ChatGPT

```
/*
 * Working_Legacy_Code_Reference.c
 *
 * Annotated reference of the legacy Emergent Money simulation code
(originally adamsmithfeb17B.c from Risto Linturi's 2010 report [1]).
 * This simulation models a networked barter economy with emergent money-like
behavior among agents [2].
 * Each agent produces goods to meet its needs, trades with a small network
of friends if beneficial (barter exchanges), and adjusts internal prices and
trust based on outcomes [3].
 * The code is written in C with heavy use of integer arithmetic scaled by 10
(INTMULTIPLIER) to avoid floating-point operations [4]. Many values are "ten-
folded" for computation.
 * Agents' state is stored in global arrays/structs for performance, and each
simulation cycle (period) updates all agents (consumption, production,
exchange, pricing, etc.) [5].
 * Global market statistics (MARKETSTRUCTURE) track aggregate production,
prices, and adjust "elastic need" each period to simulate price elasticity of
demand [6] [7].
 *
 * Each function and major logic block below is documented to explain its
purpose, how it maps to simulation behavior (barter, production, consumption,
price updates, etc.),
 * and how it connects to concepts from the Emergent Money report (e.g.,
social transparency/trust, price elasticity, role of consumer/producer/
retailer).
 */

#include <stdheaders.h>
#include <limits.h>

/* --- Simulation Parameters and Constants --- */
#define FILENAME "adamsmith.txt"    // Output file for logging results each
period
#define INTMULTIPLIER 10           // Scale factor for integer math (values
scaled by 10 to simulate one decimal) [4]
#define POPULATION 300             // Number of agents in the simulation
(population size)
#define MAXGIFTED 60               // Max number of gifted individuals per
skill (talented producers)
#define MAXGROUP 10                // Max number of social connections
(friends) per agent
#define SKILLS 31                  // Number of goods/skills (utility types)
in the economy

#define INITIALPRICE 1.0f          // Initial sales/purchase price for all
goods
```

```
#define INITIALEFFICIENCY 1.0f     // Initial production efficiency for all skills (1 unit time per unit produced)
#define GIFTEDINITIALEFFICIENCY 2.0f  // Initial efficiency for gifted individuals (faster production)
#define GIFTEDEFFICIENCYMINIMUM 1.5f   // Minimum efficiency retained for gifted skill (they don't fall below 1.5x normal)

#define INITIALSOCIALTRANSPARENCY 0.70f // Initial transparency/trust in new friendships (70%)
#define MAXSURPLUSFACTOR 2        // Max surplus factor for stock (stock limit = MAXSURPLUSFACTOR * need)
#define MAXPERIODS 2000           // Max number of cycles (periods) to simulate (for testing)

#define MAXUSEFULINDEX 10000000   // A large number used as initial "least useful" index (for friend replacement logic)

#define DISCONT 0.8f              // Discount factor for past transactions (1-DISCONT=0.2 is the fraction remembered each period) - older data fades
#define HISTORY 4                 // History length for initial statistics (HISTORY=4 means initial recent production = 4 periods of need) [8]

#define SPOILEDSURPLUSDELAY (2 *
HISTORY) // Not used explicitly in this code (placeholder for spoilage delay)

#define PRICEDEMANDELASTICITY 2   // Price-demand elasticity mode: 0 = none, 1 = linear, 2 = quadratic response of demand to price [7]
#define BASICROUNDELASTIC 1       // If 1, even the basic round uses elastic needs; if 0, only surplus rounds use elastic needs (affects need calculation)
#define SPOILSURPLUSEXCESS 0.10f  // Fraction of excess surplus that spoils each period (10% of over-stock is spoiled) [9]
#define STOCKSPOILTRESHOLD 2.0f   // Threshold (fraction of stock limit) beyond which surplus starts spoiling (here 2.0 = 200% of stock limit) [10]
#define PRICEREDUCTION 0.95f      // Factor to reduce prices (5% reduction) when lowering value thresholds
#define PRICEHIKE 1.05f           // Factor for small price increase (5% hike) when raising value thresholds
#define PRICELEAP 1.3f            // Factor for large price jump (30% increase) for drastic adjustments

// The following are defined for readability/documentation purposes:
#define NETWORK
1    // Exchange type code for network barter (always used in this model)
//#define MARKET 2   // (Not in use) code for a hypothetical centralized market mechanism
#define NOMARKET 1   // Flag to disable market mechanism (always true, only network exchanges are considered)

#define CONSUMER 10  // Role ID for Consumer (base role). Roles also act as weight multipliers in exchange value calculation [11].
#define RETAILER 11  // Role ID for Retailer (mid-level trader). A higher
```

```
role value favors that agent in exchange priority [11].
#define PRODUCER 12  // Role ID for Producer (specialist producer). Producer
role (highest value) is weighted most in exchange index [11].

#define SURPLUSDEALS 1   // Deal type code indicating an exchange aimed at
acquiring surplus (as opposed to satisfying immediate need)
#define CONSUMPTION 2    // Deal type code indicating exchange for
consumption needs

#define REGULARROUND 0   // Round type: Regular consumption/production round
#define SURPLUSROUND
1   // Round type: Surplus production round (after basic needs met)
#define LEISUREROUND 2   // Round type: Leisure production round (using
leftover time)

#define TESTINDIVIDUAL (market.testindividual)  // Index of the agent being
tracked in detail (for debugging outputs)
#define DEFAULTTESTINDIVIDUAL
10               // Default test individual ID (agent 10 by default)
#define TESTSKILL 1   // (Several TESTSKILL# defines are for debugging
specific skills of the test individual)
#define TESTSKILL2 2
#define TESTSKILL3 3
#define TESTSKILL4 4
#define TESTSKILL5 5
#define TESTSKILL6 6
#define TESTSKILL7 7
#define TESTSKILL8 8
#define TESTSKILL9 9
#define TESTSKILL10 10

char mode = '1';  // Mode for simulation run control: '0' for no pause, '1'
or others to pause each cycle for user input (used with getchar below)

/* --- Data Structures --- */

/**
 * EVENTS: Stores dyadic (pair-wise) transaction data between an agent and
one friend.
 * Each agent has an array of EVENTS (size MAXGROUP) for their social
connections.
 * - relationid: the actual agent ID of the friend (0 if slot is empty).
 * - sold[p]: how many units of product p this agent has sold to that friend
(cumulative).
 * - purchased[p]: how many units of product p this agent has purchased from
that friend.
 * - transparency[p]: trust/transparency for product p transactions with that
friend (0-1 scale, higher = more trust). Updated as transactions occur [12] [13].
 * - transactions: total number of exchanges (of any goods) with that friend
(used to measure relationship strength).
 */
```

```
typedef struct {
    short relationid;
    short sold[SKILLS];
    short purchased[SKILLS];
    float transparency[SKILLS];
    float transactions;
} EVENTS;

/**
 * PERSON: Represents an agent in the simulation, with personal state
variables.
 * Contains arrays for each good (of size SKILLS) and other per-agent
parameters:
 * - relationship[MAXGROUP]: social network connections (up to 9 friends,
index 0 unused). Each is an EVENTS struct documenting trades and trust with
one friend.
 * - efficiency[p]: production efficiency for product p (units produced per
time unit, effectively). Higher means faster production. Gifted individuals
start with >1 efficiency [14].
 * - gifted[p]: boolean (int) flag if agent has a natural talent for product
p (gains higher initial efficiency).
 * - role[p]: current role regarding product p (Consumer=10, Retailer=11,
Producer=12). This is updated each period based on agent's trade behavior [15]
[16] and used to influence exchange preferences.
 * - need[p]: amount of product p the agent needs this period (resets each
cycle, initially equal to basicneed).
 * - surplus[p]: current stock of product p that agent has (can be used to
consume or trade). Surplus is what's produced or obtained beyond immediate
need.
 * - stocklimit[p]: maximum stock agent is willing/able to hold for product
p. It is dynamically adjusted each period based on needs and recent sales [17]
[18].
 * - previousstocklimit[p]: last period's stocklimit (for smoothing changes
in stocklimit).
 * - totalsurplus: total number of all items in stock after agent finishes a
period (for statistics).
 * - salesprice[p]: agent's minimum acceptable price to sell product p (their
value threshold for giving p). Adjusted each period based on experience [19]
[20].
 * - purchaseprice[p]: agent's maximum willing price to buy product p (value
threshold for receiving p). Adjusted each period based on shortages or
surplus [21].
 * - purchasetimes[p]: number of purchase transactions of product p in the
current period (counter reset each cycle).
 * - sumperiodpurchasevalue[p]: cumulative value of product p purchases this
period (used to compute average purchase price actually paid).
 * - salestimes[p]: number of sales transactions of product p in current
period.
 * - sumperiodsalesvalue[p]: cumulative value of product p sales this period.
 * - recentlyproduced[p]: amount of product p produced in recent history
(rolling, decayed by DISCONT each period). Used for efficiency learning and
```

```
trust calculations.
 * - producedthisperiod[p]: amount of product p produced in the current
period.
 * - producedxperiod[p]: amount of product p produced last period (carried
over for comparisons).
 * - recentlypurchased[p]: amount of p purchased in recent history (decayed
each period).
 * - purchasedthisperiod[p]: amount of p purchased in the current period.
 * - purchasedxperiod[p]: amount of p purchased last period.
 * - recentlysold[p]: amount of p sold in recent history (decayed each
period).
 * - soldthisperiod[p]: amount of p sold in the current period.
 * - soldxperiod[p]: amount of p sold last period.
 * - spoils[p]: amount of product p spoiled (wasted) this cycle due to excess
surplus beyond stock limits [22].
 * - periodicspoils: total spoils across all goods for this agent in the
current period.
 * - periodremaining: remaining time units for this agent in the period
(starts at periodlength and counts down as they produce).
 * - periodremainingdebt: if negative, indicates overtime (unmet needs due to
insufficient time) carried to next period as debt (reduces next period's
available time).
 * - periodfailure: flag if agent ran out of time this period without meeting
needs (used to force need reduction).
 * - timeout: counter for how many times the agent had periodremaining < 0
(indicating they couldn't meet needs in time, which should be rare).
 * - needslevel: factor for agent's needs relative to basicneed (>=1.0). This
increases if agent consistently has surplus time/resources, and decreases if
agent fails to meet needs [23] [24].
 * - recentneedsincrement: tracks recent relative increase in needs (used in
adjusting needslevel gradually).
 * - entrepreneur: (reserved for future use, not utilized in this code).
 */
typedef struct {
    EVENTS relationship[MAXGROUP];
    double efficiency[SKILLS];
    int gifted[SKILLS];
    int role[SKILLS];
    long long need[SKILLS];
    long long surplus[SKILLS];
    long long stocklimit[SKILLS];
    long long previousstocklimit[SKILLS];
    long long totalsurplus;
    float salesprice[SKILLS];
    float purchaseprice[SKILLS];
    int purchasetimes[SKILLS];
    float sumperiodpurchasevalue[SKILLS];
    int salestimes[SKILLS];
    float sumperiodsalesvalue[SKILLS];
    long long recentlyproduced[SKILLS];
    long long producedthisperiod[SKILLS];
```

```
    long long producedxperiod[SKILLS];
    long long recentlypurchased[SKILLS];
    long long purchasedthisperiod[SKILLS];
    long long purchasedxperiod[SKILLS];
    long long recentlysold[SKILLS];
    long long soldthisperiod[SKILLS];
    long long soldxperiod[SKILLS];
    long long spoils[SKILLS];
    long long periodicspoils;
    long long periodremaining;
    long long periodremainingdebt;
    int periodfailure;
    int timeout;
    // float needsincrement; // (unused in this version; would track
incremental needs increase in extra rounds)
    float needslevel;
    float recentneedsincrement;
    int entrepreneur;
} PERSON;

PERSON individual[POPULATION];  // Array of all agents (indexed
1..POPULATION-1 since code often starts at 1)

/**
 * EXCHANGE: Temporary structure to evaluate a single exchange opportunity.
 * Used to store the best exchange found for an agent's need.
 * - index: exchange value index (positive if trade is beneficial for both
parties) 25 11 .
 * - individualid: the actual ID of the other agent involved in this
potential exchange (the friend offering the needed good).
 * - friendid: index in the current agent's relationship[] array for that
friend.
 * - surplusproductid: the ID of the product that the current agent would
give (surplus good) in exchange.
 * - switchaverage: the negotiated exchange rate (price ratio) between the
two goods for this trade, based on both agents' prices and trust 26 .
 * - exchangetype: NETWORK (1) for friend network exchanges (no other type
used in this simulation, NOMARKET is always on).
 */
typedef struct {
    float index;
    int   individualid;
    int   friendid;
    int   surplusproductid;
    float switchaverage;
    int   exchangetype;
} EXCHANGE;

EXCHANGE bestexchange;                      // Holds the best exchange
option found for current need
float fexchangeindex[MAXGROUP][SKILLS];     // Matrix of exchange indices
```

```
for all friend connections and possible gifts (populated in
getfriendexchangeindexes)

/**
 * FRIENDCANDIDATE: Used internally to choose a friend to drop when making a
new friend.
 * - index: the "usefulness" metric (here simply the number of transactions)
of a friend.
 * - individualid: (not used here for actual friend selection; would be ID of
candidate friend in some contexts).
 * - friendid: the index (1..MAXGROUP-1) of the friend slot in our
relationship array.
 */
typedef struct {
    int index;
    int individualid;
    int friendid;
} FRIENDCANDIDATE;

/**
 * MARKETSTRUCTURE: Global market aggregates and constants.
 * This struct holds economy-wide data and stats updated each cycle:
 * - period: current cycle number (period count).
 * - periodlength: total time slots available to each agent per period
(initially sum of all basic needs) [27] [28].
 * - leisuretime: threshold of unused time that triggers needs increase (set
as periodlength/SKILLS, meaning if an agent has on average >=1 time unit per
need free, they have "leisure") [27] [29].
 * - basicneed[p]: baseline need (in "units" scaled by INTMULTIPLIER) for
product p. Set to p*p*INTMULTIPLIER (so needs grow quadratically with p
index) [28].
 * - elasticneed[p]: current elastic need for product p (demand adjusted by
price elasticity). Starts equal to basicneed and is recalibrated each period
based on prices [7].
 * - previouselasticneed[p]: last period's elasticneed (for limiting rate of
change).
 * - surplus[p]: (unused in code; would hold total surplus of product p
across all agents).
 * - averageprice[p]: average production cost (price) of product p in the
market (used as a global price indicator) [30].
 * - maxefficiency[p]: max efficiency among all agents for product p (to
identify who produces cheapest).
 * - purchasetimes[p]: total number of purchase transactions of p by all
agents this period (for stats).
 * - salestimes[p]: total number of sales transactions of p by all agents
this period.
 * - sumperiodicpurchasevalue[p]: total value spent by all agents on p this
period (for stats).
 * - sumperiodsalesvalue[p]: total value earned by all agents from p sales
this period.
 * - numberofrecentlyproduced[p]: total quantity of p produced in recent
```

```
history (decayed each period by DISCONT).
 * - numberoftotalrecentproduction: total production of all goods in recent
history (for averaging).
 * - producedthisperiod[p]: total quantity of p produced this period (reset
each period).
 * - periodictcecost[p]: total "transaction cost in time" incurred for
exchanges of p this period (time lost due to less than 100% transparency in
trades).
 * - periodicspoils[p]: total amount of p spoiled across all agents this
period.
 * - costoftceintime[p]: cost of transaction cost in time converted to
standard value (cost in time * price).
 * - costofspoilsintime[p]: cost of spoiled goods in time*price.
 * - totalcostoftceintime: sum of costoftceintime for all goods (overall lost
value due to transaction inefficiencies).
 * - totalcostofspoilsintime: sum of costofspoilsintime for all goods
(overall lost value due to spoilage).
 * - priceaverage: average price of all goods (weighted by production volume)
for this period [31].
 * - totalmisurplus: total market surplus from previous period (used to
measure change in stored goods).
 * - numberofconsumers[p], numberofretailers[p], numberofproducers[p]: count
of agents whose role for product p is consumer/retailer/producer at end of
period (for stats).
 * - testindividual: ID of the agent currently designated as TESTINDIVIDUAL
(for printing detailed stats).
 * - loosers: count of agents who fell far behind (periodremainingdebt < -
periodlength) this period (a measure of "losers" who couldn't meet needs) [32]
[33].
 */
typedef struct {
    long long period;
    long long periodlength;
    long long leisuretime;
    long long basicneed[SKILLS];
    double elasticneed[SKILLS];
    double previouselasticneed[SKILLS];
    long long surplus[SKILLS];
    double averageprice[SKILLS];
    double maxefficiency[SKILLS];
    int purchasetimes[SKILLS];
    int salestimes[SKILLS];
    double sumperiodicpurchasevalue[SKILLS];
    double sumperiodsalesvalue[SKILLS];
    long long numberofrecentlyproduced[SKILLS];
    long long numberoftotalrecentproduction;
    long long producedthisperiod[SKILLS];
    long long periodictcecost[SKILLS];
    long long periodicspoils[SKILLS];
    long long costoftceintime[SKILLS];
    long long costofspoilsintime[SKILLS];
```

```
    long long totalcostoftceintime;
    long long totalcostofspoilsintime;
    double priceaverage;
    // long long totalsurplus; // (unused placeholder)
    long long totalmisurplus;
    int numberofconsumers[SKILLS];
    int numberofretailers[SKILLS];
    int numberofproducers[SKILLS];
    int testindividual;
    int loosers;
} MARKETSTRUCTURE;

MARKETSTRUCTURE market;

/* Fast inverse square-root (utility function for efficiency updates).
 * This is the famous Quake III algorithm to compute 1/sqrt(x) quickly [27].
 * Used here to update efficiency: InvSqrt is called when recalculating an
agent's efficiency based on production history (learning effect).
 */
float InvSqrt(float x) {
    float xhalf = 0.5f * x;
    int i = *(int*)&x;
    i = 0x5f3759df - (i >> 1);    // Magic constant and bit hack for initial
guess
    x = *(float*)&i;
    x = x * (1.5f - xhalf * x * x); // One iteration of Newton's method for
refinement
    return x;
}

/* initmarket: Initialize global market parameters and period constants.
 * - Sets period = 1.
 * - Calculates periodlength as sum of basic needs of all goods (which equals
total time units each agent gets per cycle).
 * - Sets basicneed[i] = i^2 * INTMULTIPLIER (need grows with square of
product index) [28].
 * - Initializes elasticneed equal to basicneed (will be adjusted after first
period based on prices).
 * - Sets averageprice[i] = 1.0 (initial average production cost for all
goods assumed 1).
 * - Initializes counters and sets initial numberofrecentlyproduced[i] to
basicneed[i] * (HISTORY * POPULATION) to simulate a production history for
statistics [34].
 * - leisuretime = periodlength / SKILLS (average time per need) for use in
needs increase calculations.
