticdemand.exe` and `entryexit.exe`. These three programs work in a similar way, but a few of the features available in the first one are not available in the latter two.

*Bologna University, CEPR and Toulouse School of Economics
†European University Institute, CEPR, Bologna University and Toulouse School of Economics (Corresponding author)
‡Bologna University and CEPR
§Bologna University
[1]We have used version 4.0.2 for R and for 1.3.959 for RStudio.

One can compile and link the source code, or directly use the ready-to-use executables created on a Windows computer that we provide. In section 2 we describe how to build the executables and how to use them.

The executables read the parameters from the external text file `A_InputParameters.txt`, which must be located in the same folder as the executable. This file is described in section 3. The working of the code, its routines and the output files are described in section 4. Section 5 illustrates how to use the output files and run the R scripts to generate figures and tables of the paper and the online appendix. Section 6 shows how to familiarize with the code and illustrates how to replicate one of the tables in the paper.

The structure of the folder of the code, `\AER_code`, is as follows:

- `\AER_code\AERsoftwareguide` contains this softer guide.
- `\AER_code\AER_fcode` contains three folders with the FORTRAN codes to generate each of the three executable and the executable compiled files as well.
- `\AER_code\AER_online_appendix_Rscripts` contains several folders, one for each figure or table of the online appendix which contain the `A_InputParameters.txt` files already tailored to the specific task (figure or table) and the associated R scripts.
- `\AER_code\AER_paper_Rscripts` contains several folders, one for each figure or table of the paper, which contain the `A_InputParameters.txt` files already tailored to the specific task (figure or table) and the associated R scripts.

The two files, "Table replication paper.pdf" and "Table replication online appendix.pdf", describe for each table and figure, respectively in the paper and in the online appendix, all associated files: the Fortran executable to run, the R/Rstudio script(s) and the needed Input file(s) generated by the executable, and the expected final Output file(s).

2 Building and running the executable programs

We generated the 64 bit executables using the Intel Visual FORTRAN Compiler XE 13.1.2.190 from within the Microsoft Visual Studio IDE. We parallelized selected portions of the code using OpenMP directives. Object modules were generated with the options

```
/nologo /O2 /fpp /Qopenmp /module:"x64\Release\\"
/object:"x64\Release\\" /Fd"x64\Release\vc100.pdb"
/libs:static /threads /c
```

and the linker options were

Table 1: Code files' names for each program

baseline	stochastic demand	entryexit
main.f90	main.f90	main.f90
globals.f90	globals.f90	globals.f90
generic_routines.f90	generic_routines.f90	generic_routines.f90
PI_routines.f90	PI_routines.f90	PI_routines.f90
QL_routines.f90	QL_routines.f90	QL_routines.f90
LearningSimulation.f90	LearningSimulation.f90	LearningSimulation.f90
ConvergenceResults.f90	ConvergenceResults.f90	ConvergenceResults.f90
ImpulseResponse.f90	ImpulseResponse.f90	ImpulseResponse.f90
EquilibriumCheck.f90	EquilibriumCheck.f90	EquilibriumCheck.f90
QGapToMaximum.f90	QGapToMaximum.f90	QGapToMaximum.f90
DetailedAnalysis.f90	DetailedAnalysis.f90	DetailedAnalysis.f90
LearningTrajectory.f90		

```
/OUT:"x64\Release\[PN].exe" /INCREMENTAL:NO /NOLOGO /MANIFEST
/MANIFESTFILE:"[PF]\x64\Release\[PN].exe.intermediate.manifest"
/MANIFESTUAC:"level='asInvoker' uiAccess='false"'
/SUBSYSTEM:CONSOLE /STACK:800000000,800000000
/IMPLIB:"[PF]\x64\Release\[PN].lib"
```

where `[PF]` and `[PN]` stand for the project's folder name and the project's name, respectively.

