How to get started quick?

What is the NESFI-ASM?

The NESFI-ASM (Norman Ehrentreich's Santa Fe Institute Artificial Stock Market Model) is an adapted version of the original SFI-ASM. There are several `generations' of the SFI-ASM on different programming platforms. An overview over the SFI market history can be found in LeBaron (2002) and Johnson (2002). A current objective-C version of the SFI-ASM using the Swarm package is maintained by Paul Johnson. It has been developed since 1989 and has been described in various papers (Arthur et al. 1997; LeBaron et al. 1999; Joshi, Parker, and Bedau 1998, Joshi, Parker, and Bedau 2002).

The NESFI-ASM corrects a serious design flaw in the mutation operator of the original SFI-ASM. This mutation operator caused an upward bias in the resulting bit distribution of the classifier system, thus suggesting increased levels of technical trading for smaller GA-invocation intervals. The NESFI-ASM, on the other hand, partly supports the Marimon-Sargent-Hypothesis that adaptive classifier agents in an artificial stock market will always discover the homogeneous rational expectation equilibrium. While agents always find the correct solution of non-bit usage, analyzing the time series data still suggests the existence of two different regimes depending on learning speed. The NESFI-ASM can be run both in SFI- and NESFI-mode in order to compare the different behavior of both versions.

If you are interested in a more in depth description of the problem and the results of the corrected version, you can download my working paper "*The Santa Fe Artificial Stock Market Model Re-Examined – Suggested Corrections*" at IDEAS, EconWPA, or EconPapers.

Requirements

The NESFI-ASM is programmed in Java and uses the Repast library. RePast is a product of the University of Chicago's Social Science Research Computing. The name Repast is an acronym for *REcursive Porous Agent Simulation Toolkit*.

In order to compile and run the NESFI-ASM, you need both Java 1.3 (or higher) and Repast 1.4.1 installed. To download Java™ 2 Platform, Standard Edition (J2SE™), version 1.4.1, click here. To download Repast, click here.

Installation

To install Java and Repast, please refer to the installation instructions that come with both packages. There are very good installation instructions provided with the Repast library, a good documentation on how to get started with Repast as well as a faq-list. I recommend to start with the "Repast How Tos" since they give you a quick overview how to write your first simulation. Maybe, before trying to compile the NESFI-ASM, you could even write your own little simulation or you can run some of the various demo programs that come with Repast.

The main file of the NESFI-ASM is `AsmModel.java`. It has all the Repast specific stuff in it and controls the model. If you have any problems compiling the NESFI-ASM, I recommend to resort to the Repast documentation.

How to run the NESFI-Model

The model can be run either in GUI-mode or in batch mode. About the different ways to start a Repast simulation, please read the "How to start a Repast Simulation" provided with Repast. I just provide two batch files with which you should be able to immediately start the model either in batch mode (runbatch.bat) or in GUI mode (runGUI.bat). Make sure you have eventually changed the classpath information before in the batch files. Right now it is assumed that Repast is installed in D:\Repast.

Model Structure

Class Hierarchy:

- AsmModel extends SimModelImpl
- Agent
  - SFIAgent extends Agent
  - NESFIAgent extends Agent
  - FastAgent extends Agent
- abstract class Asset
  - Stock extends Asset
- World
- Specialist
- TradingRule
- ExecutePeriod
- ObserverOptions
- RecorderOptions
- RecorderParamFileReader
- Stats

Class Descriptions

AsmModel.java:

This is the main file of the NESFI-ASM. It implements the SimModel interface (not the SimpleModel) and contains all the methods that are required for that model type.

The Repast template methods are

- `private void buildModel()`: Builds all the objects that are required for the simulation. If data recording is switched, it initializes the DataRecorder which is responsible for writing data into an ASCII-file.

- `private void buildDisplay()`: If `showDisplays` is `true`, then all the initializations for the displays are executed.

- `private void buildSchedule()`: Creates a schedule for the entire simulation. A schedule determines what has to be executed at every time step of the simulation and what at only at certain times or intervals.