 */
void initmarket(void) {
    market.period = 1;
    market.periodlength = 0;
    for (int i = 1; i < SKILLS; i++) {
        market.basicneed[i] = (i * i * INTMULTIPLIER);   // Define baseline
```

```
need for utility i [28]
        market.elasticneed[i] = market.basicneed[i];
        market.previouselasticneed[i] = market.basicneed[i];
        market.periodlength += market.basicneed[i];      // total time units
per agent per period (sum of all needs)
        market.averageprice[i] = 1.0f;
        market.purchasetimes[i] = 0;
        market.salestimes[i] = 0;
        market.sumperiodicpurchasevalue[i] = 0;
        market.sumperiodsalesvalue[i] = 0;
        market.numberofrecentlyproduced[i] = market.basicneed[i] * (HISTORY
* POPULATION);
        market.producedthisperiod[i] = 0;
        market.periodictcecost[i] = 0;
        market.numberofconsumers[i] = 0;
        market.numberofretailers[i] = 0;
        market.numberofproducers[i] = 0;
        market.testindividual = DEFAULTTESTINDIVIDUAL;
    }
    market.leisuretime = market.periodlength / SKILLS;
}

/* initpopulation: Initialize all agents' state at the start.
 * - periodremaining set to full periodlength for each agent (full time
budget).
 * - needslevel = 1.0 (everyone starts needing 100% of basic needs).
 * - recentneedsincrement = 1.0 (for initial calculation of needs changes,
effectively no change).
 * - timeout = 0 (no one has failed a period yet).
 * For each good j for each agent i:
 *   - need[j] = market.basicneed[j] (start each period needing the basic
amount).
 *   - purchaseprice[j] = salesprice[j] = INITIALPRICE (everyone initially
values goods equally at 1.0).
 *   - efficiency[j] = INITIALEFFICIENCY (1.0 for all skills initially; will
be raised to 2.0 later if gifted).
 *   - surplus[j] = market.basicneed[j] (initial surplus stock equal to one
period of need, so no immediate shortages) [35].
 *   - stocklimit[j] = MAXSURPLUSFACTOR * basicneed[j] (e.g. 2 * basic need,
so they can hold at most 2 periods of that good) [36].
 *   - previousstocklimit[j] initialized same as stocklimit.
 *   - gifted[j] = 0 (talents assigned in selectgiftedpopulation).
 *   - role[j] = CONSUMER (everyone starts as consumer for each good).
 *   - recentlyproduced[j] = HISTORY * basicneed[j] (pretend each agent
produced their own need for HISTORY periods, so no trade initially) [37].
 *   - producedthisperiod[j] = basicneed[j] (assume they produce their need
in the first period as a starting point).
 *   - recentlysold[j] = 0 (no exchange in history at start).
 *   - soldthisperiod[j] = 0.
 *   - recentlypurchased[j] = 0.
 *   - purchasedthisperiod[j] = 0.
```

```
 *    - purchasetimes[j] = salestimes[j] = 0.
 *    - sumperiodpurchasevalue[j] = sumperiodsalesvalue[j] = 1.0 (initialize
private values to ungifted production cost) [38].
 *    - periodfailure = 0 (no failure yet).
 * Relationship arrays: mark all friend slots as empty (relationid=0) and set
initial transparency to INITIALSOCIALTRANSPARENCY (0.7) for all goods [39].
 */
void initpopulation(void) {
    printf("initializing population");
    for (int i = 1; i < POPULATION; i++) {
        individual[i].periodremaining = market.periodlength;
        individual[i].periodremainingdebt = 0;
        individual[i].needslevel = 1.0f;
        // individual[i].needsincrement = 0; // (unused, concept of extra
needs if time left)
        individual[i].recentneedsincrement = 1.0f;
        individual[i].timeout = 0;
        for (int j = 1; j < SKILLS; j++) {
            individual[i].need[j] = market.basicneed[j];
            individual[i].purchaseprice[j] = INITIALPRICE;
            individual[i].salesprice[j] = INITIALPRICE;
            individual[i].efficiency[j] = INITIALEFFICIENCY;
            individual[i].surplus[j] = market.basicneed[j];  // Start with
one period of each good in surplus to prevent initial shortages [40]
            individual[i].stocklimit[j] = MAXSURPLUSFACTOR *
market.basicneed[j];
            individual[i].previousstocklimit[j] = MAXSURPLUSFACTOR *
market.basicneed[j];
            individual[i].gifted[j] = 0;
            individual[i].role[j] = CONSUMER;
            individual[i].recentlyproduced[j] = HISTORY *
market.basicneed[j];  // assume self-sufficient history [37]
            individual[i].producedthisperiod[j] = market.basicneed[j];
            individual[i].recentlysold[j] = 0;
            individual[i].soldthisperiod[j] = 0;
            individual[i].recentlypurchased[j] = 0;
            individual[i].purchasedthisperiod[j] = 0;
            individual[i].purchasetimes[j] = 0;
            individual[i].salestimes[j] = 0;
            individual[i].sumperiodpurchasevalue[j] = 1.0f; // initialize
private value (cost) [38]
            individual[i].sumperiodsalesvalue[j] = 1.0f;
            individual[i].periodfailure = 0;
        }
        // Initialize social relationships
        for (int k = 1; k < MAXGROUP; k++) {
            individual[i].relationship[k].transactions = 0;
            individual[i].relationship[k].relationid = 0;
            for (int l = 1; l < SKILLS; l++) {
                individual[i].relationship[k].purchased[l] = 0;
                individual[i].relationship[k].sold[l] = 0;
```

```
                individual[i].relationship[k].transparency[l] =
INITIALSOCIALTRANSPARENCY;
            }
        }
    }
}

/* randomindividual: Utility to pick a random agent index (between 1 and
POPULATION-1).
 * If rand() returns a value >= POPULATION, it subtracts POPULATION until in
range (this creates a slight bias but is simple).
 * Used in selectgiftedpopulation and makenewfriends.
 */
int randomindividual(void) {
    int randindividual = rand();
    while (randindividual > POPULATION - 1) {
        randindividual -= POPULATION;
    }
    return randindividual;
}

/* selectgiftedpopulation: Randomly assign "gifted" status for each skill to
some agents.
 * For each skill (1..SKILLS-1), up to MAXGIFTED agents are selected at
random to be gifted in that skill:
 * - Their efficiency for that skill is set to GIFTEDINITIALEFFICIENCY (e.g.,
2.0, meaning they produce twice as fast).
 * - Their gifted[skill] flag is set TRUE.
 * This creates agents with higher initial productivity in certain goods,
simulating innate talent.
 */
void selectgiftedpopulation(void) {
    for (int skill = 1; skill < SKILLS; skill++) {
        for (int j = 1; j < MAXGIFTED; j++) {
            int randindividual = randomindividual();
            individual[randindividual].efficiency[skill] =
GIFTEDINITIALEFFICIENCY;
            individual[randindividual].gifted[skill] = TRUE;
        }
    }
}

/* getmefriendindex(me, doiknowyou): Find my friend slot index for a given
individual.
 * This converts an agent ID (doiknowyou) to the index (1..MAXGROUP-1) in
agent `me`'s relationship array where that friend is stored.
 * Returns 0 if not found (meaning agent `me` does not currently know
individual `doiknowyou`).
 */
int getmefriendindex(int me, int doiknowyou) {
    int myfriend = MAXGROUP;
```

```
    while (--myfriend) {
        if (individual[me].relationship[myfriend].relationid == doiknowyou) {
            break;
        }
    }
    return myfriend; // returns index or 0 if loop fell through (no matching
relationid)
}

/* getleastusefulfriendid(me): Identify the least "useful" friend slot for
agent `me` to potentially replace with a new friend.
 * If there is any free friend slot (relationid == 0), return that slot index
immediately.
 * Otherwise, find the friend with the lowest transactions count (i.e., least
interaction) and return that friend slot index.
 * Rationale: If an agent needs to drop a friend to add a new one, they drop
the one with minimal trading activity (usefulness).
 */
int getleastusefulfriendid(int me) {
    FRIENDCANDIDATE leastusefulfriend;
    int currentindex = MAXUSEFULINDEX;  // start with a very high index
(since lower is more useful here, as we measure inactivity)
    leastusefulfriend.index = currentindex;
    for (int myfriend = 1; myfriend < MAXGROUP; myfriend++) {
        if (individual[me].relationship[myfriend].relationid == FALSE) {
            // Found an empty friend slot
            leastusefulfriend.friendid = myfriend;
            break;
        } else {
            currentindex =
individual[me].relationship[myfriend].transactions; // use total transactions
as usefulness metric
            if (currentindex < leastusefulfriend.index) {
                leastusefulfriend.index = currentindex;
                leastusefulfriend.friendid = myfriend;
            }
        }
    }
    return leastusefulfriend.friendid;
}

/* makenewfriends(me, suggestedid): Establish a new friendship for agent
`me`.
 * This is called either when:
 *   (a) `me` successfully trades with someone not currently in their network
(suggestedid = that other agent's ID), or
 *   (b) `me` pro-actively seeks a new friend (suggestedid = FALSE, meaning
no specific suggestion).
 * The function picks a friend slot to use (the least useful or an empty one)
and assigns the new friend.
 * If suggestedid == FALSE (0), it randomly picks an agent not already a
```

```
friend (and not self) to be the new friend.
 * It avoids infinite loops by a safeguard (toomanyloops).
 * Once a new friend ID is determined:
 *   - Replaces the friend in the least useful slot (possibly dropping an old
friend).
 *   - Sets relationid to the new friend's ID.
 *   - Initializes the new relationship: transactions = 2 (a small positive
start to indicate a connection), sold/purchased = 0, transparency =
INITIALSOCIALTRANSPARENCY for all goods.
 * Logs some debug info if the test individual is involved.
 */
int makenewfriends(int me, int suggestedid) {
    int leastusefulfriendid = getleastusefulfriendid(me);
    int newfriendisnotsonew;
    int toomanyloops = 0;
    // If we were not approached by a friend (no specific suggestion), pick a
random individual as a new acquaintance
    if (suggestedid == FALSE) {
        do {
            newfriendisnotsonew = FALSE;
            suggestedid = randomindividual();
            if (suggestedid == me) {
                newfriendisnotsonew = TRUE;  // avoid picking oneself
            } else {
                // Ensure `suggestedid` is not already an existing friend
                for (int somefriend = 1; somefriend < MAXGROUP; somefriend+
+) {
                    if (individual[me].relationship[somefriend].relationid
== suggestedid) {
                        newfriendisnotsonew = TRUE;
                        break;
                    }
                }
            }
            if (toomanyloops++ > 2 * MAXGROUP) {
                printf("toomanyloops, suggestedid %i ", suggestedid);
            }
        } while (newfriendisnotsonew);
    }
    // Use the identified slot (least useful friend) for the new friend
    int newfriendid = leastusefulfriendid;
    individual[me].relationship[newfriendid].relationid = suggestedid;
    individual[me].relationship[newfriendid].transactions = 2;  // give a
small positive interaction count to establish link
    if (me == TESTINDIVIDUAL)
        printf("me %i, mynewfriend %i, friendid %i \n", me, suggestedid,
newfriendid);
    if (suggestedid == TESTINDIVIDUAL)
        printf("indy %i, I am his new friend %i, friendid %i \n", me,
suggestedid, newfriendid);
    // Reset all per-good transaction data with the new friend
```

```
    for (int l = 1; l < SKILLS; l++) {
        individual[me].relationship[newfriendid].purchased[l] = 0;
        individual[me].relationship[newfriendid].sold[l] = 0;
        individual[me].relationship[newfriendid].transparency[l] =
INITIALSOCIALTRANSPARENCY;
    }
    return newfriendid;
}

/* checksurplus(debug): Debug function (currently commented out) to check for
surplus anomalies.
 * (In original code, it might reset negative surplus to 0 or cap surplus if
it exceeded 10x stocklimit, indicating a bug.)
 * This function is not actively used in the current simulation, but is left
for troubleshooting extreme cases.
 */
// void checksurplus(int debug) {
//     for (int i=1; i<POPULATION; i++) {
//         for (int j=1; j<SKILLS; j++) {
//             // If surplus goes negative or is excessively above stock
limit, log and adjust
//             // (In this version, code is commented out; in a bug scenario
it would correct the surplus).
//             // if (individual[i].surplus[j] < 0) {
//             //     if (individual[i].surplus[j] > 10 *
individual[i].stocklimit[j]) {
//             //         printf("checksurplus - surplus negative %lli,
setting to zero, debug code %i utility %i individual %i",
//             //                 individual[i].surplus[j], debug, j, i);
//             //         individual[i].surplus[j] = 0;
//             //     }
//             // }
//         }
//     }
// }

/* calculatemarketprice(need): Compute the new average market price for a
given product `need` at end of a period.
 * The market price is calculated as a weighted average of:
 *    (a) the average production cost of that good this period, and
 *    (b) last period's average price of that good,
 * weighted by the proportion of new production vs. historical production [30].
 * Specifically:
 *   - Sum the production cost across all agents: for each agent, cost =
producedthisperiod[need] / efficiency[need] (time used).
 *   - Sum the total producedthisperiod for that good across agents.
 *   - If total production this period > (POPULATION * elasticneed[need])
(i.e., supply exceeds total demand),
 *         then marketprice = weighted average of (sum of production cost /
total produced) and previous market.averageprice (weighted by new vs
historical production) [30] [41].
```

```
 *     Otherwise (supply not above demand), keep marketprice as last period's
price (no change).
 *   - Also update numberofrecentlyproduced (add this period's production)
and maxefficiency for that good.
 *   - Count how many agents are consumers/retailers/producers for this good
(based on their role).
 * This gives an emergent price signal: if a good is over-produced relative
to need, its price tends to reflect current production costs (potentially
lowering value), otherwise price remains steady 42 .
 */
float calculatemarketprice(int need) {
    float marketprice = 0.0f;
    double sumofproductioncost = 0.0f;
    market.producedthisperiod[need] = 0;
    market.maxefficiency[need] = 0.0f;
    for (int indy = 1; indy < POPULATION; indy++) {
        // accumulate total surplus for stat (totalsurplus updated later, not
used here)
        individual[indy].totalsurplus += individual[indy].surplus[need];
        // sum production cost = (output / efficiency) for each agent
        sumofproductioncost +=
((double)individual[indy].producedthisperiod[need] /
individual[indy].efficiency[need]);
        market.producedthisperiod[need] +=
individual[indy].producedthisperiod[need];
        if (individual[indy].efficiency[need] > market.maxefficiency[need]) {
            market.maxefficiency[need] = individual[indy].efficiency[need];
        }
        if (individual[indy].role[need] == CONSUMER)
market.numberofconsumers[need]++;
        if (individual[indy].role[need] == RETAILER)
market.numberofretailers[need]++;
        if (individual[indy].role[need] == PRODUCER)
market.numberofproducers[need]++;
    }
    // Determine market price:
    if (market.producedthisperiod[need] > (POPULATION *
market.elasticneed[need])) {
        // If production exceeds demand, price is a weighted average of new
production cost vs. historical price 30
        marketprice = (float)(
                        ((double)market.producedthisperiod[need] / (double)
(market.producedthisperiod[need] + market.numberofrecentlyproduced[need])) *
                        (sumofproductioncost /
(double)market.producedthisperiod[need])  // weighted by current output:
average cost this period
                        +
                        ((double)market.numberofrecentlyproduced[need] /
(double)(market.producedthisperiod[need] +
market.numberofrecentlyproduced[need])) *
                        (double)market.averageprice[need]  // weighted by
```

```
historical output: last period's price
                        );
    } else {
        // If supply does not exceed demand, keep last period's price (no
downward pressure)
        marketprice = market.averageprice[need];
    }
    market.numberofrecentlyproduced[need] +=
market.producedthisperiod[need]; // update rolling production count (for next
period's weighting)
    if (marketprice < 0 || marketprice > 100)
        printf("marketprice negative or over 100, marketprice
%f!!!!!!!!!!!!", marketprice);
    return marketprice;
}

/* evaluatemarketprices: Update global market statistics at end of a period
and adjust elastic needs (demand) based on prices.
 * Steps:
 * 1. Reset some aggregate stats (priceaverage,
numberoftotalrecentproduction, totalcost* in time).
 * 2. For each product i:
 *      - Reset role counters (numberofconsumers, etc.) for this period.
 *      - Call calculatemarketprice(i) to compute market.averageprice[i] for
this good [31].
 *      - Accumulate total recent production count and weighted price for
overall average.
 * 3. Compute market.priceaverage = weighted average price across all goods
(by production volume).
 * 4. Adjust Elastic Needs based on PRICEDEMANDELASTICITY mode:
 *      - If 0: elasticneed = basicneed (no price elasticity).
 *      - If 1: elasticneed[i] = basicneed[i] * (market.priceaverage /
market.averageprice[i])  (goods with above-average price have reduced demand,
below-average price increase demand linearly).
 *      - If 2: elasticneed[i] = basicneed[i] * ((market.priceaverage /
market.averageprice[i])^2) (quadratic elasticity: larger effect) [7].
 *    Sum all elasticneed to totalelastic.
 * 5. Limit change rate of elastic needs: ensure no good's elasticneed rises
above MAXRISEINELASTICNEED (1.01x) or drops below MAXDROPINELASTICNEED
(0.98x) of previouselasticneed [43].
 *    Re-normalize so total elasticneed matches periodlength (a two-step
normalization to correct rounding).
 * 6. Compute costofspoilsintime and costoftceintime for each good (cost in
time = quantity * price).
 * 7. Print utility statistics for each good:
 *      e.g., for each i: print average cost (prodcost), new elasticneed, max
efficiency, production vs need ratios, cost of spoils/time, cost of
transaction cost/time, and proportion of agents who are consumers/retailers/
producers [44] [45].
 * 8. Print totalelastic values for debugging (difference after
normalization).
```

```
 * This function essentially closes the economic loop each period: adjusting
next period's needs (demand) based on this period's price outcomes,
simulating a demand elasticity effect in the barter economy [46] [42].
