ZIP60: Further Explorations in the Evolutionary Design of Online Auction Market Mechanisms

Dave Cliff
Digital Media Systems Laboratory
HP Laboratories Bristol

HPL-2005-85
May 20, 2005*

algorithmic trading, online auction marketplaces, e-marketplaces, automated market mechanism design, trader-agents, ZIP traders, genetic algorithms

The Zero-Intelligence Plus ("ZIP") adaptive automated trading algorithm has been demonstrated to outperform human traders in experimental studies of continuous double auction (CDA) markets populated by mixtures of human and "robot" traders . To successfully populate a market with ZIP traders, the values of eight control parameters need to be set correctly. While these eight values can be set manually, previous papers have demonstrated that values of those parameters can be automatically optimized using a genetic algorithm (GA), to tailor ZIP traders to particular markets, and also (by adding an additional real-valued numeric parameter) to automatically discover novel new forms of auction market mechanism that are more efficient than the CDA. This paper introduces a more sophisticated version of the ZIP algorithm, which is shown to produce significantly better results. The extended variant is known as "ZIP60", because it requires 60 real-valued control parameters to be set correctly, and the original ZIP algorithm is re-named "ZIP8" accordingly. Manually choosing the correct values for 60 control parameters would be a very laborious task, but it is demonstrated here that an appropriate automatic optimization process can discover good sets of values for the parameters. A simple GA operating in the 60-dimensional parameter space is shown to produce ZIP60 traders with mean scores significantly improved over ZIP8s, but also with high variance in those improvements. A slight revision of this approach is shown to give results with even higher mean improvements and also with lower variance in those improvements. The revised approach involves giving the GA control over the dimensionality of the parameter space being searched, starting with an eight-dimensional space and then allowing the GA to automatically and gradually expand that space up to sixty-dimensional only when the increased number of parameters leads to identifiably better solutions. The results from ZIP60, while better than ZIP8, show a greatly reduced incidence of cases where the GA discovers auction mechanisms that are significantly better than the fixed CDA mechanism. This may be interpretable as evidence that the earlier ZIP8 results (where improvements on the CDA were common) were consequences of the relative lack of sophistication in the ZIP8 algorithm.

1. Introduction[1]

For thousands of years, buyers and sellers have come together to exchange money for goods or services. Economists use the word “auction” to refer to the mechanism (or rules) by which buyers and sellers interact in such marketplaces. Almost all traders in the global international financial markets interact via a particular form of auction market mechanism known as the *continuous double auction* (CDA), more details of which will be given later.[2] The CDA has been the subject of much study by economists, partially because it is so important in the world of finance, but also because CDA markets typically exhibit a very attractive characteristic: experimental studies have demonstrated that the transaction prices in a CDA market rapidly converge on the market’s theoretical *equilibrium price*. Students of microeconomics know the equilibrium price as the price at which the market’s supply and demand curves intersect; but, colloquially, the equilibrium price is important because if transactions are taking place at off-equilibrium prices then someone somewhere in the market is being ripped off. Hence, rapid equilibration is desirable in any auction. The precise reasons why CDA markets typically exhibit rapid and stable equilibration are still the topic of research and debate (see e.g. [15]).

With the advent of e-commerce, various forms of auction mechanism have become very popular for online trading, and web-based auction sites such as `www.ebay.com` have proven highly successful. As auctions dematerialize, moving online and becoming virtual “e-marketplaces”, it becomes perfectly plausible for software-agent “robot” traders to participate in those auctions. In comparison to human traders, such robots have the advantage of being very fast and very cheap, and in principle they can assimilate and act on volumes of data that would swamp even the most able of human traders.

In the past couple of years, a leading-edge issue in the technical community that work within the world’s financial markets has been the topic of *Algorithmic Trading*, where jobs performed traditionally by human traders are increasingly automated and executed by algorithms. Increasingly, algorithmic trading systems are becoming capable of going beyond mere replication of the roles of human traders, and they are now routinely capable of executing buy or sell orders in ways that are pretty much impossible for a human trader, because the human is limited by being built from slow and noisy biological materials. For example, a single big block trade (the sheer size of which might adversely affect the market price for the entity being traded) might be automatically broken up by the algorithmic trading system into a very large number of very small trades, which are then spread out over some time period to take advantage of trading opportunities as they occur, where each small trade is executed only a few microseconds after that trading opportunity opens up.