The interface methods that are required by SimModel are

- `public String[] getInitParam()`: AsmModel implements the CustomProbeable interface whose only method is getInitParam(). In GUI-mode, it returns a string with the names of all the variables that should be displayed in the NESFI-Settings windows. In batch-mode, it returns a different string. These parameters belong to the parameter objects, but Repast doesn't allow to call the get-and set accessor methods of these objects. Thus, a second set of these get- and set accessor methods are included in AsmModel.java such these parameters can be set over the parameter file. The get- and set accessor methods have to be public and are named by adding "set" or "get" before the capitalized variable name.

- `public void begin()`: Initializes the model for the start of a run. It calls `buildModel()`, `buildSchedule()`, and maybe `buildDisplays()`.

- `public void setup()`: Prepares the model for a new run. Thus, it "tears down" all previous variables, dispose graphs and schedules.

- `public Schedule getSchedule()`: Required by Repast.

- `public String getName()`: Returns the name of the model to be displayed in the GUI control bar.

There are a few other methods in AsmModel.java. One of them is `conditionalStop()` which is scheduled only at certain intervals. Here, I wanted to stop the model and record the period whenever all agents have completely abandoned technical and fundamental bit usage.

`checkVariables()` started out as a debugging procedure is kept active to have some console outputs for long simulation runs, especially in batch mode. You can easily disable it in `buildSchedule()`.

Agent

Contains all the variables and procedures that are common to all agents. Some of the procedures are overwritten in the derived classes.

SFIAgent extends Agent, NESFIAgent extends Agent, FastAgent extends NESFIAgent

Some methods that differ are `invokeGA()` or `chooseRule()`, for instance. FastAgents are like NESFI-Agents, except that they invoke the GA at a different speed (assumed: more often, though it could be the other way round).

abstract class Asset, Stock extends Asset

It contains all basic methods for different assets. The structure with an abstract asset class and one single derived class is due to the initial design of a multi-asset model. In fact, this version has been stripped down from an existing two stock version in which I create two stocks from the stock class, and an index, that also extends asset.

World

There is one world object created in the buildModel() method. It holds certain parameters that describe the economic environment of the model, e.g., interest rate, periods to run, how many agents of what kind, etc. It also has some methods to gather and report data that make only sense on a "world" level, e.g., total number or fractions of bits set, total wealth level, trading volume etc.

Specialist

The specialist handles market clearing. There is one instance created and the object finds a market clearing price. From the various types of the original SFI-ASM, only the hree-specialist and the slope specialist have been implemented.

TradingRule

Every trading rule that an agent possesses is an actual instance of this class. It has a lot of static variables for the GA and every rule handles its own fitness calculation. The class TradingRule implements the Comparable interface such that rules can compared and ranked according to their fitness.

ExecutePeriod:

This class is defined abstract and contains the action sequence within a period. The main method `execute()` of that class is called every period by the Repast scheduler.

ObserverOptions

This class allows to switch display options on or off over the GUI. In order to have it not included in the main NESFI-Settings window, a separate object is created which then can be probed from the NESFI-Settings windows. ObserverOptions basically holds Boolean values which can be set through their appropriate get- and set accessor methods. The variable names are intended to be self explanatory, yet the available options will be shortly described in the follwing:

- ShowAgentGrid: Places agents on a rectangular grid such that they can be probed for certain agent variables. I used this only for debugging purposes and has no other purpose here. The displayed variables can be easily extended by adding the appropriate get-method.
- ShowBitFractions, ShowFundamentalBits, ShowTechnicalBits: If ShowFundamentalBits or ShowTechnicalBits is turned on, they are displayed in a separate observer window. You can have absolute numbers of set bits in the whole economy displayed, or, if ShowBitFractions is true, the ratio of current set bits over maximum number of set bits is displayed. Turning this on uses a lot of computational time, though.