 */
void evaluatemarketprices(void) {
    market.priceaverage = 0;
    market.numberoftotalrecentproduction = 0;
    market.totalcostofspoilsintime = 0;
    market.totalcostoftceintime = 0;
    for (int i = 1; i < SKILLS; i++) {
        market.numberofconsumers[i] = 0;
        market.numberofretailers[i] = 0;
        market.numberofproducers[i] = 0;
        market.averageprice[i] = calculatemarketprice(i);
        market.numberoftotalrecentproduction +=
market.numberofrecentlyproduced[i];
        market.priceaverage += (market.averageprice[i] *
market.numberofrecentlyproduced[i]);
    }
    market.priceaverage = market.priceaverage /
market.numberoftotalrecentproduction;
    long long totalelastic = 0;
    // Update elastic needs based on price elasticity setting
    for (int i = 1; i < SKILLS; i++) {
        market.previouselasticneed[i] = market.elasticneed[i];
        switch (PRICEDEMANDELASTICITY) {
            case 0: // No elasticity: demand fixed at basic needs
                market.elasticneed[i] = market.basicneed[i];
                break;
            case 1: // Linear elasticity: demand inversely proportional to
price (relative to average price) [47]
                market.elasticneed[i] = (market.basicneed[i] *
((market.priceaverage / market.averageprice[i])));
                break;
            case 2: // Quadratic elasticity: stronger response
                market.elasticneed[i] = (market.basicneed[i] *
((market.priceaverage / market.averageprice[i]) * (market.priceaverage /
market.averageprice[i])));
                break;
            default:
                printf("mistaken value in PRICEDEMANDELASTICITY");
        }
        totalelastic += market.elasticneed[i];
        market.costofspoilsintime[i] = (market.periodicspoils[i] *
market.averageprice[i]);
        market.totalcostofspoilsintime += market.costofspoilsintime[i];
        market.costoftceintime[i] = (market.periodictcecost[i] *
market.averageprice[i]);
        market.totalcostoftceintime += market.costoftceintime[i];
    }
    // Calibrate elastic needs so that total remains equal to periodlength
```

```
(the total available time per agent), with limits on rate of change
    long long totalelastic2 = 0;
    for (int i = 1; i < SKILLS; i++) {
        // Normalize elasticneed by totalelastic first
        market.elasticneed[i] = market.elasticneed[i] *
((double)market.periodlength / totalelastic);
        // Limit how fast elastic need can increase or decrease compared to
last period 43
        if (market.elasticneed[i] > MAXRISEINELASTICNEED *
market.previouselasticneed[i]) {
            market.elasticneed[i] = MAXRISEINELASTICNEED *
market.previouselasticneed[i];
        }
        if (market.elasticneed[i] < MAXDROPINELASTICNEED *
market.previouselasticneed[i]) {
            market.elasticneed[i] = MAXDROPINELASTICNEED *
market.previouselasticneed[i];
        }
        totalelastic2 += market.elasticneed[i];
    }
    // Second normalization after limiting, to correct any slight drift in
totals
    long long totalelastic3 = 0;
    for (int i = 1; i < SKILLS; i++) {
        market.elasticneed[i] = market.elasticneed[i] *
((double)market.periodlength / totalelastic2);
        totalelastic3 += market.elasticneed[i];
        // Print detailed stats for this utility i
        printf("need %i prodcost %.4f elasticneed %.1f maxeff %.2f prodnow/
pEneed %.2f recentprod/prEneed %.2f COspoilsPEneed %.2f COtcecostPEneed %.2f,
Cons %.3f Ret %.3f Prod %.3f\n",
               i, market.averageprice[i], (float)market.elasticneed[i],
market.maxefficiency[i],
               (float)market.producedthisperiod[i] / (POPULATION *
market.elasticneed[i]),
               (float)market.numberofrecentlyproduced[i] / (POPULATION *
market.elasticneed[i] * (HISTORY + 1)),
               (float)(((market.averageprice[i] / market.priceaverage) *
market.periodicspoils[i]) / (market.elasticneed[i] * POPULATION)),
               (float)(((market.averageprice[i] / market.priceaverage) *
market.periodictcecost[i]) / (market.elasticneed[i] * POPULATION)),
               (float)market.numberofconsumers[i] / POPULATION,
               (float)market.numberofretailers[i] / POPULATION,
               (float)market.numberofproducers[i] / POPULATION);
    }
    printf("totalelastic %lli %lli %lli difference of 3. to periodlength
%lli\n", totalelastic, totalelastic2, totalelastic3, totalelastic3 -
market.periodlength);
}

/* getfriendexchangeindexes(me, myneed): Evaluate all possible exchanges for
```

```
agent `me` to satisfy need `myneed` via friends.
 * For each friend connection and for each good that `me` could offer
(mygift):
 *   - Check that friend exists (relationid > 0).
 *   - Determine friend's ID (myfriend) and the index in friend's
relationship where `me` appears (measfriendindex).
 *   - Only consider trades where:
 *         friend has surplus of myneed above their own requirements
(surplus[myneed] > elasticneed*needslevel + INTMULTIPLIER, meaning friend can
spare that good),
 *         friend has capacity/need for mygift (friend's surplus[mygift] is
below their stock capacity, i.e., less than giftmaxlevel),
 *         and `me` has surplus of mygift above own requirements
(me.surplus[mygift] > elasticneed[mygift]*needslevel + INTMULTIPLIER).
 *   - Compute receivingtransparency: trust factor friend has in me for
mygift (if I'm not in their friend list yet, use INITIALSOCIALTRANSPARENCY).
 *   - Compute fexchangeindex for this potential trade:
 *         fexchangeindex = (friend.purchaseprice[mygift] /
friend.salesprice[myneed]) * (me->friend transparency for myneed)
 *                          - (me.salesprice[mygift] /
(me.purchaseprice[myneed] * receivingtransparency)) 25 48 .
 *     This formula yields a positive index if the trade is mutually
beneficial (a positive-sum exchange). It essentially compares the ratio of
how the friend values mygift vs myneed (adjusted by my trust in friend for
myneed)
 *     to how I value myneed vs mygift (adjusted by friend's trust in me for
mygift).
 *   - Then adjust fexchangeindex by roles to bias toward trading with
producers/retailers:
 *         multiply by individual[me].role[mygift] and by
individual[myfriend].role[myneed]. (Roles are 10,11,12, so this boosts
exchanges where mygift is something I'm a producer/retailer of, or where
friend is producer/retailer of myneed) 11 .
 *         If I am a producer of myneed (i.e., I'm self-sufficient in it),
divide the index by 2 (less incentive to trade for something I produce) 49 .
 *   - If friend can't offer any valid exchange for myneed, set
fexchangeindex for that friend to -1 (skip).
 *   - Track the best exchange (highest index) found: store in global
bestexchange structure.
 * End result: bestexchange will contain the friend and gift that yields the
highest positive exchange index for satisfying need `myneed`, or index <= 0
if none are beneficial.
 */
void getfriendexchangeindexes(int me, int myneed) {
    int myfriend;
    int measfriendindex;
    float receivingtransparency;
    bestexchange.index = -1.0f;
    bestexchange.surplusproductid = 0;
    // Iterate through each of agent `me`'s friends
    for (int friend = 1; friend < MAXGROUP; friend++) {
```

```
        myfriend = individual[me].relationship[friend].relationid;
        measfriendindex = getmefriendindex(myfriend, me);
        if (myfriend > 0) {
            // Check all possible surplus goods `me` can offer (mygift)
            for (int mygift = 1; mygift < SKILLS; mygift++) {
                // `me` wants `myneed` and will give `mygift`. Friend
`myfriend` must accept `mygift` and give `myneed`.
                // Ensure adequate surpluses are present:
                int giftmaxlevel;
                if (individual[myfriend].role[mygift] == RETAILER) {
                    // If friend is a retailer for mygift, they can take
mygift up to stocklimit - 10 (some margin)
                    giftmaxlevel = individual[myfriend].stocklimit[mygift] -
INTMULTIPLIER;
                } else {
                    // If friend is consumer/producer, they consider up to
their elastic need (needslevel * elasticneed) minus a unit
                    giftmaxlevel = (int)((market.elasticneed[mygift] *
individual[myfriend].needslevel) - INTMULTIPLIER);
                }
                if (
                    // Friend can give myneed: has surplus of myneed beyond
their own need
                    (individual[myfriend].surplus[myneed] >
((market.elasticneed[myneed] * individual[myfriend].needslevel) +
INTMULTIPLIER))
                    &&
                    // Friend is not already at capacity for mygift (they
have room or need for mygift)
                    (individual[myfriend].surplus[mygift] < giftmaxlevel)
                    &&
                    // I have enough surplus of mygift to give (beyond my own
requirement)
                    (individual[me].surplus[mygift] >
((market.elasticneed[mygift] * individual[me].needslevel) + INTMULTIPLIER))
                   ) {
                    // Determine transparency factors for this pair:
                    if (measfriendindex == 0) {
                        receivingtransparency = INITIALSOCIALTRANSPARENCY;
                    } else {
                        receivingtransparency =
individual[myfriend].relationship[measfriendindex].transparency[mygift];
                    }
                    // Compute exchange index (positive if both parties gain)
 25 .
                    fexchangeindex[friend][mygift] =
                        ((individual[myfriend].purchaseprice[mygift] /
individual[myfriend].salesprice[myneed]) *

individual[me].relationship[friend].transparency[myneed])
                        -
```

```
                        (individual[me].salesprice[mygift] /
(individual[me].purchaseprice[myneed] * receivingtransparency));
                    // Bias towards friends who are producers/retailers:
                    // Favor trades where I'm giving something I'm more of a
seller of, and receiving something the friend is a seller of [11].
                    fexchangeindex[friend][mygift] =
individual[me].role[mygift] * fexchangeindex[friend][mygift];
                    fexchangeindex[friend][mygift] =
individual[myfriend].role[myneed] * fexchangeindex[friend][mygift];
                    if (individual[me].role[myneed] == PRODUCER) {
                        // If I produce myneed myself, I'm less desperate, so
reduce index
                        fexchangeindex[friend][mygift] =
fexchangeindex[friend][mygift] / 2;
                    }
                    // Debug prints (commented out) could show exchange index
components for TESTINDIVIDUAL
                    // if (me == TESTINDIVIDUAL) {
                    //     printf("exchindex %f for need %i from friend
%i\n", fexchangeindex[friend][mygift], myneed, myfriend);
                    //     printf("myfreceivespp %f myfgivessp %f transp %f
mygiftsp %f mereceivepp %f \n",
                    //            individual[myfriend].purchaseprice[mygift],
                    //            individual[myfriend].salesprice[myneed],
                    //
individual[me].relationship[friend].transparency[myneed],
                    //            individual[me].salesprice[mygift],
individual[me].purchaseprice[myneed]);
                    // }
                } else {
                    fexchangeindex[friend][mygift] = -1.0f; // trade not
possible or not beneficial
                }
                // Update bestexchange if this is the highest index so far
                if (bestexchange.index < fexchangeindex[friend][mygift]) {
                    bestexchange.index = fexchangeindex[friend][mygift];
                    bestexchange.individualid = myfriend;
                    bestexchange.friendid = friend;
                    bestexchange.surplusproductid = mygift;
                    bestexchange.exchangetype = NETWORK;
                }
            } // end for mygift
        } else {
            // No friend in this slot
            for (int mygift = 1; mygift < SKILLS; mygift++) {
                fexchangeindex[friend][mygift] = -1.0f;
            }
        }
    }
}
```

```
/* excecutefexchange(dealtype, me, myneed, maxneed): Execute the best
exchange (bestexchange global struct) for agent `me` to obtain `myneed`.
 * Preconditions: bestexchange is set by getfriendexchangeindexes with a
positive index and chosen friend/gift.
 * dealtype: SURPLUSDEALS (1) if this exchange is for surplus acquisition
(not immediate consumption), CONSUMPTION (2) if for satisfying needs.
 * `maxneed`: the maximum amount of `myneed` to acquire in this exchange.
 *
 * This function:
 *   - Calculates the exchange rate (switchaverage) again (some formula
adjustments compared to commented code).
 *   - Determines `maxexchange`: the quantity of myneed that will actually be
traded this iteration.
 *        * It starts as: ((my surplus of mygift beyond my requirement) *
receivingtransparency) / switchaverage.
 *          This is basically how much need I can get given how much of
mygift I can afford to give and taking into account trust (transparency
reduces effective gift) [50].
 *        * If this computed maxexchange is very small (<= INTMULTIPLIER/2,
i.e. <=5 units in scaled terms), it's not worth trading (considered
negligible), so abort this trade (set maxexchange=0 and invalidate this gift
for all friends to avoid wasted attempts) [51] [52].
 *        * Otherwise, if maxexchange > maxneed (the need we still have), cap
it to maxneed (don't trade for more than we need).
 *        * Then check the friend's available surplus of myneed:
 *              If friend’s surplus of myneed beyond their own needs (times
friend's transparency to me) is less than maxexchange, reduce maxexchange to
that (friend can't give more than they can spare) [53] [54].
 *        * If friend is a RETAILER for mygift: ensure we don't give them
more of mygift than fits in their stocklimit (if giving too much would exceed
their inventory, limit trade) [55].
 *        * If friend is not a retailer: ensure we don't give them more of
mygift than their immediate need (elasticneed*needslevel - surplus) for
mygift (non-retailers only accept what they can use now) [56].
 *        * Finally, subtract a small value (INTMULTIPLIER/3 ~ 3.33) from
maxexchange to avoid rounding issues or ending up with negative surplus due
to tight calculations [57].
 *   - If friend did not know `me` (measfriendindex was 0), call
makenewfriends for friend to add `me` to their network (since an exchange
occurred, the friendship becomes two-way).
 *   - If after all adjustments maxexchange >= INTMULTIPLIER/2 (>=5), proceed
with the exchange:
 *        * If dealtype == SURPLUSDEALS (meaning `me` is acquiring surplus
beyond need):
 *              - Increase `me.surplus[myneed]` by maxexchange (got extra
goods) [58].
 *              - (If this pushes me.surplus above stocklimit, a debug
warning is printed).
 *          Else (dealtype == CONSUMPTION):
 *              - Decrease `me.need[myneed]` by maxexchange (need satisfied
by that much) [59].
```

```
 *              - If need falls below half a unit, set it to 0 (fully
satisfied).
 *        * In either case, update `me`'s surplus of mygift: reduce by the
amount given away. The amount given of mygift = (maxexchange *
switchaverage) / receivingtransparency (the effective cost in mygift units
considering trust).
 *              - If this calculation makes mygift surplus negative, print a
warning (shouldn't happen due to earlier adjustments).
 *              - If I still have remaining need (maxneed >= INTMULTIPLIER
after trade) but now mygift surplus dropped below 1 unit, set mygift surplus
= 0 and mark that gift's index -1 for all friends (no longer available to
trade) [60].
 *        * Update friend's inventory: friend gives away myneed, so
friend.surplus[myneed] decreases by (maxexchange / friend-
>transparency_to_me_for_myneed) (they effectively spend a bit more if
transparency <1) [61].
 *        * Update global transaction cost metrics:
 *              - market.periodictcecost[myneed] += ((maxexchange / friend-
>transparency) - maxexchange). This is the extra units of myneed that had to
be given by friend due to imperfect transparency (a measure of transaction
cost for myneed) [62].
 *              - market.periodictcecost[mygift] += (((maxexchange *
switchaverage) / my->receivingtransparency) - (maxexchange * switchaverage)).
This is the extra units of mygift I had to give due to imperfect transparency
on my side [63].
 *        * Update both parties' records of sold/purchased:
 *              - `me.recentlysold[mygift]` += (maxexchange * switchaverage)
(I sold some of mygift) [64].
 *              - `me.soldthisperiod[mygift]` += same amount.
 *              - `me.recentlypurchased[myneed]` += maxexchange (I purchased
myneed) [65].
 *              - `me.purchasedthisperiod[myneed]` += maxexchange.
 *              - friend.recentlysold[myneed] += (maxexchange / friend-
>transparency_for_myneed) (friend sold myneed, had to give a bit extra if
trust < 1) [66].
 *              - friend.soldthisperiod[myneed] += same.
 *              - friend.recentlypurchased[mygift] += (maxexchange *
switchaverage) (friend effectively "purchased" mygift with their goods) [67].
 *              - friend.purchasedthisperiod[mygift] += same.
 *              - friend.surplus[mygift] += (maxexchange * switchaverage)
(friend's surplus of mygift increases by what I gave) [68].
 *              - If this pushes friend.surplus[mygift] above
friend.stocklimit*MAXNEEDSINCREASE (i.e., way beyond what they normally
store), print a warning.
 *        * Compute exchangevaluetome and exchangevaluetofriend (these are
ratios indicating how favorable the exchange was to each side: >1 means they
gained value) and print for TESTINDIVIDUAL for debug [69].
 *        * Value adjustment: If both sides had exchange value corrections >1
(meaning both benefited), accumulate their current price levels into
sumperiodpurchase/salesvalue to adjust prices later [70] [71].
 *              (This is essentially collecting the trade's implied price
```

```
into a running average that will be used in adjustpurchaseprice/
adjustsalesprice at period end.)
 *        * Increment transaction counters:
 *              - me.purchasetimes[myneed]++, me.salestimes[mygift]++ [70].
 *              - friend.purchasetimes[mygift]++, friend.salestimes[myneed]+
+ [72].
 *        * Update social relationship stats:
 *              - me.relationship[friendid].transactions++ (increment total
transactions with that friend) [73].
 *              - me.relationship[friendid].purchased[myneed]++ (record that
I bought one unit of myneed from friend) [74].
 *              - me.relationship[friendid].sold[mygift]++ (record I sold one
unit of mygift).
 *              - friend.relationship[measfriendindex].transactions++ (friend
records a transaction with me) [75].
 *              - friend.relationship[measfriendindex].purchased[mygift]++
(friend purchased mygift from me).
 *              - friend.relationship[measfriendindex].sold[myneed]++ (friend
sold myneed to me).
 *   - After executing, mark the used exchange option as used:
fexchangeindex[friendid][surplusproductid] = -1 (so it won't be chosen again
in same cycle).
 * In summary, this function updates both agents' state to reflect the trade,
including inventory changes, transaction cost accounting, and records for
price adjustments and trust.
 */
void excecutefexchange(int dealtype, int me, int myneed, long long maxneed) {
    long long maxexchange;
    int mygift = bestexchange.surplusproductid;
    int measfriendindex = getmefriendindex(bestexchange.individualid, me);
    float receivingtransparency;
    if (measfriendindex == 0) {
        receivingtransparency = INITIALSOCIALTRANSPARENCY;
    } else {
        receivingtransparency =
individual[bestexchange.individualid].relationship[measfriendindex].transparency[mygift];
    }
    // Determine exchange rate (switchaverage) - approximate "price" between
goods, taking into account both parties' valuations and transparency.
    // (The original formula commented out was inverted and partially tested.
The one used here tries to incorporate transparency more symmetrically.)
    // switchaverage is roughly: average of [friend's salesprice(myneed)/
purchaseprice(mygift)*(my transparency)  AND  my purchaseprice(myneed)/
salesprice(mygift)*(friend transparency)] [26].
    bestexchange.switchaverage =
        (
          (individual[bestexchange.individualid].salesprice[myneed] /
(individual[bestexchange.individualid].purchaseprice[mygift] *
individual[me].relationship[bestexchange.friendid].transparency[myneed]))
          +
          ((individual[me].purchaseprice[myneed] * receivingtransparency) /
```

```
individual[me].salesprice[mygift])
        ) / 2;
    // Now commit exchange and update sales & purchase prices (some
adjustments deferred to end-of-period functions).