For the purposes of this paper, the world financial markets can be considered as just another set of instances of online e-marketplaces. This is obviously true for those centralized financial marketplaces (such as many major equity exchanges like the London Stock Exchange, or NASDAQ; but not *yet* the New York Stock Exchange) where there is

[1] This paper is a revised and extended version of [11]; for completeness, much of the introductory and expository text in [11] is repeated here verbatim.
[2] For a recent review discussing the space of possible auction mechanisms, see [25].

a high degree of automation and machine processing of data. For those other major international financial systems where trading currently takes place in the absence of central exchanges (such as the $1.5bn/day international foreign-exchange markets, where decentralized networks of broker-to-broker interactions constitute much of the market), increasing penetration of appropriate automation technologies is clearly occurring, and so it is only a matter of time until they also are, in effect, online e-marketplaces.

ZIP (Zero-Intelligence-Plus) artificial trading agents, introduced in [3], are software-agent "trader robots" that use simple machine learning techniques to adapt to operating as buyers or sellers in electronic versions of the open-outcry auction-market environments similar to those used in Smith's [25] pioneering experimental economics studies of the CDA and other auction mechanisms. ZIP traders were originally developed as a solution to the pathological failures of Gode & Sunder's (1993) "ZI" (Zero-Intelligence) traders, but work done at IBM's research labs in 2001 by Das *et al.* [12] has shown that ZIP traders (unlike ZI traders) consistently out-perform human traders in human-against-robot experimental-economics CDA marketplaces. The ZIP traders consistently made profits a few percentage points higher than did the human traders they were competing against, and ZIP was the highest-scoring algorithm in the IBM study. Das *et al.* [12] wrote that the "…successful demonstration of machine superiority in the CDA … could have a … powerful financial impact – one that might be measured in billions of dollars annually", and in their conclusions they speculate on the future possibility of online e-marketplaces currently populated by human traders becoming populated entirely by trader agents.

The operation of ZIP traders has been successfully demonstrated in experimental versions of CDA markets similar to those found in the international financial markets for commodities, equities, capital, and derivatives; and in posted-offer auction markets similar to those seen in domestic high-street retail outlets [3]. In any such market, there are a number of numeric parameters that govern the adaptation and trading processes of the ZIP traders. In the original 1997 version of ZIP traders, the values of these were set by hand, using "educated guesses". However, subsequent papers [4,5] presented the first results from using a standard evolutionary computation technique to automatically optimize these parameter values, thereby eliminating the need for skilled human input in deciding the values.

Prior to the research described in [6], in all previous work using artificial trading agents – ZIP or otherwise – the market mechanism (i.e., the type of auction the agents are interacting within) had been fixed in advance. Well-known market mechanisms from human economic affairs include: the English auction (where sellers stay silent and buyers quote increasing bid-prices), the Dutch Flower auction (where buyers stay silent and sellers quote decreasing offer-prices); the Vickery or second-price sealed-bid auction (where sealed bids are submitted by buyers, and the highest bidder is allowed to buy, but at the price of the *second-highest* bid: game-theoretic analysis demonstrates that this mechanism encourages honesty and is robust to attack by dishonest means); and the CDA (where sellers announce decreasing offer prices while *simultaneously and asynchronously* the buyers announce increasing bid prices, with the sellers being free to accept any buyer's bid at any time and the buyers being free to accept any seller's offer at any time, in the absence of an auctioneer).

This paper continues the detailed exploration (started in [11]) of some specific consequences of asking the following question: if, as Das *et al.* [12] speculate, trader agents will come to replace human traders in online e-marketplaces, then why should those online e-marketplaces use auction mechanisms designed by humans, for humans? Perhaps there are new market mechanisms, *suitable only to populations of robot-traders*, that are more efficient (or otherwise more attractive) than currently-known human-based mechanisms.

Designing new market mechanisms is hard, and the space of possible mechanisms is vast. For this reason it is attractive to use an *automated search* of the space of possible mechanisms: in essence, we ask a computer to do the auction-design for us. This paper reports on exploring the application of one type of automated search/optimization algorithm, which is inspired by Darwinian notions of evolution via random variation and directed selection, and hence is known as a Genetic Algorithms (GA).