- ShowCrudePrice: Is defined as dividend over interest rate.
- ShowHreePrice: The homogenous rational expectations equilibrium price is set by the specialist in the hree-regime. It can be displayed in other regimes in order to serve as a benchmark.
- ShowHreePrice_Price: Shows the difference between the realized price and the hree-price. This could be considered as proxy for a risk premium and should come close to zero in the long run.
- ShowPrice, -Dividend, -PriceMAxx, -Price/DividendMean, should be self explanatory. They are all displayed in the same observer window.
- ShowLogReturns: Displays the logged returns of the realized prices and hree-prices after `World.firstPeriodForShowingLogs`. An offset is added to the hree-logs in order to separate the two series.
- ShowVolume: Displays trading volume in a separate observer window.
- ShowPriceValueCorr: Displays the correlation between the price series and the theoretical hree-prices. For the correlation calculation, only the last `Asset.memory` periods are taken into account. This variable can be set from the main window of the GUI settings. Turning this on uses a lot of computational time, though.
- ShowWealthxxxx: Displays average wealth levels of certain types of agents.
- UpdateFrequency: Updating the screen costs a lot of computation time. Thus, this can be scheduled to be done only at certain intervals. All recorded values are displayed then at once.

RecorderOptions

This class allows to switch data recording options on or off over the GUI. In order to have it not included in the main NESFI-Settings window, a separate object is created which then can be probed from the NESFI-Settings windows. RecorderOptions basically holds Boolean values which can be set through their appropriate get- and set accessor methods. If a parameter file exists, the preset values in the source code can be overwritten. The variable names for the recorder options should be self explanatory and are often identical to the variable names for the visual display. The ones that differ or that are added are explained below:

- ActiveRules: the more bits are set in the economy, the more specific are the trading rules, matching only certain states of the world. This options redords the total number of activated rules in the economy.
- BitAnalyzer: This records how often a each bit is set in the economy. Data files become soon very large if this option is switched on and it also uses a lot of computational time. It might be interesting to check bit usage for periodic or the random walk dividend process.
- Max-,Mean-, and MinFitness: Record some fitness-statistics across all rules in the economy.
- PVCorr: Equals ShowPriceValueCorr of the observerOptions.
- WriteFrequency: Writes the recorded data only at certain intervals to the disk.

- RecordFrequency: Sometimes one wishes to record data only at certain intervals. If, for instance, recordFrequency is set to 100, data will be recorded only every 100 periods, all other 99 periods will be neglected.
- StartFromPeriod: Additionally, one wants to wait until the model has come close to its equilibrium and record only those data. Then set StratFromPeriod to an appropriate number.
- RecordAllFromPeriod: If RecordFrequency is bigger than one, one can switch to complete data recording by setting an appropriate value.
- RecordAllToPeriod: Complete data recording can be switched off again such that one returns to periodic data recording.
- RecorderOutputFile: Specifies the complete filename of the output file.

RecorderParamFileReader

If a parameter file exists, it is read by this class and sets the recording option in RecorderOptions. This is intended for batch runs, but a parameter file can also preset the Boolean values for the GUI. The parameter file itself is a plain ASCII-file in which each line can set a certain option on or off. Please make sure that there are no empty lines or wrong variable names. No sophisticated error checking has been implemented yet. An example of a parameter file is given below:

```
price: true
fundamentalbits: true
technicalbits: true
dividend: false
hreeprice: false
crudeprice: true
pvcorr: false
tradingVolume: false
meanTradingVolume: false
bitFractions: false
activeRules: false
meanFitness: false
maxFitness: false
minFitness: false
bitAnalyzer: false
averageWealth: false
averageWealthOfClassifierAgents: false
averageWealthOfNoClassifierAgents: false
averageWealthOfFundamentalTraders: false
averageWealthOfTechnicalTraders: false
averageWealthOfFastLearner: false
averageWealthOfNormalLearner: false
averageWealthOfSFIAgents: false
averageWealthOfNESFIAgents: false
writeFrequency: 2
recordFrequency: 1
startFromPeriod: 5000000
recordAllFromPeriod: 5000000
recordAllToPeriod: 5000000
```

This example looks a little bit weird since no data seem to be written at all since the length of a simulation was set to 5000000. However, since stopAtZeroBit in the NESFI-Settings windows was turned, the simulation was stopped when bit usage was completely abandoned in the economy. A function called conditionalStop is regularly called and it records when technical and fundamental bits are not being used anymore. Thus, for a simulation runs two data points are recorded.