    // Calculate maximum exchange quantity for this trade iteration:
    maxexchange = (long long)(((individual[me].surplus[mygift] -
(individual[me].needslevel * market.elasticneed[mygift])) *
receivingtransparency) / bestexchange.switchaverage);
    if (maxexchange <= (INTMULTIPLIER / 2)) {
        // If the potential trade amount is extremely small (<=0.5 units in
scaled terms), skip it.
        // Mark this gift as not worth trying with other friends this cycle:
        maxexchange = 0;
        for (int i = 1; i < MAXGROUP; i++) {
            fexchangeindex[i][bestexchange.surplusproductid] = -1.0f;
        }
    } else {
        if (maxexchange > maxneed) {
            maxexchange = maxneed;  // Don't trade for more than we need (if
consumption) or more than stock gap (if surplus deal).
        }
        // Limit by friend's available surplus of myneed (taking into account
their transparency to me for myneed):
        if (((individual[bestexchange.individualid].surplus[myneed] -
(individual[bestexchange.individualid].needslevel *
market.elasticneed[myneed])) *
individual[me].relationship[bestexchange.friendid].transparency[myneed]) <
maxexchange) {
            maxexchange = (long long)(

(individual[bestexchange.individualid].surplus[myneed] -
(individual[bestexchange.individualid].needslevel *
market.elasticneed[myneed]))
                          *
individual[me].relationship[bestexchange.friendid].transparency[myneed]
                         );
        }
        // If friend is a retailer, limit by their stock capacity for mygift
(so we don't overload them with goods to resell) [55].
        if (individual[bestexchange.individualid].role[mygift] == RETAILER) {
            if ((individual[bestexchange.individualid].stocklimit[mygift] -
individual[bestexchange.individualid].surplus[mygift]) < (long long)
(maxexchange * bestexchange.switchaverage)) {
                maxexchange = (long long)(

(individual[bestexchange.individualid].stocklimit[mygift] -
individual[bestexchange.individualid].surplus[mygift])
                             / bestexchange.switchaverage
                            );
            }
        } else {
```

```
            // If friend is not a retailer, they will only accept as much of
mygift as they immediately need (they don't stock extra).
            if (((individual[bestexchange.individualid].needslevel *
market.elasticneed[mygift]) -
individual[bestexchange.individualid].surplus[mygift]) < (long long)
(maxexchange * bestexchange.switchaverage)) {
                maxexchange = (long long)(

((individual[bestexchange.individualid].needslevel *
market.elasticneed[mygift]) -
individual[bestexchange.individualid].surplus[mygift])
                                / bestexchange.switchaverage
                               );
            }
        }
        // Avoid edge cases where rounding errors cause negative surplus
later: subtract a small amount from maxexchange [57].
        maxexchange -= INTMULTIPLIER / 3;
        if (measfriendindex == 0) {
            // If friend didn't know me, establish reciprocal friendship now
            measfriendindex = makenewfriends(bestexchange.individualid, me);
        }
        if (maxexchange >= (INTMULTIPLIER / 2)) {  // Only proceed if a
meaningful amount remains
            maxneed = maxneed - maxexchange;
            if (dealtype == SURPLUSDEALS) {
                // I'm acquiring surplus (not directly reducing my need but
increasing stock for potential future use/trade)
                individual[me].surplus[myneed] += maxexchange;
                if (individual[me].surplus[myneed] >
individual[me].stocklimit[myneed])
                    printf("surplusdeals ... stocklimit exceeded for
myneed!!!");
            } else {
                // I'm satisfying an immediate need
                individual[me].need[myneed] -= maxexchange;
                if (individual[me].need[myneed] < INTMULTIPLIER / 2)
individual[me].need[myneed] = 0;
            }
            if (individual[me].stocklimit[myneed] <
individual[me].surplus[myneed])
                printf("surplus above stocklimit due to maxexchange
overflow??");
            // Update my records: I sold some of mygift and bought some of
myneed
            individual[me].recentlysold[mygift] += (long long)(maxexchange *
bestexchange.switchaverage);
            individual[me].soldthisperiod[mygift] += (long long)(maxexchange
* bestexchange.switchaverage);
            individual[me].recentlypurchased[myneed] += maxexchange;
            individual[me].purchasedthisperiod[myneed] += maxexchange;
```

```
            // Reduce my surplus of mygift by the amount given (adjusted for
transparency I perceive: I effectively give a bit extra if my transparency <
1)
            individual[me].surplus[mygift] -= (long long)((double)
(maxexchange * bestexchange.switchaverage) / receivingtransparency);
            if (individual[me].surplus[mygift] < 0)
                printf("mygift surplus negative when doing exchange");
            if ((maxneed >= INTMULTIPLIER) &&
(individual[me].surplus[mygift] < INTMULTIPLIER)) {
                // If I still need more (maxneed not fulfilled) but I ran out
of this gift (surplus <1 after giving), mark this gift as unavailable for
further trades
                individual[me].surplus[mygift] = 0;
                for (int i = 1; i < MAXGROUP; i++) {
                    fexchangeindex[i][mygift] = -1.0f;
                }
            }
            // Update friend's records: friend sells some of myneed and buys
mygift
            individual[bestexchange.individualid].surplus[myneed] -=
                (long long)(maxexchange /
individual[me].relationship[bestexchange.friendid].transparency[myneed]);
            // Transaction cost (time cost of exchange inefficiency)
accounting:
            market.periodictcecost[myneed] += (((maxexchange /
individual[me].relationship[bestexchange.friendid].transparency[myneed]) -
maxexchange));
            market.periodictcecost[mygift] += ((((double)maxexchange *
bestexchange.switchaverage) / receivingtransparency) - maxexchange *
bestexchange.switchaverage);
            // Friend's sales and purchase counts:
            individual[bestexchange.individualid].recentlysold[myneed] +=
                (long long)(maxexchange /
individual[me].relationship[bestexchange.friendid].transparency[myneed]);
            individual[bestexchange.individualid].soldthisperiod[myneed] +=
                (long long)(maxexchange /
individual[me].relationship[bestexchange.friendid].transparency[myneed]);
            individual[bestexchange.individualid].recentlypurchased[mygift]
+=
                (long long)(maxexchange * bestexchange.switchaverage);

individual[bestexchange.individualid].purchasedthisperiod[mygift] +=
                (long long)(maxexchange * bestexchange.switchaverage);
            // Friend's inventory update: friend gains mygift (what I gave in
exchange)
            individual[bestexchange.individualid].surplus[mygift] +=
                (long long)(maxexchange * bestexchange.switchaverage);
            if (individual[bestexchange.individualid].surplus[mygift] >
                (individual[bestexchange.individualid].stocklimit[mygift] *
MAXNEEDSINCREASE)) {
                printf("surplusdeals ... myfriends stocklimit %lli exceeded,
```

```
surplus %lli while need is %lli for mygift!!!\n",

individual[bestexchange.individualid].stocklimit[mygift],
                      individual[bestexchange.individualid].surplus[mygift],
                      (long long)(market.elasticneed[mygift] *
individual[bestexchange.individualid].needslevel));
            }
            // Calculate perceived exchange value for me and friend (if >1,
they gained compared to their own valuations).
            float exchangevaluetome =
(((float)individual[me].purchaseprice[myneed]) /
(((bestexchange.switchaverage / receivingtransparency) *
individual[me].salesprice[mygift])));
            float exchangevaluetofriend =
(((float)individual[bestexchange.individualid].purchaseprice[mygift] *
bestexchange.switchaverage) /

(individual[bestexchange.individualid].salesprice[myneed] /
individual[me].relationship[bestexchange.friendid].transparency[myneed]));
            if (me == TESTINDIVIDUAL) {
                printf("surplusdeals myneed %i  bestexchangeindex %f,
myxvalue (average >1) %f fxchvalue (average >1) %f\n",
                       myneed, bestexchange.index, exchangevaluetome,
exchangevaluetofriend);
            }
            // Adjust running totals for price adjustment later, if both
sides benefited (valuecorrection > 1 means exchange was positive-sum):
            float myvaluecorrection = 1.0f / InvSqrt(exchangevaluetome);
            float friendvaluecorrection = 1.0f /
InvSqrt(exchangevaluetofriend);
            if ((myvaluecorrection > 1) && (friendvaluecorrection > 1)) {
                // Both gained value -> incorporate this into their average
price stats
                individual[me].sumperiodpurchasevalue[myneed] +=
individual[me].purchaseprice[myneed] / myvaluecorrection;
                individual[me].sumperiodsalesvalue[mygift] +=
individual[me].salesprice[mygift] * myvaluecorrection;

individual[bestexchange.individualid].sumperiodpurchasevalue[mygift] +=

individual[bestexchange.individualid].purchaseprice[mygift] /
friendvaluecorrection;

individual[bestexchange.individualid].sumperiodsalesvalue[myneed] +=
                    individual[bestexchange.individualid].salesprice[myneed]
* friendvaluecorrection;
            } else {
                printf("mistaken valuecorrection
- !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");
                // If corrections <=1, we don't update price perception (no
clear mutual gain)
```

```
                // (Commented-out alternative adjustments are omitted for
simplicity)
            }
            // Update relationship event statistics (both directions)
            individual[me].relationship[bestexchange.friendid].transactions+
+;

individual[me].relationship[bestexchange.friendid].purchased[myneed]++;
            individual[me].relationship[bestexchange.friendid].sold[mygift]+
+;

individual[bestexchange.individualid].relationship[measfriendindex].transactions+
+;

individual[bestexchange.individualid].relationship[measfriendindex].purchased[mygift]
++;

individual[bestexchange.individualid].relationship[measfriendindex].sold[myneed]
++;
        }
    }
    // Mark this exchange option as used (so we don't consider the same
friend-gift pair again in this round)
    fexchangeindex[bestexchange.friendid][bestexchange.surplusproductid] =
-1.0f;
}

/* satisfyneedsbyexchange(me): Attempt to satisfy agent `me`'s unsatisfied
needs by trading with friends.
 * - For each need (1..SKILLS-1), if `me` does not already have enough
surplus of that item to meet their needs (surplus < needslevel *
elasticneed):
 *       * Compute bestexchange by calling getfriendexchangeindexes(me,
myneed).
 *       * If a best exchange exists (bestexchange.index > 0):
 *             While there is a beneficial exchange (index > 0) and the need
is still >= 1 unit:
 *                - If bestexchange.exchangetype == NETWORK (the only case
here), call excecutefexchange(CONSUMPTION, me, myneed,
individual[me].need[myneed]) to perform the trade for as much as needed [76].
 *                - After executing one trade, reset bestexchange.index and
surplusproductid, then search fexchangeindex matrix for the next best
exchange (since the previous might have partially fulfilled the need or
exhausted a gift).
 *                  (The code does this by scanning fexchangeindex for the
next highest value) [77] [78].
 *                - Loop until no beneficial exchange remains or need is
fully satisfied.
 *             If too many iterations occur (safety to prevent infinite loop,
though ideally should not happen), break.
 *       * If after trying all exchanges, the need is still not fully
```

```
satisfied, then that need will have to be produced by the agent.
 * - Finally, after attempting all needs, check if all needs were covered by
surplus (surpluscoversallneeds flag).
 * Returns 1 if all needs are covered by available surplus (no production
needed), 0 if some needs remain unsatisfied (will require production).
 */
int satisfyneedsbyexchange(int me) {
    int surpluscoversallneeds = 1;
    for (int myneed = 1; myneed < SKILLS; myneed++) {
        bestexchange.index = -1;
        bestexchange.surplusproductid = 0;
        // Only process needs where current surplus is less than required
need (unsatisfied need)
        if (individual[me].surplus[myneed] < (market.elasticneed[myneed] *
individual[me].needslevel)) {
            // Find the best exchange options (friend and gift) to satisfy
this need
            getfriendexchangeindexes(me, myneed);
            if ((individual[me].surplus[bestexchange.surplusproductid] <
INTMULTIPLIER) && (bestexchange.index > 0)) {
                printf("getfriendexcind brings back lacking surplus for
mygift, bestexchangeindex %f surplusprid %i\n",
                        bestexchange.index, bestexchange.surplusproductid);
            }
            int toomanyloops = 0;
            // While a beneficial exchange exists and we still have some need
left:
            while ((bestexchange.index > 0) && (individual[me].need[myneed]
>= INTMULTIPLIER)) {
                if (toomanyloops++ > (SKILLS * MAXGROUP)) {
                    printf("its me %i and exchange is running hot,
bestexchindex is %f\n", me, bestexchange.index);
                    break;
                }
                if (individual[me].surplus[bestexchange.surplusproductid] <
INTMULTIPLIER) {
                    printf("1 while loop brings back lacking surplus for me
%i and mygift, bestexchangeindex %f surplusprid %i, surplus %lli, basicneed
%lli\n",
                            me, bestexchange.index,
bestexchange.surplusproductid,

individual[me].surplus[bestexchange.surplusproductid], market.basicneed[1]);
                }
                if (bestexchange.exchangetype == NETWORK) {
                    excecutefexchange(CONSUMPTION, me, myneed,
individual[me].need[myneed]);
                } else {

printf("nonexistent optional market mechanism selected in
satisfyneedsbyexchange");
```

```
                    // If MARKET were implemented, we'd handle it here (not
used in this simulation)
                }
                // Prepare to search for the next best exchange (reset
bestexchange and find next highest in fexchangeindex)
                bestexchange.index = -0.5f;
                bestexchange.surplusproductid = 0;
                // Find next best exchange index among all friends/gifts
                for (int friend = 1; friend < MAXGROUP; friend++) {
                    for (int gift = 1; gift < SKILLS; gift++) {
                        // Only consider if this potential exchange index is
higher than current best and I still have surplus of that gift to offer
                        if ((bestexchange.index < fexchangeindex[friend]
[gift]) && (individual[me].surplus[gift] > INTMULTIPLIER)) {
                            bestexchange.index = fexchangeindex[friend]
[gift];
                            bestexchange.individualid =
individual[me].relationship[friend].relationid;
                            bestexchange.friendid = friend;
                            bestexchange.surplusproductid = gift;
                            bestexchange.exchangetype = NETWORK;
                        }
                    }
                }
                // Loop continues if a new bestexchange (index > 0) is found
and need still not zero
            } // endwhile
        } // end if (unsatisfied need)
    } // end for each need
    // After trying to satisfy by exchange, check if any need is still
unsatisfied (surplus < required level)
    for (int myneed = 1; myneed < SKILLS; myneed++) {
        if (individual[me].surplus[myneed] < (individual[me].needslevel *
market.elasticneed[myneed])) {
            surpluscoversallneeds = 0;
            break;
        }
    }
    return surpluscoversallneeds;
}

/* makesurplusdeals(me): After satisfying needs, agent `me` tries to trade to
optimize their surplus holdings.
 * Specifically, identify goods where:
 *    - recentlysold[need] > (recentlyproduced[need] - elasticneed[need])
i.e., the agent sold more than they produced (they could have sold more if
they had more) [79],
 *    - and surplus[need] < (stocklimit[need] - elasticneed[need]) i.e.,
current stock is not full (they have room to store more) [79].
 * For such goods, this agent would benefit from acquiring more of that good
(because there was demand for it).
```

```
 * Then for each such need:
 *    - Use getfriendexchangeindexes to find best trades where `me` offers
some other surplus to get more of this `myneed`.
 *    - While a beneficial exchange exists and the agent's surplus for that
need is still below stock limit:
 *         If bestexchange.exchangetype == NETWORK, call
excecutefexchange(SURPLUSDEALS, me, myneed, (stocklimit - current surplus))
80 .
 *         (This will trade enough to try to fill the stock up to the limit.)
 *         Then same loop mechanism as satisfyneedsbyexchange: reset
bestexchange and find next best until no more beneficial trades or stock is
full.
 * This simulates "entrepreneurial" trading: if the agent sees that a product
was in high demand (they sold a lot relative to what they had), they try to
acquire more of it to sell next time.
 */
void makesurplusdeals(int me) {
    // We target products where demand exceeded our supply and we have room
to stock more
    for (int myneed = 1; myneed < SKILLS; myneed++) {
        bestexchange.index = -1;
        bestexchange.surplusproductid = 0;
        if ((individual[me].recentlysold[myneed] >
(individual[me].recentlyproduced[myneed] - market.elasticneed[myneed])) &&
            (individual[me].surplus[myneed] <
(individual[me].stocklimit[myneed] - market.elasticneed[myneed]))) {
            getfriendexchangeindexes(me, myneed);
            if ((individual[me].surplus[bestexchange.surplusproductid] <
(INTMULTIPLIER)) && (bestexchange.index > 0)) {

printf("getfriendexcind in spldls brings back lacking surplus for mygift,
bestexchangeindex %f surplusprid %i\n",
                        bestexchange.index, bestexchange.surplusproductid);
            }
            int toomanyloops = 0;
            // While beneficial exchange exists and haven't filled stock
limit for this need:
            // (Note: condition uses surplus < stocklimit, so agent is trying
to fill up storage for this good)
            while ((bestexchange.index > 0) &&
(individual[me].surplus[myneed] < (individual[me].stocklimit[myneed]))) {
                if (toomanyloops++ > (SKILLS * MAXGROUP)) {
                    printf("its me %i and exchange is running hot in
mksplsdls, bestexchindex is %f\n", me, bestexchange.index);
                    break;
                }
                if (individual[me].surplus[bestexchange.surplusproductid] <
INTMULTIPLIER) {
                    printf("2while loop brings back lacking surplus for
mygift, bestexchangeindex %f surplusprid %i, exchtype %i\n",
                            bestexchange.index,
```

```
bestexchange.surplusproductid, bestexchange.exchangetype);
                }
                if (bestexchange.exchangetype == NETWORK) {
                    // Execute exchange to acquire as much of myneed as
possible (up to stocklimit) 80
                    excecutefexchange(SURPLUSDEALS, me, myneed, (long long)
(individual[me].stocklimit[myneed] - individual[me].surplus[myneed]));
                    // (Debug for test individual commented out)
                    // if (me == TESTINDIVIDUAL) printf("surplusdeals myneed
%i bestexchangeindex %f, myexchangevalue %f myfriendexchvalue %f\n", myneed,
bestexchange.index);
                } else {
                    printf("nonexistent market mechanism selected in
makesurplusdeals");
                    // If MARKET mechanism existed, it would be handled here
(not applicable in this model)
                }
                // Reset and find next best exchange for this need
                bestexchange.index = -0.5f;
                bestexchange.surplusproductid = 0;
                for (int friend = 1; friend < MAXGROUP; friend++) {
                    for (int gift = 1; gift < SKILLS; gift++) {
                        if ((bestexchange.index < fexchangeindex[friend]
[gift]) && (individual[me].surplus[gift] > INTMULTIPLIER)) {
                            bestexchange.index = fexchangeindex[friend]
[gift];
                            bestexchange.individualid =
individual[me].relationship[friend].relationid;
                            bestexchange.friendid = friend;
                            bestexchange.surplusproductid = gift;
                            bestexchange.exchangetype = NETWORK;
                        }
                    }
                }
            } // endwhile
        } // end if conditions
    } // end for each need
}

/* produceneed(me): Use remaining time for agent `me` to produce goods to
fulfill any unmet needs.