The code files to be compiled and linked to generate the three executables are listed in Table 1. Files `main.f90` contain the calling program, all others contain modules implementing specific computations, which will be detailed in the next sections. These code files are explained in details in Section 4 of this guide. Even when the name of the code files are the same across the three executables, there are slight differences in the code, designed to accomodate specific features of each task, as we explain next.

To create the executables from the source code it is necessary to have a FORTRAN compiler installed. Alternatively, one could use the executables we provide, designed to work on Windows computers and requiring the preliminary installation of the Intel FORTRAN Compiler 2017 redistributable libraries available at `https://software.intel.com/en-us/articles/redistributables-for-intel-parallel-studio-xe-2017-composer-edition-for-windows`.

Running the executable for some of the tasks (e.g. to draw heat maps) may require a significant amount of time (up to a few days on a 40 cores workstation) and may generate very large output files (in some cases several gigabytes). Also, to create some tables it might be necessary to run the same executables with different common parameters (e.g., Table A8 in the appendix on the effect of varying the substitutability parameter μ) or different executables (e.g., Table A6 in the appendix on the effects of allowing for stochastic demand). In these cases the folder corresponding to these tasks contains multiple input parameters files that must be used with the appropriate executable.

3 The input parameters file

The first step to run the code and generate the results for an experiment is to provide the executable with the input parameters defining the task. This is done through the input text file `A_InputParameters.txt`. The input file can be created or modified with any text editor, although we found it convenient to work on it using a spreadsheet such as Microsoft Excel. Using a single input file, it is possible to run several experiments sharing the same common base characteristics (i.e. number of agents, number of available prices, type of demand function, ...) but with different learning or demand function parameters.

The input parameter file is divided in two parts. The first one defines values for the parameters which are common to all experiments. They are read at the beginning of the code execution, and must be provided in the line immediately below their label (labels are skipped by the programs). The second part contains values for the parameters that are specific to each experiment; they must be provided on a single line. The second part starts with a line containing column name labels, which is skipped by the program.

The following two sections illustrate the two parts of the input parameter file in more detail.

3.1 The "common parameters" section

Table 2 lists the parameters contained in the first part of the input file, including a brief description of the operations they control, in the order they must appear. A few parameters are available in some executables only; the last column reports the executable(s) for which each option is available. Note that, as explained above, the parameter value must be provided in the (even-numbered) line immediately after the parameter label (odd-numbered) line, as in the input parameters files that we provide.

Table 2: Common parameters description

Label	**Description and Commands**	**Available in**
Number of experiments	The following line must contains two parameters, denoted M_1 and M_2. M_1: Number of experiments M_2: "Total" number of experiments. This is used when the M_1 experiments to be run are part of a larger set of M_2 experiments. This can be useful when for some reason a set of experiments is split into a number of subsets, e.g., to distribute the entire task across multiple computers. If this is not the case, just set $M_1 = M_2$.	All
Number of cores	Number of CPU cores used in parallelized sections of the code. It can be set to 1 to avoid using more than one core. In general, this parameter should never be larger than $C-1$, where C is the number of available cores.	All
Number of sessions	Number of simulation sessions in each experiment.	All
Iterations per episode	Number of iterations in each episode. Episodes are groups of iterations, which allow to count iterations with a simpler scale, e.g., episodes could be days and iterations seconds. Although one could set this parameter to any positive integer number, to avoid errors in counting the number of iterations we recommend to set it to a large integer. We used 25,000.	All
Maximum number of episodes	Maximum number of episodes allowed for the learning algorithm. (This also implicitly determines the maximum number of iterations.)	All
Performance measurement period length	Number of iterations in which strategies must not change.	All
Number of agents	Number of agents.	All
Memory	Length of memory, in periods.	`baseline`
Number of prices	Number of prices.	All