Stats

This class is also defined as abstract and implements two correlation calculations.

NESFI-ASM Settings

Like the original SFI-ASM, the NESFI-ASM has a lot of parameters to be set. Most of them can be set through the GUI or through a parameter file, but some of them are only hard coded in the source code. The following section will shortly describe these program parameters and shows how to set them over a parameter file:

The parameters in the main GUI parameter window ("NESFI-ASM Settings") are intended to be logically arranged.

1. **`SFIAgent, NESFIAgent`**: SFI-Agents and NESFI-Agents are slightly different, e.g., with respect to the GA they employ. Thus, certain static parameters for these two classes can be set separately. Clicking on them opens for each class another parameter window in which certain parameters can be set.
   - 1.1. `InitialCash`: Determines the initial amount of cash with which agents are endowed with.
   - 1.2. `MinActiveRules`: The minimum number of activated rules from which the Roulette-Wheel mechanism chooses one to act upon. If less rules are active, the agent chooses as forecast parameters a fitness weighted average over all his rules. In the original SFI-ASM, MinActiveRules was effectively set to 1, which probably lead to instabilities in the ordering behavior of agents. Thus, for NESFI-Agents MinActiveRules should be bigger than 1.
   - 1.3. `MinCash`: Agents can have negative cash balances up to this specified amount.
   - 1.4. `NumRules`: The number of trading rules per agent. Due to the corrected mutation operator, the NESFI-ASM usually works with less specific rules, thus, the rule set doesn't need to be as big as in the original SFI-ASM.
   - 1.5. `ProbCrossover:` The probability with which the crossover operator is called once an agent invokes the GA on his rule set. With probability (1–ProbCrossover), the mutation operator is called.
   - 1.6. `ReplaceXXRules:` Typically, one fifth of the rule set is scheduled to be replaced by new rules which are created by the GA. If NESFI-Agents have smaller rule sets, this number should be set accordingly.
   - 1.7. `RiskAversion:` This set the level of risk aversion which is needed for the CARA utility optimization. It should be equal for both types of agents.

2. NumberOfSFIAgents, NumberOfNESFIAgents: How many agents of each types constitute the population of the model economy.
3. SelectionMethod: What method is used by the agents to choose a particular rule from the set of activated rule to act upon. SelectBest always chooses the best rule, SelectAverage uses for the forecast parameters a fitness weighted average over all rules, and RouletteWheel chooses fitter rules with a higher probability than less fitter rules.
4. FracClassifierAgents: This sets the fraction of NESFI-agents that have access to a full classifier system, especially the condition part of a trading rule. Simulations showed that in the long run NESFI-Agents completely abandon the use of technical and fundamental bits. Thus, the use of a classifier systems becomes obsolete and agents could start out without the ability to detect different states of the world. This might work just fine for the Ohrnstein-Uhlenbeck dividend process, but not for periodic dividend data such as square waves. For the latter, classifier agents will do substantially better. **Note:** *This is a fraction of the NESFI-Agents. Trading rules of SFI-Agents will always have a condition part.*
5. FracTechnicians: This is the fraction of NESFI-classifier agents that have access to the technical trading bits. Even though the agents in the original SFI-ASM could be classified as either technical or fundamental, in the NESFI-ASM they all have access to technical trading bits.
6. FracFastAgents: This is the fraction of NESFI-classifier agents that learn at a different pace than the "normal learners". While the latter invoke the GA every GaInterval periods, the "fast learners" do this every GaIntervalFastAgents periods. Obviously, the "fast learners" could also be called "slow learners" if GaInterval < GaIntervalFastAgents.
7. FirstGATime: Specifies the initial period for which no GA is invoked at all. It seems to be important that this period is bigger than θ, the parameter that determines what time window is taken into account for a rule's fitness updating.
8. NumberOfPeriods: Specifies the (maximum) number of periods that the simulation should run.
9. StopAtZeroBit: If you are only interested at what period agents have completely abandoned technical and fundamental bit usage and have that recorded, you should check that box. Thus, a simulation run can be stopped before NumberOfPeriods have passed.
10. InterestRate: Sets in the interest rate in the economy. This is the yield of the risk free bond (agents are assumed to hold their cash in bonds), and it is also the rate agents have to pay if they have negative cash balances. In order to avoid exponential growth in the economy, this is also the rate at which agents pay taxes to their capital gains.
11. Memory: For the correlation determination between price and hree-price, agents only take into account the last memory periods.
12. Hree: Puts the model into hree-mode. This serves as a benchmark. The specialist sets the price such that it is market clearing without any trading going on. Each agent holds exactly one unit of stock and the price is set such that they don't wan't to deviate rom that position.
13. Stock: The static properties of the stock can be set here:

13.1. `DividendMean:` The mean of the dividend process (provided that it is defined as an Ohrnstein-Uhlenbeck-process).

13.2. `DividendProcess`: The model allows so far for

13.2.1. mean reverting Ohrnstein-Uhlenbeck processes,

13.2.2. a constant dividend (was used once for debugging purposes),

13.2.3. a Random Walk,

13.2.4. periodic dividend data such as sinus wave, square wave, and triangular wave form. The square wave form seems to be most difficult for the classifier system to learn, but altering the hard wired key break points in the classifier system probably could improve its performance. Experimenting with periodic data and comparing classifier agents with non-classifier agents is very interesting and instructive.

13.3. `NoiseVar`: this is the variance of the white noise added to the dividend process. I suggest to decrease this number if you work with a random walk dividend process. For the Ohrnstein-Uhlenbeck process the preset value is calibrated according the description in the various articles.

13.4. `Rho`: The parameter for mean reversion for the Ohrnstein-Uhlenbeck process. If you set `Rho` at 1, it is a pure random walk, if it is zero, the dividend process is a stationary white noise process with mean around `DividendMean` and variance `NoiseVar`.

14. `TradingRule:` Sets the static parameters for the trading rules of an agents. They mainly determine specifics of the genetic algorithm and fitness updating.

14.1. `BitCost:` The amount by which the fitness of a trading reduced for each non-# bit. Thus, it not a monetary cost, yet it favors the use of more general rules. It should secure that every bit that survives in the selection process indeed contains a useful information.

14.2. `BitProb:` The probability with which bits are set when trading rules are initialized.

14.3. `GenFrac, MaxNonActive:` Once a rule hasn't been activated for more than `MaxNonActive`, it is scheduled for the generalization procedure during the next GA-invocation. Every bit in the condition part is then set to # with a probability of `GenFrac`.

14.4. `MaxSpecificity:` Determines the maximum specificity, i.e., the maximum number of bits that can be set, of a rule. It is needed for fitness updating to prevent a negative fitness and you shouldn't change it unless you alter the fundamental and technical condition word. In case of any doubt, set this parameter quite high.

14.5. `MaxDev:` Defines the maximum deviation of the realized (`price+dividend`) from its forecast.

14.6. `MinCount:` The minimum number of activated trading rules before an agents uses Roulette Wheel selection. Otherwise, for the forecast parameters a fitness weighted average over all rules in his trading set is used.

14.7. `Nhood:` "Size of the neighborhood" for short jump for the mutation of the real valued parameters.

14.8. ProbLongJump, (ProbShortJump): Probability to for a "long jump" ("short jump") of the real valued parameters in the mutation operator, i.e., outside (inside) the defined neighborhood.

14.9. ProbMutation: This defines the mutation probability for a single bit once a trading rule has been chosen for mutation with probability (1–ProbCrossover).

14.10. Subrange: Determines the initial homogeneity of trading rules of an agent. A middle point for the real valued parameters is calculated (it is the hree-value) around which, i.e. in the interval [middle point – Subrange* middle point, middle point + Subrange* middle point], the real valued parameters are initialized. Subrange= 0 gives homogenous trading rules.