 * For each need:
 *   If need > 0, produce that amount:
 *       - Decrease periodremaining by (need / efficiency) time units.
 *       - Increase recentlyproduced and producedthisperiod by the amount
produced.
 *       - Log production if this is the TESTINDIVIDUAL (for debugging).
 *       - Set need to 0 (fully satisfied by production).
 * After producing all needed items, if periodremaining < 0 (meaning they ran
out of time before finishing), increment timeout (indicates unsustainable
situation).
```

```
 * This is the core consumption production step: whatever an agent could not
obtain through trade, they produce themselves (if they have time).
 */
void produceneed(int me) {
    for (int myneed = 1; myneed < SKILLS; myneed++) {
        if (individual[me].need[myneed] > 0) {
            // Use time to produce the needed goods
            individual[me].periodremaining -=
((float)individual[me].need[myneed] / individual[me].efficiency[myneed]);
            individual[me].recentlyproduced[myneed] +=
individual[me].need[myneed];
            individual[me].producedthisperiod[myneed] +=
individual[me].need[myneed];
            if (me == TESTINDIVIDUAL) {
                printf("Producing myneed %i %lli elasticneed %lli
periodremaining %lli producednow %lli\n",
                       myneed, individual[me].need[myneed], (long
long)market.elasticneed[myneed],
                       (long long)individual[me].periodremaining,
individual[me].producedthisperiod[myneed]);
            }
            individual[me].need[myneed] = 0;
        }
    }
    if (individual[me].periodremaining < 0) {
        // If time ran out (negative remaining), mark a timeout (needs not
fully satisfied in time)
        // This is rare if at all, since needslevel adjustments try to
prevent unsustainable demand.
        individual[me].timeout++;
    }
}

/* producesmallneeds(me): Similar to produceneed, but intended to handle very
small remaining needs (less than 1 unit).
 * This function is used in extra rounds (like if an agent gets a small
additional need due to leisure spending).
 * It produces any need that is >0 but <1 (INTMULTIPLIER) to clear out
fractional needs without significantly impacting time.
 * Essentially, it's a finer-grained production step for incremental needs.
 */
void producesmallneeds(int me) {
    for (int myneed = 1; myneed < SKILLS; myneed++) {
        if ((individual[me].need[myneed] > 0) &&
(individual[me].need[myneed] < INTMULTIPLIER)) {
            individual[me].periodremaining -=
((float)individual[me].need[myneed] / individual[me].efficiency[myneed]);
            individual[me].recentlyproduced[myneed] +=
individual[me].need[myneed];
            individual[me].producedthisperiod[myneed] +=
individual[me].need[myneed];
```

```
            individual[me].need[myneed] = 0;
            if (me == TESTINDIVIDUAL) {
                printf("Producing mysmallneed %i periodrem %lli producednow
%lli\n",
                       myneed, (long long)individual[me].periodremaining,
individual[me].producedthisperiod[myneed]);
            }
        }
    }
}

/* surplusproduction(me, needtype): If agent `me` still has time remaining
after fulfilling needs and exchanges, use that time to produce surplus goods.
 * This simulates agents utilizing free time to produce goods for future
trade or profit.
 * - needtype: indicates which round we are in (REGULARROUND or SURPLUSROUND;
this influences how productionlimit is calculated for gifted goods).
 * The strategy:
 *   While the agent has at least 1 time unit remaining:
 *       - Determine which good to produce as surplus. We consider only goods
where the agent is gifted (TRUE) for this routine (skilled production).
 *       - For each gifted good (myneed):
 *             Calculate productionlimit = (soldxperiod + soldthisperiod) +
((MAXNEEDSINCREASE * needslevel * elasticneed) - surplus).
 *             (This formula tries to gauge how much could be sold plus how
much stock is needed if needs increase, effectively suggesting an amount that
might be useful to produce) 81 82 .
 *             If needtype == SURPLUSROUND, optionally adjust productionlimit
to (stocklimit - surplus) (to avoid producing beyond storage).
 *             If productionlimit > 1 (INTMULTIPLIER) and either:
 *                 - The agent hasn't purchased any of this good
(purchasetimes == 0) and already has a lot of surplus (meaning it's in
demand),
 *                 OR
 *                 - The agent's production cost (1/efficiency) is <= 1.05 *
salesprice (meaning producing more is profitable at current price),
 *               then compute a productionindex = efficiency - (1.0 /
salesprice) (profit margin indicator).
 *             Use productionindex to pick the good with the highest apparent
profit or demand.
 *       - After scanning all goods, if selectedproduction == 0, break (no
profitable surplus to produce).
 *       - Otherwise, produce as much as possible of selectedproduction:
 *             maxproduction = efficiency[selected] * periodremaining (max
units could produce with all remaining time).
 *             If maxproduction < productionlimitselectedproduction (i.e.,
not enough time to produce full desired amount):
 *                 add maxproduction to surplus, producedthisperiod,
recentlyproduced, and set periodremaining = 0 (use up all time).
 *             Else (enough time to produce the desired amount):
 *                 produce exactly productionlimitselectedproduction, add to
```

```
surplus and stats, then subtract time used = (productionlimit / efficiency)
plus a SWITCHTIME overhead (1 time unit overhead) from periodremaining [83] [84].
 *       - Loop continues to see if after producing one item fully, there's
still time to produce another item.
 * This encourages agents to produce the item that gives them the highest
return or which they can sell (based on efficiency and price).
 * (If no gifted goods have demand, this loop will break without using
remaining time; then leisureproduction may handle unskilled production.)
 */
void surplusproduction(int me, int needtype) {
    int selectedproduction;
    long long productionlimit;
    long long productionlimitselectedproduction = 0;
    long long maxproduction;
    float productionindex;
    float currentproductionindex; // (currently unused, could incorporate
efficiency into decision)
    long long maxtimeforsurplus;
    while (individual[me].periodremaining >= 1) {
        productionlimit = 0;
        productionindex = 0;
        selectedproduction = 0;
        maxtimeforsurplus = 0;
        // Consider each skill where agent is gifted (skilled production)
        for (int myneed = 1; myneed < SKILLS; myneed++) {
            if (individual[me].gifted[myneed] == TRUE) {
                // Compute how much of this good might be worth producing
                productionlimit = ((individual[me].soldxperiod[myneed] +
individual[me].soldthisperiod[myneed]) +
                                   (((MAXNEEDSINCREASE *
individual[me].needslevel) * market.elasticneed[myneed]) -
individual[me].surplus[myneed]));
                // If this is the surplus round, consider producing up to
stock limit (if we had earlier production in regular round)
                if (needtype == SURPLUSROUND) {
                    productionlimit = individual[me].stocklimit[myneed] -
individual[me].surplus[myneed];
                }
                if ((productionlimit > INTMULTIPLIER) && (
                     // If never purchased this good (all supply is from
self) and already have large surplus, or if production cost is sufficiently
low relative to price:
                     ((individual[me].purchasetimes[myneed] == 0) &&
individual[me].surplus[myneed] > (MAXSURPLUSFACTOR * market.basicneed[1])) ||
                     ((1.0f / individual[me].efficiency[myneed]) <=
(PRICEHIKE * individual[me].salesprice[myneed]))
                    )) {
                    // Calculate a heuristic production index (profit margin
= efficiency - (1/cost)) to decide priority [85] [86].
                    // (Note: The code below is double-checking the same
condition, effectively setting the same productionindex twice.)
```

```
                    // if (productionindex <=
individual[me].efficiency[myneed] - (1.0f /
individual[me].salesprice[myneed])) {
                    //     productionindex =
individual[me].efficiency[myneed] - (1.0f /
individual[me].salesprice[myneed]);
                    // }
                    if (productionindex <=
(individual[me].efficiency[myneed] - (1.0f /
individual[me].salesprice[myneed]))) {
                        productionindex = individual[me].efficiency[myneed]
- (1.0f / individual[me].salesprice[myneed]);
                        selectedproduction = myneed;
                        productionlimitselectedproduction = productionlimit;
                        // Debug print for production decision
                        // printf("productionindex %f myneed %i
productionlimit %lli\n", productionindex, myneed, productionlimit);
                    }
                }
            }
        }
        if (selectedproduction == 0) {
            // No beneficial surplus production found, break out
            // printf("selectedprodzero, but hello its me %i", me);
            break;
        }
        if (me == TESTINDIVIDUAL) {
            printf("Me %i needtype %i Surplusproducing %i,
productionlimitselectedproduction %lli surplus %lli efficiency %f
timeremaining %lli, stocklimit %lli\n",
                   me, needtype, selectedproduction,
productionlimitselectedproduction,
individual[me].surplus[selectedproduction],
                   individual[me].efficiency[selectedproduction], (long
long)individual[me].periodremaining,
individual[me].stocklimit[selectedproduction]);
        }
        // Determine how much can be produced with remaining time
        maxproduction = (long long)
(individual[me].efficiency[selectedproduction] *
individual[me].periodremaining);
        if (maxproduction < productionlimitselectedproduction) {
            // Not enough time to produce the desired amount; produce as much
as possible
            individual[me].surplus[selectedproduction] += maxproduction;
            individual[me].producedthisperiod[selectedproduction] +=
maxproduction;
            individual[me].recentlyproduced[selectedproduction] +=
maxproduction;
            individual[me].periodremaining = 0; // used all remaining time
            if (me == TESTINDIVIDUAL) printf("if\n");
```

```
        } else {
            // Enough time to produce the full
productionlimitselectedproduction
            maxtimeforsurplus = SWITCHTIME + (long long)
(productionlimitselectedproduction /
individual[me].efficiency[selectedproduction]);
            // We include a SWITCHTIME overhead (time to switch tasks, etc.)
            individual[me].surplus[selectedproduction] +=
productionlimitselectedproduction;
            individual[me].producedthisperiod[selectedproduction] +=
productionlimitselectedproduction;
            individual[me].recentlyproduced[selectedproduction] +=
productionlimitselectedproduction;
            individual[me].periodremaining -= maxtimeforsurplus;
            if (me == TESTINDIVIDUAL) printf("else\n");
        }
        if (individual[me].periodremaining < 0) printf("periodremaining
negative");
        if (me == TESTINDIVIDUAL) {
            printf("Produced surplus %i, efficiency %f timeremaining %lli,
stocklimit %lli, surplus %lli, productionlimselprod %lli\n",
                   selectedproduction,
individual[me].efficiency[selectedproduction], (long
long)individual[me].periodremaining,
                   individual[me].stocklimit[selectedproduction],
individual[me].surplus[selectedproduction],
productionlimitselectedproduction);
        }
    }
}

/* leisureproduction(me): If agent `me` still has leftover time after surplus
production of gifted goods, use remaining time to produce something even if
not gifted.
 * This covers the scenario where an agent has idle time and could produce
goods they're not specialized in, rather than wasting the time (simulating
that they use all available labor).
 * Strategy:
 *   While periodremaining >= 1:
 *       - Consider only skills where agent is NOT gifted (because gifted
ones presumably already produced if profitable).
 *       - For each such skill:
 *             productionlimit = stocklimit - surplus (how much more they can
store of this good).
 *             If productionlimit > 1 and (productionindex <=
purchaseprice[skill]):
 *                (Here productionindex is reused to choose the one with
highest purchaseprice, meaning the good the agent values the most to have.)
 *             Choose the good with the highest purchaseprice (the agent's
own value) to produce.
 *       - Produce as much as possible of that selected good:
```

```
 *             maxproduction = periodremaining (since efficiency for
unskilled goods is 1 unit per time by default).
 *             If maxproduction < productionlimit: produce maxproduction
units, add to surplus, use all time.
 *             Else: produce productionlimit units, subtract that many time
units (here effectively 1:1 since efficiency ~1).
 *       - Continue until no time or no production needed.
 * Essentially, in leisure time, the agent produces goods for which they have
capacity and which they consider valuable (high purchase price) even if not
particularly efficient at them [87] [88].
 */
void leisureproduction(int me) {
    int selectedproduction;
    long long productionlimit;
    long long productionlimitselectedproduction = 0;
    long long maxproduction;
    float productionindex;
    float currentproductionindex; // (could be used to include efficiency in
the decision)
    long long maxtimeforsurplus;
    while (individual[me].periodremaining >= 1) {
        productionlimit = 0;
        productionindex = 0;
        selectedproduction = 0;
        maxtimeforsurplus = 0;
        for (int myneed = 1; myneed < SKILLS; myneed++) {
            if (individual[me].gifted[myneed] == FALSE) {
                // Only consider goods where agent is not gifted (less
preferred normally)
                productionlimit = individual[me].stocklimit[myneed] -
individual[me].surplus[myneed];
                if ((productionlimit > INTMULTIPLIER) && (productionindex <=
individual[me].purchaseprice[myneed])) {
                    // Choose the good with the highest personal
purchaseprice (the one agent values most to have) to produce
                    productionindex = individual[me].purchaseprice[myneed];
                    selectedproduction = myneed;
                    productionlimitselectedproduction = productionlimit;
                }
            }
        }
        if (selectedproduction == 0) {
            // Nothing selected (either no capacity or no value), break
            // printf("selectedprodzero, but hello its me %i", me);
            break;
        }
        if (me == TESTINDIVIDUAL) {
            printf("Me %i Leisureproducing %i,
productionlimitselectedproduction %lli surplus %lli efficiency %f
timeremaining %lli, stocklimit %lli\n",
                   me, selectedproduction,
```

```
productionlimitselectedproduction,
individual[me].surplus[selectedproduction],
                    individual[me].efficiency[selectedproduction], (long
long)individual[me].periodremaining,
individual[me].stocklimit[selectedproduction]);
        }
        maxproduction = (long long)individual[me].periodremaining;
        if (maxproduction < productionlimitselectedproduction) {
            // Use all remaining time to produce that many units (efficiency
~1 for non-gifted typically)
            individual[me].surplus[selectedproduction] += maxproduction;
            individual[me].producedthisperiod[selectedproduction] +=
maxproduction;
            individual[me].recentlyproduced[selectedproduction] +=
maxproduction;
            individual[me].periodremaining = 0;
            if (me == TESTINDIVIDUAL) printf("zeroedleisuretime\n");
        } else {
            // Have more time than needed to hit stocklimit, produce up to
stocklimit
            maxtimeforsurplus = productionlimitselectedproduction;
            individual[me].surplus[selectedproduction] +=
productionlimitselectedproduction;
            individual[me].producedthisperiod[selectedproduction] +=
productionlimitselectedproduction;
            individual[me].recentlyproduced[selectedproduction] +=
productionlimitselectedproduction;
            individual[me].periodremaining -= maxtimeforsurplus;
            if (me == TESTINDIVIDUAL) printf("timeremaining\n");
        }
        if (individual[me].periodremaining < 0) printf("periodremaining
negative");
        if (me == TESTINDIVIDUAL) {
            printf("Leisureproduced surplus %i, efficiency %f timeremaining
%lli, stocklimit %lli, surplus %lli, productionlimselprod %lli\n",
                    selectedproduction,
individual[me].efficiency[selectedproduction], (long
long)individual[me].periodremaining,
                    individual[me].stocklimit[selectedproduction],
individual[me].surplus[selectedproduction],
productionlimitselectedproduction);
        }
    }
}

/* consumesurplus(me): Use any existing surplus to satisfy current needs
before producing or trading.
 * For each good j:
 *    If surplus[j] >= need[j]:
 *         subtract need[j] from surplus (consume from stock) and set need[j]
= 0.
```

```
 *    Else if surplus[j] > 0 but less than need:
 *         reduce need[j] by surplus[j] (consume all surplus) and set
surplus[j] = 0.
 *    Print a warning if surplus is astronomically high (close to LLONG_MAX)
which could indicate overflow (for debug).
 *    If any surplus goes negative (should not happen logically), set it to 0
(correct error).
 * This function ensures that at the start of a cycle, an agent first tries
to use what they already have in inventory to meet needs.
 * It represents consumption of stored goods.
 */
void consumesurplus(int me) {
    for (int j = 1; j < SKILLS; j++) {
        if (individual[me].surplus[j] > individual[me].need[j]) {
            individual[me].surplus[j] -= individual[me].need[j];
            individual[me].need[j] = 0;
        } else if (individual[me].surplus[j] > 0) {
            individual[me].need[j] -= individual[me].surplus[j];
            individual[me].surplus[j] = 0;
        }
        if (individual[me].surplus[j] > (LLONG_MAX / MAXGROUP)) {
            printf("Agent %i with surplus %i closes on llong_max \n", me, j);
        }
        if (individual[me].surplus[j] < 0) {
            printf("consumesurplus negative, set to zero");
            individual[me].surplus[j] = 0;
        }
    }
}

/* calibrateftransparency(me): Recalculate social transparency (trust) values
for agent `me`'s friendships at end of period.
 * Transparency increases with more transactions, and with consistent
purchasing from that friend, etc., up to a limit (1.0).
 * - Start each friend transparency at INITIALSOCIALTRANSPARENCY (0.7).
 * - If transactions > 0, increase transparency by 0.7*(transactions/
(transactions+SKILLS)) of the remaining gap to 1 [89].
 * - Also increase transparency by 0.7*((10*purchased[l])/(10*purchased[l]
+period)) of remaining gap (if agent has purchased a lot from friend relative
to simulation length) [90] [91].
 * - Also increase transparency by 0.7*(recentlypurchased[l]/
(recentlypurchased[l] + 10*HISTORY*INTMULTIPLIER)) of remaining gap (if high
recent volume purchased from friend) [92].
 * - If the friend is gifted in that good, add an extra 0.5 of remaining gap
(because a gifted friend is a reliable source) [93].
 * - Finally, if transactions > 1, decay transactions count by 0.9 (so old
transactions have diminishing effect) [94].
 * This formula means trust increases with each successful trade, more so if
a friend consistently supplies a particular good, and trust grows faster for
reliable producers.
 */
```

```
void calibrateftransparency(int me) {
    for (int k = 1; k < MAXGROUP; k++) {
        for (int l = 1; l < SKILLS; l++) {
            individual[me].relationship[k].transparency[l] =
INITIALSOCIALTRANSPARENCY;
            if (individual[me].relationship[k].transactions > 0) {
                individual[me].relationship[k].transparency[l] +=
                    (((1.0f -
individual[me].relationship[k].transparency[l]) * 0.7f) *
                     ((float)individual[me].relationship[k].transactions /
(float)(individual[me].relationship[k].transactions + SKILLS)));
            }
            individual[me].relationship[k].transparency[l] +=
                (((1.0f - individual[me].relationship[k].transparency[l]) *
0.7f) *
                 (float)((10 *
individual[me].relationship[k].purchased[l]) / (float)((10 *
individual[me].relationship[k].purchased[l]) + market.period)));
            individual[me].relationship[k].transparency[l] +=
                (((1.0f - individual[me].relationship[k].transparency[l]) *
0.7f) *
                 ((float)individual[me].recentlypurchased[l] /
                  (float)(individual[me].recentlypurchased[l] + ((10 *
HISTORY) * INTMULTIPLIER))));
            if (individual[me].gifted[l] == TRUE) {
                individual[me].relationship[k].transparency[l] +=
                    ((1.0f - individual[me].relationship[k].transparency[l])
* 0.5f);
            }
        }
        if (individual[me].relationship[k].transactions > 1) {
            individual[me].relationship[k].transactions = 0.9f *
individual[me].relationship[k].transactions;
        }
    }
}

/* adjustpurchaseprice(me, myneed): Adjust agent `me`'s purchase price
threshold for good `myneed` based on this period's outcomes (heuristics).