Number of markets	Number of a_0 levels in the case of stochastic demand experiments.	`stochasticdemand`
Type of Exploration Mechanism	Type of exploration. Use one of the following numbers: 1 for exponentially decreasing probability of exploration 2 for Boltzmann exploration In both cases, the exploration probability is determined by the β parameters provided in the Experiment Parameters section (see sect. 3.2 for details).	`entryexit` and `stochasticdemand` only allow exponentially decreasing probability of exploration (option 1)
Type of input for payoffs	Method to compute the rewards. Use one of the following numbers: 1 for linear demand (Singh and Vives, 1984) 2 for logit demand 3 for logit demand with perfect substitutability ($\mu = 0$)	`entryexit` and `stochasticdemand` only allow logit demand (option 2)
Compute Impulse Response analysis with a one-period deviation to static Best Response	Binary flag. If 1, computes the average Impulse Response (IR) with a one-period deviation to static Best Response; if 0, the average IR is not computed.	`baseline`
Compute Impulse Response analysis with a temporary or permanent deviation to Nash	Indicates whether and how to compute the average Impulse Response (IR) to deviation to Nash price: • 0: the average IR to a Nash deviation is not computed • 1000: for computing the average IR with a permanent deviation to Nash • $0 < X < 1000$: for computing the average IR with a X period deviation to Nash	`baseline`

Compute Impulse Response analysis with a one-period deviation to all prices	Binary flag. If 1, computes the average Impulse Response with a one-period deviation to all prices. If 0, this step is skipped.	`baseline`
Compute Equilibrium Check	Binary flag. If 1, computes the summary statistics on the frequency of best response and equilibrium play on several subsets of the state space. If 0, this step is skipped.	`baseline`
Compute Q Gap w.r.t. Maximum	Binary flag. If 1, computes the average difference between learned and true Q values on several subsets of the state space. If 0, this step is skipped.	`baseline`
Compute Learning Trajectory	This must contain two parameters, denoted L_1 and L_2. L_1: If positive, the program records and writes to file L_1 average intermediate profit gains and other information at the beginning of the learning trajectory. If 0, this step is skipped. L_2: If $L_1 > 0$, $L_2 > 0$ is the length of the interval between intermediate observations used for the computation performed in the previous point.	`baseline`
Compute detailed analysis	Binary flag. If 1, computes and writes to file detailed information about the results of the limit strategies of each individual session in the experiment; if 0, no such activity performed.	All

3.2 The "experiment parameters" section

Table 3 lists the parameters contained in the second part of the input file. This part begins at the line immediately below that containing the value of the last parameter of the "common parameters" section, and always starts with a row of labels. The exact content of the labels row depends on the executable and on the values of some of the parameters specified in the common parameters section of the input file. As usual, the labels are not important, as the entire line is skipped by the program. It is however necessary that the experiment parameters are provided in the exact order. Everywhere, we denote with N the number of agents in the experiment (as

specified in the common parameters section of the input file).

Table 3: Experiment parameters description

Label	**Description**	**Available in**
Experiment	A code denoting the experiment, used in output file. Can be any integer. In the following lines of this table this numeric code will be denoted with `XXXX`.	All
PrintQ	Binary flag. If 1, the program writes to disk as many text files as sessions, each one containing the Q matrices at convergence of all agents (on top of each other). The files are named `Q_XXXX_YYYY.txt`, where `YYYY` is the session number.	All
Alpha1 ⋮ AlphaN	Learning rate parameters for the N agents, which can differ across agents.	All
Beta_1 ⋮ Beta_N	A first set of N parameters determining the probability of exploration, which can differ across agents. • When Type of Exploration Mechanism (see above) is equal to 1, the probability of exploration is computed as $\exp(-\beta t)$, where t is the iteration index. In this case Beta_1, ..., Beta_N are the β coefficients for each agent. Notice that these parameters must be specified in the scale defined by Iterations per Episode in the first section of the input parameters file. For example, consider the benchmark model in the paper in which $\beta = 4 \times 10^{-6}$, and assume that Iterations per episode$= 25,000$. In this case Beta_i should be specified as $4 \times 10^{-6} \times 2.5 \times 10^4 = 0.1$. • With Boltzmann exploration (Type of Exploration Mechanism equal to 2), the temperature is computed as $\tau = 1000 \times \beta^t$. In this case Beta_1, ..., Beta_N are the β coefficients for each agent. Notice that these parameters must be specified as $\log_{10}(1-\beta)$. For example, consider the case in which $\beta = 0.999$; then, Beta_i should be specified as $\log_{10}(1-0.999) = -3$.	All

