**Societies Technical Documentaion**

**This code runs an agent-based simulation of an economy that grows from the ground up.**

**Agents are self-interested utility-maximizers who extract, consume, and trade resources.**

**They also invent tools, machines, factories, and industries that reduce the effort required to extract resources. Simple technologies are gradually repurposed from their original uses into components of more complex technology.**

**Through trade and technology the economy evolves from a group of isolated, self-sustaining but resource-poor agents to a society with a high degree of specialization, interdependence and wealth. The computer simulations show that introducing trade, tools, innovation, a**

**nd diffusion of technology increases total resources extracted, productivity, degree of specialization, and interlocking complexity.**

**The extraction of each resource gradually becomes dependent on other resources,**

**to the point that introducing scarcity in one resource causes an economic collapse until new technologies are invented. At the same time these generate increasingly wide variance in wealth across agents.**

**SOCIETIES was originally written in Python (in 2010-2012) and then was converted to C++ in 2013 to significantly reduce run times of each simulation.**

# Install/Setup:

## Pull libraries:

### C++ Libraries: When using the C++ version of the code 2 external libraries also need to downloaded: libboost-filesystem-dev and libboost-dev

* + - * To do this, go to the command line and type aptitude install (Insert library name):
        + aptitude install libboost-filesystem-dev
        + aptitude install libboost-dev

### Other Libraries: Two external libraries need to be downloaded before the program can be launched.

* + - * scipy: can be downloaded from the following link <http://sourceforge.net/projects/scipy/files/> or can be found in the Societies installation package.
      * numpy: can be downloaded from the following link <http://sourceforge.net/projects/numpy/files/> or can be found in the Societies installation package.
      * both of these can be installed by typing the following into the command line.
      * apt-get install python-numpy python sci-py.
      * apt-get install python-matplotlib.

### Pull the programs from GitHub:

<https://github.com/JohnSherwood7/SOCIETIES>

## Running Societies

### The first five directions only need to be completed before building the first time:

* + - 1. Use the command line to go into src/rudeconfig-5.0.5/
      2. type: ./configure --prefix=`pwd` (note: those are backsticks around pwd)
      3. type: make
      4. type: make install
      5. then go back one directory and go into the cpp

### These are the normal directions after rudeconfig has been built

* + - * + go into the src/cpp folder
        + type: make
        + type:./societies

### If the program will not run because of the libboost libraries error message:

* + - * + type: make clean
        + type: make
        + This needs to be done because of the dynamically linked libraries building paths specific to the versions available on the computer in which the program was compiled on.

### Typical Runs

Typically, the program will be run with several additional parameters that specify the location of the input parameter file, the destination folder for the output .csv files, and the level of messages that are to be sent to the screen.

Command Line options

-h [ --help ] produce help message

-p [ --parameter ] arg use given file for global variable values (

default: ../conf/defaultValues.conf)

-z [ --agents ] arg use the given file for parameters to define agent types

(default:../conf/AgentTypes.csv)

-s [ --save ] arg write results to folder (default: no results saved)

-S [ --seed ] arg initialize random number generator to given seed so that the results

are exactly the same for each iteration (default = 6)

-t [ --title ] arg give the simulation (batch) a title (default: sim\_default)

-n [ --norun ] load and print the current config values, then exit

-a [ --agent ] arg remove an agent mid-run

1st arg: which agent to remove

2nd arg: which day to remove the agent

-r [ --resource ] arg remove a resource mid-run (

first arg: which resource to remove

second arg: which day to remove the resource

third arg: 1 remove holdings, 0 don't remove)

-v [ --verbose ] arg set level of debugging output from 0 to 3. 0 = nothing; 3 = everything