 * Heuristics:
 *   - If agent has less surplus than one period's need (surplus <
elasticneed) => scarcity:
 *         * If role is CONSUMER and they ended up consuming most of what
they produced (recentlypurchased < recentlyproduced + 1 unit),
 *                 set purchaseprice = production cost (they're willing to
pay at least what it costs them to make it) [21].
 *           Else if they made no purchases (purchasetimes==0) and their
current purchaseprice is below production cost,
 *                 increase purchaseprice by PRICELEAP (30% jump) to signal
they're willing to pay more next time [95].
 *         * If role is RETAILER and either they bought less than they sold
```

```
(indicating unmet demand) or less than half their stock,
 *                 increase purchaseprice by PRICEHIKE (5% up) to be more
competitive in acquiring that good [96].
 *         * If role is PRODUCER, no action in this case (producers usually
don't buy their own product).
 *   - If agent has moderate surplus (~ >= elasticneed but < stocklimit):
 *         * CONSUMER: no specific rule (likely no change).
 *         * RETAILER: if they have made multiple purchases (purchasetimes>1)
and purchasedthisperiod > half their stocklimit,
 *                 set purchaseprice to the average purchase value actually
paid this period (sumperiodpurchasevalue/purchasetimes) with some weighting
(including HISTORY as prior) [97] [98].
 *                 Then, if new purchaseprice > salesprice (paying more than
selling price), cap it at 0.95 * salesprice (don't overpay beyond a small
margin) [99].
 *         * PRODUCER: if they bought more than they produced (i.e. had to
buy to meet demand), reduce purchaseprice by 5% (PRICEREDUCTION) because they
overpaid and should lower willingness to pay next time [100].
 *   - If agent has large surplus (surplus > stocklimit + elasticneed,
meaning stock completely full):
 *         * CONSUMER: reduce purchaseprice by 5% (they value it less since
they have plenty) [101].
 *         * RETAILER: if they have ever bought this good (purchasetimes >
0), reduce purchaseprice by 5% [102].
 *         * PRODUCER: if ever bought, reduce purchaseprice by 5% [103].
 * After adjusting, if purchaseprice gets > 10 (arbitrary high threshold),
print a debug message.
 * These adjustments represent learning: if an agent had difficulty obtaining
a good, they'll pay more next time; if they have too much, they'll pay less.
 */
void adjustpurchaseprice(int me, int myneed) {
    float productioncost = 1.0f / individual[me].efficiency[myneed]; // cost
to produce one unit
    switch ((individual[me].surplus[myneed] > (market.elasticneed[myneed])) +
            (individual[me].surplus[myneed] >
(individual[me].stocklimit[myneed] + market.elasticneed[myneed]))) {
        case 0:  // Little surplus (scarcity scenario)
            switch (individual[me].role[myneed]) {
                case CONSUMER:
                    if (individual[me].recentlypurchased[myneed] <
individual[me].recentlyproduced[myneed] + INTMULTIPLIER) {
                        individual[me].purchaseprice[myneed] =
productioncost;
                    } else if ((individual[me].purchasetimes[myneed] == 0)
&& (individual[me].purchaseprice[myneed] < productioncost)) {
                        individual[me].purchaseprice[myneed] = PRICELEAP *
individual[me].purchaseprice[myneed];
                    }
                    break;
                case RETAILER:
                    if ((individual[me].purchasedthisperiod[myneed] <
```

```
individual[me].soldthisperiod[myneed] + INTMULTIPLIER) ||
                        (individual[me].purchasedthisperiod[myneed] <
individual[me].stocklimit[myneed] / HISTORY)) {
                        individual[me].purchaseprice[myneed] = PRICEHIKE *
individual[me].purchaseprice[myneed];
                    }
                    break;
                case PRODUCER:
                    // Producers rarely purchase their own product, no change
                    break;
                default:
                    printf("this is unused option in switch/case when
finetuning sales and purchase prices");
            }
            break;
        case 1:  // Average surplus (some extra on hand)
            switch (individual[me].role[myneed]) {
                case CONSUMER:
                    // no special adjustment in moderate surplus case for
consumers
                    break;
                case RETAILER:
                    if ((individual[me].purchasetimes[myneed] > 1) &&
                        (individual[me].purchasedthisperiod[myneed] >
(individual[me].stocklimit[myneed] / 2))) {
                        // Adjust purchaseprice to average purchase value
this period (blended with history) 104 105 .
                        individual[me].purchaseprice[myneed] =
                            ((individual[me].sumperiodpurchasevalue[myneed]
+ (HISTORY * individual[me].purchaseprice[myneed]))
                             / (float)(individual[me].purchasetimes[myneed]
+ HISTORY));
                        if (me == TESTINDIVIDUAL) {
                            printf("setting myneed %i  avpurchvalue %f
sumperpurchval %f sumperpurchtimes %i\n",
                                   myneed,
individual[me].purchaseprice[myneed],
individual[me].sumperiodpurchasevalue[myneed],
individual[me].purchasetimes[myneed]);
                        }
                    }
                    if (individual[me].purchaseprice[myneed] >
individual[me].salesprice[myneed]) {
                        individual[me].purchaseprice[myneed] =
(PRICEREDUCTION * individual[me].salesprice[myneed]);
                    }
                    break;
                case PRODUCER:
                    if (individual[me].purchasedthisperiod[myneed] >
individual[me].producedthisperiod[myneed]) {
                        individual[me].purchaseprice[myneed] =
```

```
PRICEREDUCTION * individual[me].purchaseprice[myneed];
                    }
                    break;
                default:
                    printf("this is unused option in switch/case when
finetuning sales and purchase prices");
            }
            break;
        case 2:  // Large surplus (stock completely full or more)
            switch (individual[me].role[myneed]) {
                case CONSUMER:
                    individual[me].purchaseprice[myneed] = PRICEREDUCTION *
individual[me].purchaseprice[myneed];
                    break;
                case RETAILER:
                    if (individual[me].purchasetimes[myneed]) {
                        individual[me].purchaseprice[myneed] =
PRICEREDUCTION * individual[me].purchaseprice[myneed];
                    }
                    break;
                case PRODUCER:
                    if (individual[me].purchasetimes[myneed]) {
                        individual[me].purchaseprice[myneed] =
PRICEREDUCTION * individual[me].purchaseprice[myneed];
                    }
                    break;
                default:
                    printf("this is unused option in switch/case when
finetuning sales and purchase prices");
            }
            break;
        default:
            printf("this is unused option in switch/case when finetuning
sales and purchase prices");
    }
    if (individual[me].purchaseprice[myneed] > 10) {
        printf("Role %i purchaseprice %f\n", individual[me].role[myneed],
individual[me].purchaseprice[myneed]);
    }
}

/* adjustsalesprice(me, myneed): Adjust agent `me`'s sales price threshold
for good `myneed` based on outcomes.
 * Heuristics:
 *   - If little surplus (scarcity):
 *         * CONSUMER: If current salesprice is below the smaller of
(productioncost, purchaseprice), raise it to that smaller value and then
apply PRICEHIKE (5% profit margin) [106].
 *   * (Meaning if they were selling cheap, at least charge what it cost or
what they'd pay themselves, then add a profit margin.)
 *         * RETAILER: If they had multiple sales (salestimes>1) and sold a
```

```
lot (>50% of stocklimit),
 *                   set salesprice to (sumperiodsalesvalue + current
salesprice) / (salestimes+1) (averaging this period's sales value) [107],
 *                   but cap any large jump: if new price > 130% of
previoussalesprice, cap at 1.3x previous (PRICELEAP) [108].
 *                   Ensure salesprice >= purchaseprice (never sell below
what they paid) [109].
 *                   If no one bought (salestimes==0), then set salesprice =
1.05 * purchaseprice (they might not have sold because price was too low?
Actually raising price here is counterintuitive, but the code does it to
ensure margin) [110].
 *         * PRODUCER: If salesprice < productioncost, raise to 1.05 *
productioncost (don't sell at a loss) [111].
 *    - If moderate surplus:
 *         * CONSUMER: set salesprice to max(productioncost, purchaseprice)
(ensures not selling below cost or what they'd pay) and if surplus > half
stock,
 *                     set salesprice to either double purchaseprice or
productioncost, whichever is lower (so if they have excess, they might raise
price but not above cost) [112] [113].
 *         * RETAILER: If soldthisperiod < elasticneed (meaning didn't sell
as much as typical demand), reduce salesprice by 5% [114].
 *         * PRODUCER: If soldthisperiod < elasticneed, reduce salesprice by
5% (they have trouble selling all, so lower price) [115].
 *                   Also if after that salesprice < productioncost, raise to
1.05 * productioncost (never go below cost) [116].
 *    - If large surplus (way too much stock unsold):
 *         * CONSUMER: reduce salesprice by 5% (willing to take lower price
to get rid of goods) [117].
 *         * RETAILER: If soldthisperiod < elasticneed (not much sold):
 *                         if salesprice > purchaseprice, lower to
purchaseprice (can't sell above what they paid) [118];
 *                         else reduce by 5% more.
 *         * PRODUCER: If salesprice > productioncost, set to 1.05 *
productioncost (cap price at a modest profit over cost) [119].
 *                     If soldthisperiod < elasticneed, also set to exactly
1.05 * productioncost (further ensure price is at cost level, they had
oversupply) [120].
 * After adjustments, these new salesprices influence future exchange indexes
and production incentives.
 * Essentially: if a good sold easily, price goes up; if not, price goes
down, but never below cost or what they'd pay themselves.
 */
void adjustsalesprice(int me, int myneed) {
    float productioncost = 1.0f / individual[me].efficiency[myneed];
    float previoussalesprice = individual[me].salesprice[myneed];
    switch ((individual[me].surplus[myneed] > market.elasticneed[myneed]) +
            (individual[me].surplus[myneed] >
(individual[me].stocklimit[myneed] + market.elasticneed[myneed]))) {
        case 0: // Little surplus (almost everything sold or used)
            switch (individual[me].role[myneed]) {
```

```
                case CONSUMER:
                    // Ensure price is at least min(productioncost,
purchaseprice) then add small profit
                    if (individual[me].salesprice[myneed] < (productioncost
> individual[me].purchaseprice[myneed] ?
individual[me].purchaseprice[myneed] : productioncost)) {
                        individual[me].salesprice[myneed] = (productioncost
> individual[me].purchaseprice[myneed] ?
individual[me].purchaseprice[myneed] : productioncost);
                        individual[me].salesprice[myneed] = PRICEHIKE *
individual[me].salesprice[myneed];
                    }
                    break;
                case RETAILER:
                    if ((individual[me].salestimes[myneed] > 1) &&
(individual[me].soldthisperiod[myneed] >
(individual[me].stocklimit[myneed]) / 2)) {
                        individual[me].salesprice[myneed] =
                            ((individual[me].sumperiodsalesvalue[myneed] +
individual[me].salesprice[myneed])
                             / (float)(individual[me].salestimes[myneed] +
1));
                        if (individual[me].salesprice[myneed] > (PRICELEAP *
previoussalesprice)) {
                            individual[me].salesprice[myneed] = (PRICELEAP *
previoussalesprice);
                        }
                    }
                    if (individual[me].salesprice[myneed] <
individual[me].purchaseprice[myneed]) {
                        individual[me].salesprice[myneed] =
individual[me].purchaseprice[myneed];
                    }
                    if (individual[me].salestimes[myneed] == 0) {
                        individual[me].salesprice[myneed] = PRICEHIKE *
individual[me].purchaseprice[myneed];
                    }
                    break;
                case PRODUCER:
                    if (individual[me].salesprice[myneed] < productioncost) {
                        individual[me].salesprice[myneed] = PRICEHIKE *
productioncost;
                    }
                    break;
                default:
                    printf("this is unused option in switch/case when
finetuning sales and purchase prices");
            }
            break;
        case 1: // Average surplus
            switch (individual[me].role[myneed]) {
```

```
                case CONSUMER:
                    // If moderate surplus, adjust to at least cost or what
they'd pay:
                    individual[me].salesprice[myneed] = (productioncost >
individual[me].purchaseprice[myneed] ? productioncost :
individual[me].purchaseprice[myneed]);
                    if (individual[me].surplus[myneed] >
(individual[me].stocklimit[myneed] / 2)) {
                        // if stockpile is half full, maybe raise price but
not above double what they'd pay or cost
                        individual[me].salesprice[myneed] =
(((individual[me].purchaseprice[myneed] * 2) < productioncost) ?

(individual[me].purchaseprice[myneed] * 2) : productioncost);
                    }
                    break;
                case RETAILER:
                    if (individual[me].soldthisperiod[myneed] <
(market.elasticneed[myneed])) {
                        individual[me].salesprice[myneed] =
individual[me].salesprice[myneed] * PRICEREDUCTION;
                    }
                    break;
                case PRODUCER:
                    if (individual[me].soldthisperiod[myneed] <
(market.elasticneed[myneed])) {
                        individual[me].salesprice[myneed] =
individual[me].salesprice[myneed] * PRICEREDUCTION;
                    }
                    if (individual[me].salesprice[myneed] < productioncost) {
                        individual[me].salesprice[myneed] = PRICEHIKE *
productioncost;
                    }
                    break;
                default:
                    printf("this is unused option in switch/case when
finetuning sales and purchase prices");
            }
            break;
        case 2: // Large surplus (oversupply, unsold goods)
            switch (individual[me].role[myneed]) {
                case CONSUMER:
                    individual[me].salesprice[myneed] =
individual[me].salesprice[myneed] * PRICEREDUCTION;
                    break;
                case RETAILER:
                    if (individual[me].soldthisperiod[myneed] <
(market.elasticneed[myneed])) {
                        if (individual[me].salesprice[myneed] >
individual[me].purchaseprice[myneed]) {
                            individual[me].salesprice[myneed] =
```

```
individual[me].purchaseprice[myneed];
                        } else {
                            individual[me].salesprice[myneed] =
individual[me].salesprice[myneed] * PRICEREDUCTION;
                        }
                    }
                    break;
                case PRODUCER:
                    if (individual[me].salesprice[myneed] > productioncost) {
                        individual[me].salesprice[myneed] = PRICEHIKE *
productioncost;
                    }
                    if (individual[me].soldthisperiod[myneed] <
(market.elasticneed[myneed])) {
                        individual[me].salesprice[myneed] = productioncost *
PRICEHIKE;
                        // If needed, could further reduce by PRICEREDUCTION,
but code commented out:
                        // individual[me].salesprice[xmyneed] =
individual[me].salesprice[myneed]*PRICEREDUCTION;
                    }
                    break;
                default:
                    printf("this is unused option in switch/case when
finetuning sales and purchase prices");
            }
            break;
        default:
            printf("this is unused option in switch/case when finetuning
sales and purchase prices");
    }
}

/* evaluatestock(me): Evaluate if agent `me`'s current surplus/wealth can
sustain their needslevel.
 * It computes a "stocklevel" ratio = (totalneedsvalue + wealthminusneeds) /
totalneedsvalue.
 * - wealthminusneeds starts from market.periodlength (a baseline
representing one cycle of needs time).
 * - For each need:
 *       surplusvalue = surplus - (elasticneed * needslevel *
MAXNEEDSINCREASE).
 *         If surplusvalue is negative (meaning agent lacks some amount):
 *            if total historically purchased > the shortfall, then assume
they'd buy that shortfall: surplusvalue *= purchaseprice (cost of buying the
lacking part) 121 ;
 *            else assume they'd have to produce it: surplusvalue /=
efficiency (time cost to produce shortfall).
 *         If surplusvalue is positive (excess surplus beyond 1.5x needs):
 *            if agent was unable to sell some of it (recentlysold >
surplusvalue):
```

```
 *                  cap surplusvalue to MAXSURPLUSFACTOR * (soldthisperiod -
soldxperiod + current need) (meaning only count the portion they actually
could have sold) [122],
 *                  then surplusvalue *= salesprice (value of that surplus if
sold).
 *            else (they sold everything they could):
 *                  if surplusvalue >
(elasticneed*needslevel*MAXNEEDSINCREASE) cap it at that (don't count beyond
1.5x needs as wealth) [123],
 *                  then surplusvalue *= min(purchaseprice, 1/efficiency)
(value the surplus either at what they'd pay or cost to produce) [124].
 *         Add surplusvalue to wealthminusneeds.
 *         Compute totalneedsvalue += (elasticneed * needslevel) valued at
cost: if recentlypurchased > need, use purchaseprice*need (they did buy more
than needed, so price relevant) else use need/efficiency (time to produce
need) [125].
 * - Finally stocklevel = (totalneedsvalue + wealthminusneeds) /
totalneedsvalue.
 * If stocklevel >= 1, the agent's current resources cover their needs (or
more); if < 1, they're in deficit.
 * This metric is used in evolution() to adjust needslevel: if stocklevel is
low, reduce needslevel (they can't sustain current consumption), if high,
allow needs to increase [23] [24].
 */
float evaluatestock(int me) {
    float stocklevel = 1.0f;
    double wealthminusneeds = market.periodlength;
    double surplusvalue = 0.0f;
    double totalneedsvalue = 0.0f;
    for (int need = 1; need < SKILLS; need++) {
        surplusvalue = individual[me].surplus[need] -
((market.elasticneed[need] * individual[me].needslevel) * MAXNEEDSINCREASE);
        if (surplusvalue < 0) {
            // Not enough surplus to cover 1.5x need:
            if (individual[me].purchasedxperiod[need] > -1 * surplusvalue) {
                // If historically they have purchased enough to cover this
shortfall, value it at purchase price (they would buy it)
                surplusvalue = surplusvalue *
individual[me].purchaseprice[need];
            } else {
                // Otherwise, value it in time (they'd have to produce the
shortfall)
                surplusvalue = surplusvalue /
individual[me].efficiency[need];
            }
        } else {
            // surplus beyond 1.5x need
            if (individual[me].recentlysold[need] > surplusvalue) {
                // If unable to sell all surplus (they had leftover unsold):
                if (surplusvalue > MAXSURPLUSFACTOR *
((individual[me].soldthisperiod[need] - individual[me].soldxperiod[need]) +
```

```
(market.elasticneed[need] * individual[me].needslevel))) {
                    surplusvalue = MAXSURPLUSFACTOR *
((individual[me].soldthisperiod[need] - individual[me].soldxperiod[need]) +
(market.elasticneed[need] * individual[me].needslevel));
                }
                surplusvalue = surplusvalue *
individual[me].salesprice[need];
            } else {
                if (surplusvalue > ((market.elasticneed[need] *
individual[me].needslevel) * MAXNEEDSINCREASE)) {
                    surplusvalue = (market.elasticneed[need] *
individual[me].needslevel) * MAXNEEDSINCREASE;
                }
                surplusvalue = surplusvalue *
(individual[me].purchaseprice[need] < (1.0f /
individual[me].efficiency[need]) ?