Delta	The discount rate δ. This is set to zero for zero-memory experiments.	All
a0	Only used with logit demand. Specifies the a_0 parameter of the demand function.	`baseline`, `entryexit`
a0_1 $\vdots$ a0_M	Only used with logit stochastic demand, specifies M different values for the the a_0 parameter, with M fixed in Number of Markets in the common parameters section.	`stochasticdemand`
a1 $\vdots$ aN	Only used with logit demand. Specifies the vertical differentiation parameters a_i, $i = 1, \ldots, N$.	All
c1 $\vdots$ cN	Only used with logit demand. Specifies the marginal cost parameters c_i, $i = 1, \ldots, N$ (effects of costs in the linear demand models are accounted with the demand intercept parameters, without loss of generality).	All
mu	Only used with logit demand, the horizontal differentiation parameter μ. If the Type of payoff input in the common parameters section has been set to 3, μ must be set to 0.	All
gamma	Only used with linear demand, specifies the differentiation parameter γ in the quadratic utility function (it ranges from 0 to 1).	`baseline`
extend1 extend2	Two parameters determining the width of the prices grids. • If both are positive, they specify ξ_1 and ξ_2 mentioned in sect. 3.2 of the paper. • If extend1 is negative and extend2 is nonnegative, then the lowest price is 0 with linear demand and $c_i(1+\xi_1)$ with logit demand, while the highest price is $p_i^{Coop}(1+\xi_2)$ with linear demand and $p_i^{Coop} + \xi_2(p_i^{Coop} - p_i^{Nash})$ with logit demand.	All, but only the first option is available in `entryexit` and `stochasticdemand`
EntryProb ExitProb	Specifies the ρ_1 and ρ_2 parameters in section 6.4 of the paper, the conditional probabilities of entry and exit of the outsider agent. We use $\rho_1 = \rho_2$.	`entryexit`

ac(a_0)	The autocorrelation of consecutive a_0 values of the stochastic demand model.	`stochasticdemand`
NashP1 ⋮ NashPN	List of N Nash prices, one for each agent (computed with a Mathematica script available upon request).	`baseline`
NashP1In ⋮ NashPNIn NashP1Out ⋮ NashPNOut	List of $N + N$ Nash prices, two for each agent. The first N apply when the outsider is in the market, the second N when it is out of the market (computed with a Mathematica script available upon request).	`entryexit`
NashP1_1 ⋮ NashP1_M ⋮ NashPN_1 ⋮ NashPN_M	List of $M \times N$ Nash prices, M for each agent, each one corresponding to a different a_0 value in the stochastic demand case (computed with a Mathematica script available upon request).	`stochasticdemand`
CoopP1 ⋮ CoopPN	List of N monopoly prices, one for each agent (computed with a Mathematica script available upon request).	`baseline`
CoopP1In ⋮ CoopPNIn CoopP1Out ⋮ CoopPNOut	List of $N + N$ monopoly prices, two for each agent. The first N apply when the outsider is in the market, the second N when it is out of the market (computed with a Mathematica script available upon request).	`entryexit`
CoopP1_1 ⋮ CoopP1_M ⋮ CoopPN_1 ⋮ CoopPN_M	List of $M \times N$ monopoly prices, M for each agent, each one corresponding to a different a_0 value in the stochastic demand case. Usually computed using a Mathematica script (available upon request)	`stochasticdemand`