14.11. Theta: Determines the size of the time windows that agents take into account when updating their rule fitness. Affects strongly the speed of accuracy adjustment. If Thetha goes to infinity, then all past information is used, thus implicitly assuming a stationary world, if Thetha = 1, the rules would be judged only on their last periods performance.

15. Showisplays: No matter what options in ObserverOptions have been activated, no visual is displayed unless this option is switched on. With this option you can even have displays when the model is run in batch mode.

16. RecordData: No matter what options in RecorderOptions have been activated, there are no data written to an output file unless this option is switched on.

17. The Repast parameter in the section below are of no interest (I just don't know how to switch them off) except for RandomSeed. By setting a specific RandomSeed you can exactly repeat the sequence of random numbers generated and thus, can have identical simulation runs.

For the options offered in the "Repast Actions" window please read the "Repast How Tos". I haven't used them. "Make Movie" and "Take Snapshot" doesn't work with the OpenSequenceGraph I use, it would work only with the agent grid which is of no particular value for this simulation.

A sample parameter file for batch runs of the NESFI-ASM is given below. If you don't understand the structure, please check out the "Repast How Tos". To give you such an example at this place just wants to point out the available options in the parameter file:

Example parameter file Asm.pf:

```
runs: 25
/*
You can have much more elaborate parameter files than this
one. Please read the "Repast How Tos" in order to get an idea
what jobs you can schedula at once. You can test, for in-
stance, whole parameter ranges and repeat each parameter set-
ting for 25 runs.
Here are two problems that I have encountered:
1. First, the : must follow immediately the set commands. This
   is obviusly required by the Repast-parameter file reader.
2. You need to have at least one set_list: command in the
   parameter file. Otherwise the batch runs don't stop after
   the specified number of runs. This is a Repast bug which
   will hopefully remedied in future versions.
```

```
*/
NumberOfSFIAgents {
   set: 0
}
NumberOfNESFIAgents {
   set: 25
}
fracClassifierAgents {
   set: 1.0
}
fracTechnicians {
   set: 1.0
}
fracFastAgents {
   set: 0.0
}
gaInterval {
   set_list: 10 25 250 1000
}
gaIntervalFastAgents {
   set: 25
}
firstGATime {
   set: 250
}
numberOfPeriods {
   set: 250
}
stopAtZeroBit {
   set_boolean: false
}
interestRate {
   set: 0.1
}
memory {
   set: 2500
}
Hree {
  set_boolean: false
}
showDisplays {
   set_boolean: false
}
recordData {
   set_boolean: true
}
/*
If you want the model to start with identical random seeds,
uncomment the next section.
*/
//RngSeed {
//   set: 1
```

```
//}
/*
The following lines set parameters that don't show up in the
main NESFI-ASM Settings window. This is because these variable
names are not contained in the string returned by getInit-
Param() if the model is run in GUI mode, even though their ap-
propriate set-accessor methods are contained in AsmModel.java.
When the model is run in batch-mode, these variable names are
returned by getInitParam() such that they can be recognized by
the parameter reader
*/
crossoverProbability {
   set: 0.3
}
bitCost {
   set: 0.01
}
maxNonActive {
   set: 2000
}
riskAversion {
   set: 0.3
}
recorderOutputFile {
   set_string: I:\TimeSeries.txt
}
recorderParamFile {
   set_string: H:\dissertation\java\nesfionestock\recorder.pf
}
```

Literature (without hyperlink)

Arthur, W.B., Holland, J., LeBaron, B., Palmer, R., Tayler, P., (1997): "*Asset pricing under endogenous expectations in an artificial stock market*", in: Arthur, W.B., Durlauf, S., Lane, D. (Eds.), *The Economy as an Evolving Complex System* II, Addison-Wesley, Reading, MA., pp. 15–44.

LeBaron, B., Arthur, B.W., Palmer, R., (1999): "*Time series properties of an artificial stock market*". in: *Journal of Economic Dynamics and Control* **23**, 1487--1516.

If you have any questions regarding the software or the working paper, feel free to contact me at ehrentreich@wiwi.uni-halle.de.