The first results from experiments where a GA optimizes not only the parameter values for the ZIP trading agents, but also the style of market mechanism in which those traders operate, were presented in [6]. To do this, a space of possible market mechanisms was created for evolutionary exploration. The space includes the CDA and also one-sided auctions similar (but not actually identical to) the English Auction (EA) and the Dutch Flower Auction (DFA). Significantly, this space is *continuously variable*, allowing for any of an *infinite* number of peculiar hybrids of these auction types to be evolved, which have no known correlate in naturally occurring (i.e., human-designed) market mechanisms. While there is nothing to prevent the GA from settling on solutions that correspond to the known CDA auction type or the EA-like and DFA-like one-sided mechanisms, it was found that hybrid solutions can lead to the most desirable market dynamics. Although the hybrid market mechanisms could easily be implemented in online electronic marketplaces, they have not been designed by humans: rather they are the product of an automated search through a continuous space of possible auction-types. Thus, the results in [6] were the first demonstration that radically new market mechanisms for artificial traders may be designed by automatic means.

This is not a trivial academic point: although the efficiency of the evolved market mechanisms are typically only a few percentage points (or even only a few basis points) better than those of the established human-designed mechanisms, the economic consequences could be highly significant. According to figures released by the New York Stock Exchange (NYSE), the total value of trades on the CDA-based NYSE for the year 2000 was $11060bn (i.e., a little over 11 trillion dollars: see [19]). If only 0.1% of that liquidity could be eliminated or captured by a more efficient evolved market mechanism, the value saved (or profit generated) would still be in excess of $10bn. And that is just for one market: similar savings could presumably be made at NASDAQ, at European exchanges such as LSE and LIFFE, and at similar exchanges (central or distributed) elsewhere around the globe.

Section 2 gives an overview of ZIP traders and of the experimental methods used, including a description of the continuously-variable space of auction types. This description is largely identical to the account given in previous papers [6, 7], albeit extended to describe how the new experiments whose results are presented here differ from the previous work. The new ZIP60 results are presented in Section 3. The Appendix presents illustrative source code.

2. Methods

2.1 Eight-parameter Zero-Intelligence-Plus (ZIP8) Traders

The original eight-parameter ZIP trading agents were described fully in a lengthy report [3], which included sample source-code in the C programming language. For the purposes of this paper a high-level description of the eight key parameters is sufficient. Illustrative C source-code for the ZIP60 revision is presented in Appendix A of this paper.

ZIP[3] traders deal in arbitrary abstract commodities. Each ZIP trader i is given a private (i.e., secret) limit-price, λ_i, which for a seller is the price below which it must not sell and for a buyer is the price above which it must not buy. If a ZIP trader completes a transaction at its λ_i price then it generates zero utility ("profit" for the sellers or "saving" for the buyers). For this reason, each ZIP trader i maintains a time-varying utility margin $\mu_i(t)$ and generates quote-prices $p_i(t)$ at time t according to $p_i(t)=\lambda_i(1+\mu_i(t))$ for sellers and $p_i(t)=\lambda_i(1-\mu_i(t))$ for buyers. The "aim" of traders is to maximize their utility over all trades, where utility is the difference between the accepted quote-price and the trader's λ_i value. Trader i is given an initial value $\mu_i(0)$ (i.e., $\mu_i(t)$ for $t=0$) which is subsequently adapted over time using a simple machine learning technique known as *the Widrow-Hoff rule* which is also used in back-propagation neural networks [23] and in learning classifier systems [27]. This rule has a "learning rate" parameter β_i that governs the speed of convergence between trader i's quoted price $p_i(t)$ and the trader's idealized "target" price $\tau_i(t)$. When calculating $\tau_i(t)$, traders introduce a small random absolute perturbation generated from[4] $\boldsymbol{U}[0,c_a]$ (this perturbation is positive when increasing $\tau_i(t)$, negative when decreasing) and also a small random relative perturbation generated from $\boldsymbol{U}[1-c_r,1]$ (when decreasing $\tau_i(t)$) or $\boldsymbol{U}[1,1+c_r]$ (when increasing $\tau_i(t)$). Here c_a and c_r are global system constants. To smooth over noise in the learning system, there is an additional "momentum" parameter γ_i for each trader (such momentum terms are also commonly used in back-propagation neural networks).

Thus, adaptation in each ZIP trader i has the following parameters: initial margin $\mu_i(0)$; learning rate β_i; and momentum term γ_i. In an entire market populated by ZIP traders, values for these three parameters are randomly assigned to each trader via the following expressions: $\mu_i(0)=\boldsymbol{U}(\mu_{min}, \mu_{min}+\mu_\Delta)$; $\beta_i=\boldsymbol{U}(\beta_{min}, \beta_{min}+\beta_\Delta)$; and $\gamma_i=\boldsymbol{U}(\gamma_{min}, \gamma_{min}+\gamma_\Delta)$.