TypeQ1 Par1Q1 ⋮ ParNQ1 TypeQ2 Par1Q2 ⋮ ParNQ2 ⋮ TypeQN Par1QN ⋮ ParNQN	A set of N blocks of $N+1$ parameters, each one specifying how to initialize the Q matrix of each agent. Some initialization methods work by selecting a value for each Q cell; others specify a strategy for each agent, and compute the *true* Q matrix corresponding to that set of strategies (see the paper for more details). Different agents can have their Q matrix initialized differently. The possible values and meaning of the ParjQi parameters depend on the associated TypeQi parameter used for agent i. The latter can be set as follows. TypeQi is a one-character string. Possible values are: **F** The Q matrix of the i-th agent is computed assuming that agent j selects the ParjQi-th price in every state, $j = 1, 2, \ldots, N$. **G** The Q matrix of the i-th agent is computed under a *grim trigger* strategy assumption: if at the previous period all agents selected the PariQ1-th price, they keep playing the same price, otherwise they play the PariQ2-th price forever. **O** The Q matrix of the i-th agent is computed as the discounted payoff that would accrue to it if opponents randomized uniformly. This is described in formula (8) in the paper. **T** The Q matrix of the i-th agent is randomly drawn from the set of Q matrices learned upon convergence by the same agent in a previous experiment, with Experiment code PariQ1, and stored in the `trained_Q/` subfolder. **R** Each cell of the Q matrix of the i-th agent is initialized with a random draw from a uniform distribution between PariQ1 and PariQ2. **U** Each cell of the Q matrix of the i-th agent is initialized at PariQ1.	All

4 The structure of the code and output files

This section describes the content of the files in the `baseline` executable. As mentioned in table 2 and 3, some of the features of this executable are not available in the `entryexit` and `stochasticdemand` versions of the code (and vice-versa). We also describe how each module delivers the outcomes of the simulation to external `.txt` output files that can then be read with standard applications (like spreadsheets, editors, or other software).

4.1 `main.f90`

This file contains the calling program that calls other modules. Depending on the input parameters provided by the external file `A_InputParameters.txt`, the executable will compute summary statistics across several simulation sessions or store in a text file detailed information on each session.

4.2 `globals.f90`

This module defines some global variables and contains subroutines that read the two sections of the input parameters file and allocate and deallocate memory for the global variables.

4.3 `generic_routines.f90`

This module contains some general purpose functions and subroutines. One of them (`ran2`) generates the uniform random numbers used in the simulations. This is the RAN2 function in Press et al. (1992), p. 272, modified to allow the parallelization of the code.

4.4 `PI_routines.f90`

This module contains functions to compute the price grid and the reward matrix of each agent. With linear demand the price grids and the reward matrices are the same for all agents and they are computed by the program. The formulae used to define the price grids are detailed above and in the paper.

With logit demand the price grids and the payoff matrices can differ across agents and are computed by the program. In the perfect substitutability case ($\mu = 0$) the demand and associated payoffs are exactly computed in a separate subroutine designed to handle this case.

4.5 `QL_routines.f90`

This module contains functions and subroutines used repeatedly in the routines implementing the learning stage and in those evaluating the performance of the trained algorithms. The two most important are `ComputeQCell`, which given a strategy for each agent, a current state and a current action for an agent computes the true value of the corresponding cell in its Q matrix, and `initQmatrices`, which initializes the Q matrices at the start of the Q-Learning iterations.

4.6 LearningSimulation.f90

This module contains two subroutines: `computeModel`, which implements the Q-Learning iterations, and `computePPrime`, which implements the exploration mechanism. In the former, the loop over the simulation sessions is parallelized, allowing a significant reduction in computing time.

When exiting computations, `computeModel` always generates three types of text output files. The first one consists of the file `A_res.txt`. This file collects some preliminary descriptive statistics on the results of the training. It features one experiment for each line and the last four columns contain:

- `numConv`: number of sessions attaining convergence
- `avgTTC`: average across sessions of the number of episodes needed to attain convergence
- `seTTC`: standard deviation across sessions of the number of episodes needed to attain convergence
- `medTTC`: median across sessions of the number of episodes needed to attain convergence

The second type of output file is named `InfoExperiment_XXXX.txt`, where `XXXX` is the experiment code. Differently from the previous file (`A_res.txt`), there is one `InfoExperiment` file for each experiment; they are used in `ConvergenceResults.f90`, described below. They contain detailed information on each simulation session, specifically: the session number, a binary flag to indicate convergence, the number of episodes to reach convergence, the last state visited by the algorithms and the strategies upon convergence.