individual[me].purchaseprice[need] : (1.0f /
individual[me].efficiency[need]));
            }
        }
        wealthminusneeds += surplusvalue;
        if (individual[me].recentlypurchased[need] >
(market.elasticneed[need] * individual[me].needslevel)) {
            totalneedsvalue += (individual[me].purchaseprice[need] *
(market.elasticneed[need] * individual[me].needslevel));
        } else {
            totalneedsvalue += ((market.elasticneed[need] *
individual[me].needslevel) / individual[me].efficiency[need]);
        }
    }
    stocklevel = (totalneedsvalue + wealthminusneeds) / totalneedsvalue;
    return stocklevel;
}

/* endmyperiod(me): Finalize agent `me`'s state at the end of a period
(cycle).
 * This function updates dynamic properties in preparation for the next
period:
 *   - (Code for extra spending if lots of time left is commented out;
originally would increase needs if too much free time.)
 *   - Reset individual[i].periodicspoils = 0 (will accumulate spoils in this
period).
 *   - For each good:
 *        (Optionally adjust need for next period to elasticneed if
BASICROUNDELASTIC is true; in current setting BASICROUNDELASTIC=1, meaning
next period's base need will be elasticneed anyway via evolution).
 *        Calculate new stocklimit:
 *           stocklimit = MAXSURPLUSFACTOR * (elasticneed * needslevel) +
recentlysold (so stock capacity grows with needs and recent sales) 17 126 .
 *           Limit how sharply stocklimit can change: not below 95% of
```

```
previous, not above 120% of previous (smooth out changes) [18].
 *            Update previousstocklimit = stocklimit.
 *         If gifted:
 *            Recalculate efficiency (learning):
 *               new efficiency = 1 / InvSqrt((recentlyproduced + 1) /
(HISTORY * basicneed)) [127].
 *               This formula increases efficiency if recent production >
historical average.
 *               Cap efficiency changes: not below 0.98 * old
(MAXEFFICIENCYDOWNGRADE), not above 1.05 * old (MAXEFFICIENCYUPGRADE) [128].
 *               Also ensure gifted efficiency doesn't fall below
GIFTEDEFFICIENCYMINIMUM (they retain some advantage) [129].
 *         If not gifted and is TESTINDIVIDUAL and sold more than need
(meaning they traded a lot of a product they don't have talent in), print a
debug line (note: just debug).
 *         Call checksurplus(11) (mostly a debugging function to catch weird
surplus, with code commented out).
 *         Update role:
 *            default role = CONSUMER.
 *            If (recentlyproduced > recentlypurchased) AND (recentlysold >
(recentlyproduced+recentlypurchased)/2), set role = PRODUCER (they produce
most of what they get and sell most of it) [15].
 *            Else if (recentlyproduced < recentlypurchased) AND
(recentlysold > (recentlyproduced+recentlypurchased)/2), set role = RETAILER
(they buy more than produce and still sell most of it) [16].
 *         adjustpurchaseprice(me, myneed) and adjustsalesprice(me, myneed)
for this good (update personal price thresholds using the heuristics).
 *         If TESTINDIVIDUAL, print a summary line for this need (role,
surplus, stocklimit, efficiency, produced, purchased, purchase times, average
purchase value, purchase price, sold, sales price) [130] [131].
 *         Spoilage: If surplus > MAXSURPLUSFACTOR * elasticneed (more than
2x period need), and surplus > STOCKSPOILTRESHOLD * stocklimit (here
threshold 2.0):
 *               If surplus > stocklimit (beyond capacity), spoil
SPOILSURPLUSEXCESS (10%) of the excess (surplus - stocklimit) [132] [133].
 *               Subtract spoils from surplus, add to
individual.periodicspoils and market.periodicspoils.
 *               Print warning if calculation leads to negative values
(overflow checks).
 *         Discount older stats:
 *               recentlyproduced = recentlyproduced * DISCONT (keep 80%),
similarly for recentlysold and recentlypurchased (memory of past decays by
20%) [134].
 *         checksurplus(12) call (debug).
 *         Reset period counters for next cycle:
 *               purchasetimes, salestimes = 0; sumperiodpurchasevalue,
sumperiodsalesvalue = 0 (we have updated prices now) [135].
 *               producedxperiod = producedthisperiod; soldxperiod =
soldthisperiod; purchasedxperiod = purchasedthisperiod (store this period's
realized values for next period comparisons) [136].
 *   - After updating all goods, call calibrateftransparency(me) to update
```

```
trust in social links based on this period's transactions.
 * This function essentially wraps up the period: updating capacities,
learning (efficiency), roles, and adjusting price expectations and trust,
preparing the agent for the next cycle.
 */
void endmyperiod(int me) {
    // (Optional extra spending round code is commented out; was intended to
simulate additional consumption if a lot of time remains, but not used here.)
    // individual[me].recentneedsincrement = (individual[me].needsincrement +
(HISTORY * individual[me].recentneedsincrement)) / (float)(HISTORY + 1);
    // individual[me].needsincrement = 0;
    // long long currentneedsincrement = 0;
    // if (individual[me].periodremaining > market.leisuretime) {
    //     individual[me].needsincrement = -1.0f + ((double)
(market.periodlength) / (double)(market.periodlength + market.leisuretime -
individual[me].periodremaining));
    //     if (individual[me].needsincrement >
((individual[me].recentneedsincrement + 1.0f) * MAXNEEDSINCREASE) - 1.0f)
    //         individual[me].needsincrement =
((individual[me].recentneedsincrement + 1.0f) * MAXNEEDSINCREASE) - 1.0f;
    //     if (me == TESTINDIVIDUAL) printf("\n RERUN!! needs %f, recently %f
periodremaining %lli \n", (1.0f + individual[me].needsincrement), (1.0f +
individual[me].recentneedsincrement), individual[me].periodremaining);
    //     float needsbuffer = individual[me].needsincrement;
    //     long long originalperiodremaining =
individual[me].periodremaining;
    //     while (needsbuffer > 0.1f) {
    //         needsbuffer = needsbuffer * SPENDINGFROMEXCESS;
    //         for (int myneed = 1; myneed < SKILLS; myneed++) {
    //             individual[me].need[myneed] = (long long)
(market.elasticneed[myneed] * needsbuffer);
    //         }
    //         consumesurplus(me);
    //         makesurplusdeals(me);
    //         producesmallneeds(me);
    //         satisfyneedsbyexchange(me);
    //         produceneed(me);
    //         if (individual[me].periodremaining < (originalperiodremaining
* SPENDINGFROMEXCESS)) needsbuffer = 0;
    //         if (me == TESTINDIVIDUAL) printf("periodremaining %lli
needsbuffer %.3f originalperiodsrem %lli SPENDINGFRX %f\n",
individual[me].periodremaining, needsbuffer, originalperiodremaining,
SPENDINGFROMEXCESS);
    //     }
    //     surplusproduction(me, SURPLUSROUND);
    //     checksurplus(10);
    // }
    // individual[me].periodremainingdebt =
(individual[me].periodremainingdebt + individual[me].periodremaining);
    // if (individual[me].periodremainingdebt > 0)
    //     individual[me].periodremainingdebt = 0;
```

```
    individual[me].periodicspoils = 0;
    for (int myneed = 1; myneed < SKILLS; myneed++) {
        // Determine next period's base need - in practice, evolution() will
set needs to elasticneed * needslevel.
        // Update stock limits:
        individual[me].stocklimit[myneed] =
            ((MAXSURPLUSFACTOR * (market.elasticneed[myneed] *
individual[me].needslevel)) + individual[me].recentlysold[myneed]);
        if (individual[me].stocklimit[myneed] < (MAXSTOCKLIMITDECREASE *
individual[me].previousstocklimit[myneed])) {
            individual[me].stocklimit[myneed] = (MAXSTOCKLIMITDECREASE *
individual[me].previousstocklimit[myneed]);
        }
        if (individual[me].stocklimit[myneed] > (MAXSTOCKLIMITINCREASE *
individual[me].previousstocklimit[myneed])) {
            individual[me].stocklimit[myneed] = (MAXSTOCKLIMITINCREASE *
individual[me].previousstocklimit[myneed]);
        }
        individual[me].previousstocklimit[myneed] =
individual[me].stocklimit[myneed];
        if (individual[me].gifted[myneed] == TRUE) {
            float previousefficiency = individual[me].efficiency[myneed];
            // Learning: increase efficiency if producing more than history
average (recentlyproduced vs HISTORY*basicneed) 127
            individual[me].efficiency[myneed] = (1.0f / InvSqrt((float)
(individual[me].recentlyproduced[myneed] + 1) /
                                                                (float)
(HISTORY * market.basicneed[myneed])));
            if (individual[me].efficiency[myneed] < (previousefficiency *
MAXEFFICIENCYDOWNGRADE)) {
                individual[me].efficiency[myneed] = (previousefficiency *
MAXEFFICIENCYDOWNGRADE);
            } else if (individual[me].efficiency[myneed] >
(previousefficiency * MAXEFFICIENCYUPGRADE)) {
                individual[me].efficiency[myneed] = (previousefficiency *
MAXEFFICIENCYUPGRADE);
            }
            if (individual[me].efficiency[myneed] < GIFTEDEFFICIENCYMINIMUM)
{
                individual[me].efficiency[myneed] = GIFTEDEFFICIENCYMINIMUM;
            }
        } else if ((me == TESTINDIVIDUAL) &&
(individual[me].recentlysold[myneed] > (market.elasticneed[myneed]))) {
            printf("Nongifted trading  --- myneed %i rsales %lli salesprice
%f rpurchases %lli purchaseprice %f stocklimit %lli surplus %lli\n",
                   myneed, individual[me].recentlysold[myneed],
individual[me].salesprice[myneed], individual[me].recentlypurchased[myneed],
                   individual[me].purchaseprice[myneed],
individual[me].stocklimit[myneed], individual[me].surplus[myneed]);
        }
        checksurplus(11);
```

```
        // Determine new role based on production and trade balance:
        individual[me].role[myneed] = CONSUMER;
        if ((individual[me].recentlyproduced[myneed] >
individual[me].recentlypurchased[myneed]) &&
            ((individual[me].recentlysold[myneed] >
(individual[me].recentlyproduced[myneed] +
individual[me].recentlypurchased[myneed]) / 2))) {
            individual[me].role[myneed] =
PRODUCER; // Over half of what they get is produced and sold (net producer)
 15
        } else if ((individual[me].recentlyproduced[myneed] <
individual[me].recentlypurchased[myneed]) &&
                   ((individual[me].recentlysold[myneed] >
(individual[me].recentlyproduced[myneed] +
individual[me].recentlypurchased[myneed]) / 2))) {
            individual[me].role[myneed] =
RETAILER; // Over half of what they get is sold but majority was acquired
from others (net intermediary) 16
        }
        adjustpurchaseprice(me, myneed);
        adjustsalesprice(me, myneed);
        if (me == TESTINDIVIDUAL) {
            printf("Need %i, role %i srplus %6lli, stcklmt %6lli, eff %3.1f,
rprdcd %7lli prod %7lli rpurch %5lli purch %5lli purchtimes %i avpurchval %.
3f p$ %.2f rsls %5lli sold %5lli sp$ %.2f\n",
                   myneed, individual[TESTINDIVIDUAL].role[myneed],
individual[TESTINDIVIDUAL].surplus[myneed],
individual[TESTINDIVIDUAL].stocklimit[myneed],
individual[TESTINDIVIDUAL].efficiency[myneed],
                   individual[TESTINDIVIDUAL].recentlyproduced[myneed],
individual[TESTINDIVIDUAL].producedthisperiod[myneed],
                   individual[TESTINDIVIDUAL].recentlypurchased[myneed],
individual[TESTINDIVIDUAL].purchasedthisperiod[myneed],
individual[TESTINDIVIDUAL].purchasetimes[myneed],
                   (individual[TESTINDIVIDUAL].purchasetimes[myneed] ?
(float)(individual[TESTINDIVIDUAL].sumperiodpurchasevalue[myneed] /
individual[TESTINDIVIDUAL].purchasetimes[myneed]) : 0.0f),
                   individual[TESTINDIVIDUAL].purchaseprice[myneed],
                   individual[TESTINDIVIDUAL].recentlysold[myneed],
individual[TESTINDIVIDUAL].soldthisperiod[myneed],
                   individual[TESTINDIVIDUAL].salesprice[myneed]);
        }
        // Spoilage: reduce surplus if exceeding storage capacity
        individual[me].spoils[myneed] = 0;
        if (individual[me].surplus[myneed] > (MAXSURPLUSFACTOR *
market.elasticneed[myneed])) {
            if (individual[me].surplus[myneed] > (STOCKSPOILTRESHOLD *
individual[me].stocklimit[myneed])) {
                if (individual[me].surplus[myneed] >
(individual[me].stocklimit[myneed])) {
                    individual[me].spoils[myneed] = (long long)
```

```
((individual[me].surplus[myneed] - individual[me].stocklimit[myneed]) *
SPOILSURPLUSEXCESS);
                    individual[me].surplus[myneed] -=
individual[me].spoils[myneed];
                    individual[me].periodicspoils +=
individual[me].spoils[myneed];
                    market.periodicspoils[myneed] +=
individual[me].spoils[myneed];
                    if ((individual[me].surplus[myneed] <
individual[me].spoils[myneed]) || (individual[me].spoils[myneed] < 0)) {
                        printf("Spoils-calculation overflow need %i, surplus
%lli spoils %lli elasticneed %lli, recentlysold %lli???",
                               myneed, individual[me].surplus[myneed],
individual[me].spoils[myneed], (long long)market.elasticneed[myneed],
individual[me].recentlysold[myneed]);
                    }
                }
                if (individual[me].surplus[myneed] < 0) {
                    printf("surplus negative, need %i surplus %lli", myneed,
individual[me].surplus[myneed]);
                }
                if (individual[me].periodicspoils < 0) {
                    printf("periodicspoils negative, indy %i  periodic %lli
need %i spoils %lli\n", me, individual[me].periodicspoils, myneed,
individual[me].spoils[myneed]);
                }
            }
        }
        // Discount old data to emphasize recent events in next cycle
        individual[me].recentlyproduced[myneed] = (long long)(DISCONT *
(double)individual[me].recentlyproduced[myneed]);
        individual[me].recentlysold[myneed] = (long long)(DISCONT *
(double)individual[me].recentlysold[myneed]);
        individual[me].recentlypurchased[myneed] = (long long)(DISCONT *
(double)individual[me].recentlypurchased[myneed]);
        checksurplus(12);
        // Reset counters for new period
        individual[me].purchasetimes[myneed] = 0;
        individual[me].salestimes[myneed] = 0;
        individual[me].sumperiodpurchasevalue[myneed] = 0;
        individual[me].sumperiodsalesvalue[myneed] = 0;
        individual[me].producedxperiod[myneed] =
individual[me].producedthisperiod[myneed];
        individual[me].soldxperiod[myneed] =
individual[me].soldthisperiod[myneed];
        individual[me].purchasedxperiod[myneed] =
individual[me].purchasedthisperiod[myneed];
    }
    calibrateftransparency(me);
}
```

```
// (resetperiodicstat function is commented out, as its functionality is
integrated in main loop and above resets)
/*
void resetperiodicstat(void) {
    for (int i=1; i<POPULATION; i++) {
        for (int need=1; need < SKILLS; need++) {
            individual[i].producedthisperiod[need] = 0;
            individual[i].soldthisperiod[need] = 0;
            individual[i].purchasedthisperiod[need] = 0;
        }
    }
    for (int need=1; need < SKILLS; need++) {
        market.periodictcecost[need] = 0;
        market.periodicspoils[need] = 0;
    }
    for (int i=1; i<SKILLS; i++) {
    }
}
*/

/* evolution(): The main per-period update routine for all agents.
 * This function is called once per cycle (period) after resetting global
counters.
 * It loops through every agent (i from 1 to POPULATION-1) and performs:
 *    - Check for extreme periodremainingdebt or periodremaining values and
log if overflow (if an agent accumulated huge time debt or surplus, set as
testindividual to inspect).
 *    - Reset each agent's periodremaining to full periodlength (new cycle
time budget).
 *    - Reset per-period production counters:
 *        producedthisperiod set to 0 for all goods,
 *        soldthisperiod adjusted: subtract soldxperiod (so soldthisperiod
now represents only new sales this cycle, as old values are carried from
endmyperiod) [137],
 *        purchasedthisperiod adjusted similarly.
 *    - Compute stocklevel = evaluatestock(i) to see if agent could sustain
or not [138].
 *    - Adjust needslevel:
 *        If stocklevel < MAXNEEDSREDUCTION (0.7) OR agent had periodfailure
flag, then reduce needslevel by factor 0.7 (drastically cut consumption if
they couldn't sustain) [138].
 *        Else if periodremainingdebt > -(1 - MAXNEEDSINCREASE)*periodlength
(i.e. they had leftover time, indicating easy period) AND stocklevel >
MAXNEEDSINCREASE (1.5), then increase needslevel by factor 1.5 (allow jump if
they clearly can handle more) [139].
 *        Else if stocklevel > SMALLNEEDSINCREASE (1.05), increase
needslevel by 1.05 (small upward adjustment for slight surplus).
 *        Else, if none of above, reduce needslevel by 0.95
(SMALLNEEDSREDUCTION) to gradually lower if just below break-even) [140].
 *    - Print debug info for TESTINDIVIDUAL on stocklevel and needslevel.
 *    - Ensure needslevel doesn't drop below 1 (baseline minimum).
```

```
 *    - (Older approach commented out: loop reduce needslevel until
satisfyneedsbyexchange succeeds, which has been replaced by above direct
adjustments.)
 *    - If periodremainingdebt < -periodlength (agent has more than one
period of debt), then reset needslevel = 1 and log that agent "starts from
scratch" (they failed so badly we reset them) [141].
 *    - Set needs for next period:
 *         for each need: if needslevel >= 1.0 (which it always is due to min
clamp):
 *                            need = elasticneed * needslevel (scale base
needs by current needslevel) [142].
 *                        else (if needslevel < 1, which we prevent, there's
commented alternative to use basicneed).