Config file options

--START\_DAY arg the day to start on when restarting a simulation

--DAY\_LENGTH arg the length of a day

--NUM\_DAYS arg number of days in the simulation

--NUM\_RESOURCES arg number of resources in the simulation

--RES\_TRADE\_ROUNDS arg number of resource trading rounds

--RES\_TRADE\_ATTEMPTS arg number of resource trading attempts

--DEVICE\_TRADE\_ROUNDS arg number of device trading rounds

--DEVICE\_TRADE\_ATTEMPTS arg number of device trading attempts

--DEVICE\_TRADE\_MEMORY\_LENGTH arg

--DEVICE\_PRODUCTION\_MEMORY\_LENGTH arg

--MIN\_DEVICE\_FOR\_DEV\_DEVICE\_CONSIDERATION arg

--MIN\_RES\_HELD\_FOR\_DEVICE\_CONSIDERATION arg

--DAILY\_EXP\_PENALTY arg

--PRODUCTION\_EPSILON arg

--RESOURCES\_IN\_TOOL arg number of resources needed to make a tool

--INVENTOR\_DEVICE\_EXPERIENCE arg

--NUM\_DEVICE\_COMPONENTS arg

--DAILY\_DEVICE\_DECAY arg

--MIN\_HELD\_DEVICE\_EXPERIENCE arg

--MIN\_RES\_UTIL arg

--TRADE\_EPSILON arg

--TOOL\_PROBABILITY\_FACTOR arg

--DEVICE\_PROBABILITY\_FACTOR arg

--TOOL\_FACTOR arg

--TOOL\_LIFETIME arg

--MACHINE\_FACTOR arg

--MACHINE\_LIFETIME arg

--FACTORY\_FACTOR arg

--FACTORY\_LIFETIME arg

--INDUSTRY\_FACTOR arg

--INDUSTRY\_LIFETIME arg

--DEV\_MACHINE\_FACTOR arg

--DEV\_MACHINE\_LIFETIME arg

--DEV\_FACTORY\_FACTOR arg

--DEV\_FACTORY\_LIFETIME arg

--DAYS\_OF\_DEVICE\_TO\_HOLD arg

--TRADE\_EXISTS arg

--DEVICES\_EXIST arg

--TOOLS\_ONLY arg

**Description of the Program**

|  |  |
| --- | --- |
| **Python Snapshot May 24th 2012**: | **C++ Version** |
| Agent.py   * Purpose: * Classes:   + Agent: | * Agent.cpp * Purpose: This class manages all the attributes and daily actions of the agent. Some of these actions would include:   + the initialization of agent attributes   + the process of working (including decision calculations)   + device invention and building   + daily update from penalties, decay, trade acquisitions, and resetting daily values back to 0. * Includes:   + - globals.h     - agent.h     - device.h * agent.h   + Purpose:   + Includes:     - properties.h   + Classes:     - Agent |
| config.py   * Purpose: This is one of the initializing classes for the program, and sets the global variables. The operation of this class largely depends on what is triggered through the options.py. This will read in config files and set corresponding variables for agents, resources, save files, graphing component, program output, and etc. This also prepares the stats and has many print options. * Classes: none | * globals.cpp   + Purpose:   + Includes:     - globals.h     - agent.h     - resource.h     - marketplace.h     - ran.h * globals.h   + Purpose:   + Includes:     - resource.h   + Classes:     - Globals |
| device.py   * Purpose: There are two:   + 1. Agents evaluate the time constraints and the potential gain for having a certain device.   + 2. Set the attributes for each type of device. * Classes:   + Device:     - Purpose:       * Determine who owns a device       * Determine the time and resource constraints of that device   + DevDevice: (Device that makes devices)     - Purpose: Gives agents a function that allows them to consider whether would it be worth having a device that speeds up the making of other devices   + DevMachine (Device that makes machines)     - Purpose: Set the properties of this device   + DevFactory (Device that makes factories)     - Purpose: Set the properties of this device   + Tool:     - Purpose:       * Set the properties of this device       * allow agents to calculate the cost and whether would it be worth having this.   + Machine:     - Purpose:       * Set the properties of this device       * allow agents to calculate the cost and would it be worth having this.   + Factory:     - Purpose:       * Set the properties of this device       * allow agents to calculate the cost and would it be worth having this.   + Industry:     - Purpose:       * Set the properties of this device       * allow agents to calculate the cost and would it be worth having this. | * device.cpp   + Purpose:   + Includes:     - device.h     - agent.h * device.h   + Purpose:   + Includes:     - globals.h   + Classes:     - Device     - DevDevice     - DevMachine     - DevFactory     - Tool     - Machine     - Factory     - Industry |
| device.pyx   * Purpose: Same as device. The only difference is a small inclusion at line 276 |  |
| **devmarketplace.py**   * + Purpose:     - allows agents to break apart and have attempts to trade if devices are invented.     - When trades do occur, this updates memory and stats.   + Classes:     - DeviceMarketplace       * Purpose:         + allows agents to see if they have any favorite device traders (Starting with the most complex devices to the most simple)         + if not, it allows them to see if there is anyone else they can trade with.         + any agents who have not yet traded, are paired off with a random partner.     - PairDeviceMarketMemory       * Purpose: This facilitates the trading of devices. This keeps trade of the surplus they have received from previous trades along with updating the stats, making offers, counter offers, and trading. | device .cpp  device.h |