Finally, if PrintQ is set to 1 in the second column of the experiment parameters section, `computeModel` writes to file the Q matrices at convergence of all agents, for all sessions. They can be used as an alternative initialization method for the Q-Learning iterations, as discussed in section 7.2 of the paper.

4.7 ConvergenceResults.f90

This module contains a single subroutine, `ComputeConvResults`, which reads from the disk the `InfoExperiment` files generated by `computeModel` and identifies the limit path followed by the algorithms according to their strategy at convergence. This allows to compute the averages across sessions of the profit accruing to each agent and their standard deviations, the corresponding profit gains and the frequency of each state being visited in a limit path. All these informations are averages across sessions and are written as output in the last columns of the text output file `A_convResults.txt`.

The same information, but at the individual session level (i.e., without averaging over the sessions) are written to an output file with the same name as the file generated by `computeModel` and described in the previous section. For each simulation session, the new file contains: the session number, the convergence flag, number of episodes to convergence, the length (in periods)

of the limit path cycle, the states visited, the prices selected and the profits experience by each agent in the limit path, together with the strategies at convergence.

4.8 `ImpulseResponse`

This module contain four subroutines used to compute Impulse Responses, i.e. trajectories of states, prices and profits visited by the agents along the path moving from a given state to a (possibly cyclic) limit path. Even though the paper focuses on unilateral deviations from the limit path, the code allows to consider arbitrary starting states which may correspond to deviations by more than one agent from their respective limit path. The final cycle can be the same one as identified by `ComputeConvResults` (indeed, this is almost always the case) or another one.

Depending on the types of Impulse Responses required by the binary flags in the first section of the input file, these subroutines generate some text output files named `A_irToBR.txt`, `A_irToNash.txt` or `A_irToAll.txt`. Impulse Responses are computed both for prices and profits, and provided both at the individual agent level and distinguishing between the "Deviating agent" and the "Nondeviating agents" (with $N > 2$ agents, the latter averages over $N - 1$ of them). Prices are stored in the following columns:

- `AggrPricePre`: average (across agents and sessions) pre-deviation price
- `AggrDevPriceShockPerYYY`: average (across sessions) price of the deviating agent in period `YYY` (the deviation occurs in period 1)
- `AggrNonDevPriceShockPerYYY`: average (across sessions and possibly $N-1$ agents) price of the nondeviating agent(s) in period `YYY`
- `AggrDevPricePost`: average (across sessions) price of the deviating agent upon convergence to a limit cycle
- `AggrNonDevPricePost`: average (across sessions) price of the nondeviating agent(s) upon convergence to a limit cycle

Columns with a similar name but with `se` prepended contain standard deviations.

4.9 `EquilibriumCheck.f90`

This module contains two subroutines that check whether the strategies learned by the agents correspond to best response or equilibrium behavior. Best Response (BR) is ascertained separately for each agent by checking, state by state, whether the action chosen by the strategy at convergence is the one yielding the largest *true* Q value, computed by subroutine `ComputeQCell` in module `QL_routines`, and then aggregating over a set of states. Equilibrium (EQ) occurs when all agents are simultaneously best responding. As sets of states, we consider the whole state space, the limit path states and the off limit path states separately.

The text output file is named `A_ec.txt`. Two types of statistics are computed and stored. Those with name starting with `Flag` are averages across sessions of binary flags equal to 1 when BR or EQ occurs on *all* the states in the subset under consideration, denoted by `All`, `OnPath` or `OffPath`. The statistics with name starting with `Freq` contain the *percentage* of states in which BR or EQ occurs in the subset under consideration. Each statistic is also reported disaggregated by the limit cycle length – this is denoted with the string `lenZZ` in the statistic's name, where `ZZ` is the length of the limit cycle and `ZZ` $= 0$ denotes all lengths – and, for BR, by the agent – denoted with the string `AgY` in the name, where `Ag0` denotes all agents.