 *    - Now the main cycle for this agent:
 *         consumesurplus(i) (first use stored surplus to meet needs),
 *         satisfyneedsbyexchange(i) (attempt barter exchanges for remaining
needs),
 *         produceneed(i) (produce any remaining needs with time),
 *         if periodremaining < 0, set periodfailure=1 else 0 (note if they
ran out of time) [143].
 *         makenewfriends(i, FALSE) (at end of cycle, attempt to form one new
random friendship),
 *         If periodremainingdebt < 0, allow partial recovery: add half of
negative periodremaining to periodremaining and half to periodremainingdebt
(carry half the overtime to next period) [144].
 *         surplusproduction(i, SURPLUSROUND) (use remaining time for surplus
production in gifted skills),
 *         makesurplusdeals(i) (try to trade for more surplus if possible),
 *         Update periodremainingdebt = periodremaining + periodremainingdebt
(carry leftover or debt forward fully),
 *         If periodremainingdebt > 0, reset it to 0 (no carry if we somehow
have positive leftover, though typically periodremaining would be 0 or debt).
 *         leisureproduction(i) (finally, use any remaining time to produce
ungifted goods if possible).
 *         endmyperiod(i) (finalize this agent's period: update prices,
roles, transparency, etc.).
 * In summary, evolution() orchestrates the actions each agent takes within a
single cycle in order: consumption from surplus, exchanges, production,
friendship updates, and surplus usage, then calls end-of-period adjustments.
 */
void evolution(void) {
    // printf("evolution starting\n");
    for (int i = 1; i < POPULATION; i++) {
        // If an agent has extreme leftover or debt, log and track them
        if ((individual[i].periodremainingdebt < -2 * market.periodlength)
|| (individual[i].periodremaining > (2 * market.periodlength))) {
            printf("periodremaining overflow, me %i, needslevel %.3f periods
%lli, debt %lli \n",
                   i, individual[i].needslevel,
individual[i].periodremaining, individual[i].periodremainingdebt);
            market.testindividual = i;
```

```
        }
        // Reset time for new period
        individual[i].periodremaining = market.periodlength;
        // Reset period counters (some already handled in endmyperiod, but
ensure production is zeroed)
        for (int need = 1; need < SKILLS; need++) {
            individual[i].producedthisperiod[need] = 0;
            individual[i].soldthisperiod[need] =
individual[i].soldthisperiod[need] - individual[i].soldxperiod[need];
            individual[i].purchasedthisperiod[need] =
individual[i].purchasedthisperiod[need] -
individual[i].purchasedxperiod[need];
            // (Note: producedthisperiod was reset in endmyperiod,
soldthisperiod and purchasedthisperiod now hold only carryover differences if
any.)
        }
        float stocklevel = 0.0f;
        stocklevel = evaluatestock(i);
        if ((stocklevel < MAXNEEDSREDUCTION) || individual[i].periodfailure)
{
            individual[i].needslevel = individual[i].needslevel *
MAXNEEDSREDUCTION; // major reduction if they couldn't sustain
        } else if ((individual[i].periodremainingdebt > ((1.0f -
MAXNEEDSINCREASE) * market.periodlength)) && (stocklevel >
MAXNEEDSINCREASE)) {
            individual[i].needslevel = MAXNEEDSINCREASE *
individual[i].needslevel;   // big increase if they had surplus time and
plenty resources
        } else if (stocklevel > SMALLNEEDSINCREASE) {
            individual[i].needslevel = individual[i].needslevel *
SMALLNEEDSINCREASE; // slight increase if doing well
        } else {
            individual[i].needslevel = individual[i].needslevel *
SMALLNEEDSREDUCTION; // slight decrease if struggling a bit
        }
        if (i == TESTINDIVIDUAL) {
            printf("Kilroy %i was here and stocklevel is %.3f, needslevel is
%f, periodleft is %lli and debt is %lli!\n",
                   i, stocklevel, individual[i].needslevel, (long
long)individual[i].periodremaining, (long
long)individual[i].periodremainingdebt);
        }
        if (individual[i].needslevel < 1) individual[i].needslevel = 1;
        // (Old loop that repeatedly tried exchange until needslevel
sufficiently reduced is commented out for performance and replaced by above
adjustments.)
        // while (individual[i].needslevel > 0.99) {
        //     if (i==TESTINDIVIDUAL) printf("Kilroy %i was here and
stocklevel is %.3f!", i, stocklevel);
        //     if (satisfyneedsbyexchange(i)) {
        //         break;
```

```
        //     }
        //     individual[i].needslevel = individual[i].needslevel *
NEEDSREDUCTION;
        // }
        // if (individual[i].needslevel < 1) individual[i].needslevel = 1;
        if (individual[i].periodremainingdebt < (-1 * market.periodlength)) {
            // If agent fell an entire period behind in time, reset them
            individual[i].needslevel = 1.0f;
            printf("Kilroy %i starts from scratch, ends did not meet -
needslevel dropped to 1, part of debts collected each round!\n", i);
        }
        // Set initial needs for this period, scaled by current needslevel
(this is each agent's demand for the cycle)
        for (int need = 1; need < SKILLS; need++) {
            if (individual[i].needslevel >= SMALLNEEDSINCREASE) {
                individual[i].need[need] = market.elasticneed[need] *
individual[i].needslevel;
            } else {
                individual[i].need[need] = market.elasticneed[need];
            }
        }
        // Execute the cycle steps for agent i:
        consumesurplus(i);                  // 1. consume from own surplus
        satisfyneedsbyexchange(i);          // 2. barter exchange to satisfy
needs
        produceneed(i);                     // 3. produce remaining needs
        if (individual[i].periodremaining < 0) {
            individual[i].periodfailure = 1;
        } else {
            individual[i].periodfailure = 0;
        }
        makenewfriends(i, FALSE);           // 4. potentially form a new
random friend connection each period
        if (individual[i].periodremainingdebt < 0) {
            // If they had overtime (negative remaining time last period),
give back half of that time if possible
            individual[i].periodremaining +=
(individual[i].periodremainingdebt / 2);
            individual[i].periodremainingdebt =
(individual[i].periodremainingdebt / 2);
        }
        surplusproduction(i, SURPLUSROUND); // 5. use remaining time to
produce surplus goods (gifted skills)
        makesurplusdeals(i);                // 6. trade surplus if possible
for other goods
        individual[i].periodremainingdebt = individual[i].periodremaining +
individual[i].periodremainingdebt;
        if (individual[i].periodremainingdebt > 0) {
            individual[i].periodremainingdebt = 0; // ensure no positive
carry (reset if somehow we ended with extra time)
        }
```

```
        // Optionally do leisure production (non-gifted) after all trades
(this was commented out in original, but we include it to utilize any
leftover time)
        // printf("starting leisureproduction");
        leisureproduction(i);
        // finalize period for this agent (update stats, prices, roles, etc.)
        endmyperiod(i);
    }
}

/* main: The main function runs the simulation loop and prints results.
 * Steps:
 *   - Open output file "adamsmith.txt" for writing.
 *   - Print CSV header line for period results.
 *   - Print initial simulation parameters (population, group size, skills,
transparency, intmultiplier).
 *   - Initialize market, population, gifted distribution.
 *   - Initialize totalmisurplus to 0 (market.totalmisurplus tracks last
period's total surplus).
 *   - For each agent, initialize producedxperiod to basicneed (they produced
that much historically), soldxperiod and purchasedxperiod to 0 (no prior
trade).
 *   - Identify and print which goods the test individual is gifted in (for
debug).
 *   - Loop period from 1 to MAXPERIODS:
 *         * (Optionally call resetperiodicstat() - commented out, as we do
resets manually in evolution and evaluatemarketprices).
 *         * Reset totalsurplus for each agent to 0 (to accumulate fresh).
 *         * Reset market.periodictcecost and periodicspoils arrays to 0.
 *         * Set market.period = current period number.
 *         * Call evolution() to update all agents for this period.
 *         * Call evaluatemarketprices() to update global prices and print
utility statistics (this prints per-need line outputs each cycle).
 *         * Compute totalproduction and totalperiodictcecost for this period
by summing over all goods.
 *         * Decay market.numberofrecentlyproduced by DISCONT (older
production memory fades).
 *         * Print detailed stats line for TESTINDIVIDUAL (Need, role,
surplus, etc.) after every cycle 145 146 .
 *         * Check for overflow warnings for totals.
 *         * Sum up some overall metrics: totalisurplus (sum of all agents'
totalsurplus), sumremainingperiods, sumneeds, sumrecentneeds, sumofspoils,
sumovertimes, sumpurchaseprice1.
 *         * Count loosers (agents with large negative periodremainingdebt)
and reset everyone's timeout for next cycle.
 *         * Compute recentneedsdeviation (average absolute deviation of
recentneedsincrement from the mean).
 *         * Compute purchaseprice1average and its average deviation across
population.
 *         * Print summary statistics line (period, average free time,
average needs, spoils/time, tcecost/time, stored/time, production/time,
```

```
production balance, loosers count, needs deviation, deviation/avg increment,
a metric of money emergent value, average overtime, transaction cost per
time, spoil cost per time) [147] [148].
 *         * Write similar data to file.
 *         * Update market.totalmisurplus = totalisurplus (store total
surplus for next period's change calculation).
 *         * If mode != 'O', wait for user input (getchar) to proceed (this
allows stepping through periods if desired).
 *   - After loop, close file and exit.
 */
int main(int argc, char *argv[]) {
    FILE *f = fopen(FILENAME, "w");
    fprintf(f, "period, utility, spoils, tcecost, stored, production,
needsdeviation, dev/aveinc, prod1maxIndy, overtime, pprice1, pp1dev,
sumprod1, tcepertime, spoilspertime, maxef1*maxef1, totneed1inc\n");
    fprintf(f, "AdamSmith - population %i group %i skills %i transparency %.
2f intmultiplier %i\n\n", POPULATION, MAXGROUP, SKILLS,
INITIALSOCIALTRANSPARENCY, INTMULTIPLIER);
    printf("hey Adam Smith, it's Jan21 2011!\n");
    initmarket();
    initpopulation();
    selectgiftedpopulation();
    market.totalmisurplus = 0;
    for (int i = 1; i < POPULATION; i++) {
        for (int need = 1; need < SKILLS; need++) {
            individual[i].producedxperiod[need] = market.basicneed[need];
            individual[i].soldxperiod[need] = 0;
            individual[i].purchasedxperiod[need] = 0;
        }
    }
    for (int j = 1; j < SKILLS; j++) {
        if (individual[TESTINDIVIDUAL].gifted[j] == TRUE)
            printf("indy is gifted for need %i \n", j);
    }
    printf("evoluutio alkaa\n");
    for (int period = 1; period < MAXPERIODS; period++) {
        // resetperiodicstat(); // not used as we manually reset where needed
        for (int me = 1; me < POPULATION; me++) {
            individual[me].totalsurplus = 0;
        }
        for (int need = 1; need < SKILLS; need++) {
            market.periodictcecost[need] = 0;
            market.periodicspoils[need] = 0;
        }
        market.period = period;
        evolution();
        evaluatemarketprices(); // This will print detailed per-good
statistics each period
        long long totalproduction = 0;
        long long totalperiodictcecost = 0;
        for (int i = 1; i < SKILLS; i++) {
```

```
            // Note: market.surplus is not updated, but total surplus across
agents is computed below as totalisurplus.
            totalproduction += market.producedthisperiod[i];
            totalperiodictcecost += market.periodictcecost[i];
            market.numberofrecentlyproduced[i] = (long long)
((double)market.numberofrecentlyproduced[i] * DISCONT);
            printf("Need %i, role %i srplus %6lli, stcklmt %6lli, eff %3.1f,
rprdcd %7lli prodnow %7lli rpurch %6lli, purchnow %6lli p$ %.2f rsls %6lli
soldnow %6lli sp$ %.2f\n",
                    i,
                    individual[TESTINDIVIDUAL].role[i],
individual[TESTINDIVIDUAL].surplus[i],
individual[TESTINDIVIDUAL].stocklimit[i],
                    individual[TESTINDIVIDUAL].efficiency[i],
individual[TESTINDIVIDUAL].recentlyproduced[i],
individual[TESTINDIVIDUAL].producedthisperiod[i],
                    individual[TESTINDIVIDUAL].recentlypurchased[i],
individual[TESTINDIVIDUAL].purchasedthisperiod[i],
                    individual[TESTINDIVIDUAL].purchaseprice[i],
individual[TESTINDIVIDUAL].recentlysold[i],
individual[TESTINDIVIDUAL].soldthisperiod[i],
                    individual[TESTINDIVIDUAL].salesprice[i]);
        }
        if (totalperiodictcecost > (LLONG_MAX / MAXGROUP))
printf("totalperiodictcecost closes on llongmax");
        if (totalproduction > (LLONG_MAX / MAXGROUP))
printf("totalproduction closes on llongmax");
        double sumremainingperiods = 0;
        long long totalisurplus = 0;
        long long sumofspoils = 0;
        float sumovertimes = 0.0f;
        double sumneeds = 0.0f;
        double sumrecentneeds = 0.0f;
        double sumpurchaseprice1 = 0.0f;
        market.loosers = 0;
        for (int i = 1; i < POPULATION; i++) {
            totalisurplus += individual[i].totalsurplus;
            sumremainingperiods += individual[i].periodremaining;
            // sumneedsincrement was an old metric (not used now), but we
consider needslevel:
            sumneeds += individual[i].needslevel;
            sumrecentneeds += individual[i].recentneedsincrement;
            sumofspoils += individual[i].periodicspoils;
            sumovertimes += individual[i].timeout;
            if (individual[i].periodremainingdebt < (-1 *
market.periodlength)) {
                // This agent could be considered a "loser" who didn't meet
needs (they had to reset needslevel).
                market.loosers++;
            }
            individual[i].timeout = 0; // reset timeout counter for next
```

```
period
            sumpurchaseprice1 += individual[i].purchaseprice[1];
        }
        // Compute average deviation of recentneedsincrement for population
(how much needs adjustments varied)
        if (totalisurplus > (LLONG_MAX / MAXGROUP)) printf("totalisurplus
closes on llongmax");
        double recentneedsdeviation = 0;
        float indydeviation = 0.0f;
        float averageneeds = sumneeds / POPULATION;
        float averagerecentneeds = sumrecentneeds / POPULATION;
        for (int i = 1; i < POPULATION; i++) {
            indydeviation = (averagerecentneeds -
individual[i].recentneedsincrement);
            recentneedsdeviation += (indydeviation < 0 ? -indydeviation :
indydeviation);
        }
        recentneedsdeviation = recentneedsdeviation / POPULATION;
        double sumpurchase1deviation = 0.0f;
        float purchaseprice1average = sumpurchaseprice1 / POPULATION;
        float averagepurchaseprice1deviation = 0.0f;
        for (int i = 1; i < POPULATION; i++) {
            sumpurchase1deviation += fabs(purchaseprice1average -
individual[i].purchaseprice[1]);
        }
        averagepurchaseprice1deviation = (float)(sumpurchase1deviation /
POPULATION);
        // Print summary metrics for the period:
        printf("\np%i freetime %.3f utility %.3f spoilitptime %.4f
tcecostitptime %.4f storeditptime %.3f totproditptime %.3f prodbalance %.
3f\nloosers %i needsdeviation %.4f dev/aveinc %.4f maxneed1satisfied %.4f
overtime %.3f tcepertime %.3f spoilstime %.3f\n\n",
               period, (float)(sumremainingperiods / (POPULATION *
market.periodlength)), (averageneeds),
               (float)sumofspoils / (market.periodlength * POPULATION),
               (float)((double)totalperiodictcecost / (POPULATION *
market.periodlength)),
               (float)((double)(totalisurplus - market.totalmisurplus) /
(POPULATION * market.periodlength)),
               (float)((double)totalproduction / (market.periodlength *
POPULATION)),
               (((float)((double)totalproduction / (market.periodlength *
POPULATION))) -
                (((float)((double)(totalisurplus - market.totalmisurplus) /
(POPULATION * market.periodlength)) +
                   (float)sumofspoils / (market.periodlength * POPULATION) +
(averageneeds)) +
                 (float)((double)totalperiodictcecost / (POPULATION *
market.periodlength)))),
               market.loosers, (float)recentneedsdeviation, (float)
(recentneedsdeviation / averagerecentneeds),
```

```
                (float)(market.maxefficiency[1] * market.maxefficiency[1] *
market.basicneed[1]) / ((averageneeds) * market.elasticneed[1]),
                sumovertimes / POPULATION,
                (float)market.totalcostoftceintime / (market.periodlength *
POPULATION),
                (float)market.totalcostofspoilsintime / (market.periodlength
* POPULATION));
        // Write CSV line to file (some terms here appear to mismatch the
printed ones, likely focusing on certain key metrics for analysis)
        fprintf(f,
"%i, %.3f, %.3f, %.4f, %.4f, %.3f, %.3f, %.4f, %.4f, %.3f, %.3f, %.3f, %.3f,
%.3f, %.3f, %.3f, %.3f\n",
                period, (averageneeds),
                (float)((double)sumofspoils / POPULATION) /
market.periodlength,
                (float)((double)totalperiodictcecost / (POPULATION *
market.periodlength)), (float)((double)(totalisurplus -
market.totalmisurplus) / (POPULATION * market.periodlength)),
                (float)((double)totalproduction / (market.periodlength *
POPULATION)), (float)recentneedsdeviation, (float)(recentneedsdeviation /
(averagerecentneeds)),
                (float)(market.maxefficiency[1] * market.maxefficiency[1]) /
((averageneeds) * (market.priceaverage / market.averageprice[1]) *
(market.priceaverage / market.averageprice[1])),
                sumovertimes / POPULATION,
                purchaseprice1average, averagepurchaseprice1deviation,
                (float)market.numberofrecentlyproduced[1] / (POPULATION *
market.basicneed[1] * (HISTORY + 1)), (float)(market.maxefficiency[1] *
market.maxefficiency[1]), (float)((averageneeds) * (market.priceaverage /
market.averageprice[1]) * (market.priceaverage / market.averageprice[1])),
                (float)market.totalcostoftceintime / (market.periodlength *
POPULATION),
                (float)market.totalcostofspoilsintime / (market.periodlength
* POPULATION),
                (float)(market.maxefficiency[1] * market.maxefficiency[1]),
((averageneeds) * (market.priceaverage / market.averageprice[1]) *
(market.priceaverage / market.averageprice[1])));//consider elasticneed...
        market.totalmisurplus = totalisurplus;
        if (mode != '0') mode = getchar();  // If interactive mode, wait for
user key press to continue
    }
    fclose(f);
    return 0;
}
```

---

1 2 3 4 5 8 14 42 46 Modernizing the Emergent Money Simulation – Comprehensive Plan.pdf
file://file-RTqSZ9WShYs6mKNJ78JWzu

6 7 9 10 11 12 13 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
37 38 39 40 41 43 44 45 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67
68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96
97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123
124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148

adamsmithfeb17B.c

file://file-Vxip8qw8SHXxXvPN9oNA1r