| graph.py   * Purpose: Use the data generated from the stats classes that have been kicked into csv files and upload those back to make graphs * Classes: none |  |
| marketplace.py   * Purpose: * Classes:   + Marketplace     - Purpose: Agents are put into pairs, make offers, attempt to trade, make counter offers, make the trade, and update the stats   + PairMarketMemory     - Purpose: This sets who will be the arbitrary agentA and agentB during a pair’s trading rounds. | * marketplace.cpp   + Purpose:   + Includes:     - marketplace.h * marketplace.h   + Purpose:   + Includes:     - globals.h   + Classes:     - ResourceMarketplace     - ResourcePair |
| myPow.pyx   * Purpose: Calculate the resource marginal utility curves using cython. Using cython is intended to give the program a speed up. * Classes: none |  |
| options.py   * Purpose: The options module takes user input from the command line to set the values of certain global variables. * Classes: none | * options.cpp   + Purpose:   + Includes:     - globals.h * options.h   + Purpose:   + Includes: none   + Classes: none |
| properties.py   * Purpose: This sets the properties of a given resource and device. * Classes:   + ResProperties     - Purpose: An agent will create an object of this class for a single resource. The object contains all the properties for that resource.   + DevProperties     - Purpose: An agent will create an object of this class for each device and for each resource of that device. It contains all the properties of that device's resource. | * properties.cpp   + Purpose:   + Includes:     - properties.h     - globals.h * properties.h   + Purpose:   + Includes: none   + Classes: none |
| resource.py   * Purpose: This sets the basic values of resources. * Classes:   + Resource | * resource.cpp   + Purpose:   + Includes:     - resources.h     - globals.h * resource.h   + Purpose:   + Includes:   + Classes: |
| setup.py   * Purpose: This is the setup for the mypow module, which is a cythonized class intended to speed up the marginal utility calculations * Classes: none |  |
| soc\_gui.py   * Purpose: This is the graphical interface that can be accessed through the command line. * Classes:   + App     - Purpose: |  |
| societies.py   * Purpose: This is the module that contains the main function of the simulation * Classes: none | societies  main.cpp   * Purpose: * Includes:   + globals.h   + utils.h   + options.h |
| statstracker.py   * Purpose: The statstracker module contains definitions for the TradeStats class, the DayExchangeStats class, the ProductionStats class, and the OtherStats class. Each of these classes acts as a pseudo-database to store simulation statistics. * Classes:   + TradeStats     - Purpose: First this sets up the necessary lists to stand as a database for the trading transactions. Then everyday each agents trades should be added and information of which resources were bought and sold should be kept.   + DayExchangeStats     - Purpose: This should keep track of the number of resources exchanged, and for each device keep track of how many were bought and sold.   + ProductionStats     - Purpose: This keeps track of stats for each agent, each device, and each resource. Also this calculates stats on a global level. Example: of all (agent/device/resources) how many/much came from agent x device y or resource z.   + OtherStats     - Purpose: This keeps track of active agents, sum of resources or utility, and number of devices invented (for each type). | statstracker.cpp   * Purpose: * Includes:   + statstracker.h   + Globals.h   + Agent.h   + Device.h   statstracker.h   * Purpose: * Includes:   + globals.h * Classes:   + ProductionStats   + OtherStats   + TradeStats   + DeyExchangeStats |
| utils.py   * Purpose: The utils module includes commonly-used functions for use during simulations.   + This direct the agents’ activities   + This handles the clean up at the end of each day   + Updates daily stats   + Prints end day stats   + Checks for and removes of agents/resources   + Handels mid-run save   + Ends the simulation on the last day   + Saves results to a collection of files   + Timer function   + Loads mid-run data * Classes: none | * utils.cpp   + Purpose:   + Includes:     - globals.h     - utils.h     - agent.h     - device.h     - marketplace.h * utils.h   + Purpose:   + Includes:     - globals.h   + Classes:     - globals.h |
|  | **Makefile** |
|  | * ran.h   + Purpose:   + Includes:   + Classes:     - Ran01     - RanBinomial     - IRan     - Ran     - Nran |