4.10 `QGapToMaximum.f90`

The two subroutines in this module compute the loss incurred by the agents when following a suboptimal strategy. As for `EquilibriumCheck`, the loss is computed on the basis of the *true* Q matrix associated to the strategies at convergence, and it is reported in the form of averages over subsets of sessions (identified by the length of the limit cycle) and agents. In addition to the entire state space, the subset of states in the limit cycle and its complement, we also consider four additional subsets of states (all those in which at least one agent is not best responding, those in the limit cycle in which at least one agent is not best responding, all those in which the agents' behavior is not an equilibrium and those in the limit cycle with the same property). The paper focuses on the first three statistics, which are written to the text output file `A_qg.txt` in the columns labeled `QGapTot`, `QGapOnPath` and `QGapNotOnPath`. Summary statistics disaggregated by limit cycle length and by agent are denoted with the strings `PL` and `Ag` in the columns names, respectively.

4.11 `DetailedAnalysis.f90`

The subroutine in this module computes several statistics at the individual session level, i.e. without averaging or otherwise aggregating over all sessions or a subset of them. For each session, several lines are written to the file reporting most of the statistics mentioned thus far (prices, profits, profit gains, statistics about the occurrence of equilibrium behavior, the Q loss due to suboptimal response, impulse responses to deviation to all prices, and others) for each possible observed agent, deviating agent and initial state (among those contained in the limit cycle). The text output file is named `A_det_XXXX.txt`, where `XXXX` is the experiment code. The output files generated by this code are frequently quite large, and may take up to several GB of disk space.

4.12 `LearningTrajectory.f90`

This module contains one subroutine, `computeLearningTrajectory`, which can only be used by the `baseline` executable. Its purpose is to compute statistics about the performance in terms of profit gain (`PG`), strength of the response of nondeviating agents in the period following a deviation (`IR`) and the frequency of validity of the Incentive Compatibility condition in the

subset of limit cycle states (`IC`) in the early stages of the learning process. The code first computes for each session the averages over blocks of consecutive iterations of the three indicators, and then aggregates across sessions using averages, standard deviations and selected quantiles. The number of intermediate measurements and the length of the blocks of iterations is specified under the label "Compute Learning Trajectory" in the first section of the input parameters file. The final trajectories are written to the text output file `LTrajectories_XXXX.txt`, where `XXXX` is the code of the experiment.

5 Running the R scripts

In addition to the FORTRAN code, we provide the input parameters files and the R script that are specific to the figures and the tables of the paper and the online appendix.

For each of these elements there is a specific folder that contains the input parameter file that is already tailored to the specific task.

Running the executable in the same folder of the task produces the task-specific output txt file. We do not provide these task-specific output txt files for each task because these are often very large files, in some cases several gigabites.

With the task-specific output txt file, by running the task-specific R script in the same folder produces either a pdf file for a figure, or a txt file for a table, depending on the task.

For some tasks, a "`..._load_data.R`" script is available. In these cases, this script must be run befor running the task-specific R script.

6 An example

To familiarize with the code, one could begin with the following example that allows to replicate Table I in the paper. The same procedure can be replicated for all other figures and tables in the paper and in the online appendix.

1. Copy the relevant executable, in this case `baseline.exe`, to the folder of the specific task, i.e. `...\AER_code\AER_paper_Rscripts\table_I`.

2. Run that executable by double clicking on it. This operation will generate a few task-specific output txt files.

3. Run the task-specific R script that is available in the same folder for table I, i.e. (`...\tableI\table_I.R`). This will generate a formatted txt file of Table I.

References

Calvano E., Calzolari G., Denicolò V. and Pastorello S. (2019), Artificial intelligence, algorithmic pricing and collusion, working paper, University of Bologna, Italy.

Press, W.H., Teukolsky S.A., Vetterling, W.T. and Flannery B.P. (1992), *Numerical Recipes in Fortran 77 (2nd ed.)*, Cambridge University Press.