Hence, to initialize an entire ZIP-trader market it is necessary to specify values for the six market-initialization parameters μ_{min}, μ_Δ, β_{min}, β_Δ, γ_{min}, and γ_Δ; and also for the two global

[3] Herein, the acronym "ZIP" with no numeric suffix is intended to mean "both ZIP8 and ZIP60".
[4] Here $v=\boldsymbol{U}[x,y]$ denotes a random real value v generated from a uniform distribution over the range $[x,y]$.

system constants c_a and c_r. And so it can be seen that any set of initialization parameters for a ZIP-trader market exists within an eight-dimensional real space. Vectors in this 8-space can be considered as “genotypes” in a genetic algorithm (GA), and from an initial population of such genotypes it is possible to allow a GA to find new genotype vectors that best satisfy an appropriate evaluation function. This is exactly the process that was introduced in [4, 5], and that is described further below. Before that, the issue of simulating the passage of time is discussed.

When monitoring events in a real auction, as more precision is used to record the time of events, so the likelihood of any two events occurring at exactly the same time is diminished. For example, if two bid-quotes made at five minutes past nine are both recorded as occurring at 09:05, then they appear to be simultaneous; but a more accurate clock would have been able to reveal that the first bid was made at 09:05:01 and the second at 09:05:02. Even if two events occur absolutely at the same time, some random process (e.g. what direction the auctioneer is looking in) may break the simultaneity.

Thus, we may simulate real marketplaces (and implement electronic marketplaces) using techniques where each significant event always occurs at a unique time. We may choose to represent these by real high-precision times, or we may abstract away from precise time-keeping by dividing time (possibly irregularly) into discrete *slices*, numbered sequentially, where one significant event is known to occur in each slice. In the ZIP-trader markets explored here, we use such a time-slicing approach. In each time-slice, the atomic “significant event” is one quote being issued by one trader and the other traders then responding either by ignoring the quote or by one of the traders accepting the quote. (NB Das *et al.* [12] used a continuous-time formulation of the ZIP-trader algorithm).

In the markets described here (and in [3,4,5,6,7,8,9]), on each time-slice a ZIP trader i is chosen at random from those currently able to quote (i.e. those who hold appropriate stock or currency), and trader i’s quote price $p_i(t)$ then becomes the “current quote” $q(t)$ for time t. Next, all traders j on the contraside (i.e. all buyers j if i is a seller, or all sellers j if i is a buyer) compare $q(t)$ to their own current quote price $p_j(t)$ and if the quotes cross (i.e. if $p_j(t)<=q(t)$ for sellers, or if $p_j(t)>=q(t)$ for buyers) then the trader j is able to accept the quote. If more than one trader is able to accept, one is chosen at random to make the transaction. If no traders are able to accept, the quote is regarded as “ignored”. Once the trade is either accepted or ignored, the traders update their $\mu(t)$ values using the learning algorithm outlined above, and the current time-slice ends. This process repeats for each time-slice in a trading period, with occasional injections of fresh currency and stock, or redistribution of λ_i limit prices, until either a maximum number of time-slices have run, or a maximum number of sequential quotes have been ignored.

2.2. A Space of Possible Auctions

Now consider the case where we implement a ZIP-trader continuous double auction (CDA) market. In any one time-slice in a CDA either a buyer or a seller may quote, and in the definition of a CDA a quote is equally likely from each side. One way of implementing a CDA is, at the start of each time-slice, to generate a random binary variable to determine whether the next quote will come from a buyer or a seller, and then to randomly choose one individual as the quoter from whichever side the binary value points to. Here, as in previous ZIP work [3, 4, 5, 6, 7, 8, 9] the random binary variable is always independently and identically distributed over all time-slices.

So, let $Q=b$ denote the event that a buyer quotes on any one time-slice and let $Q=s$ denote the event that a seller quotes, then for the CDA we can write $Pr(Q=s)=0.5$ and note that because $Pr(Q=b)=1.0-Pr(Q=s)$ it is only necessary to specify $Pr(Q=s)$, which we will abbreviate to Q_s hereafter. Note additionally that in an English Auction (EA) we have $Q_s=0.0$, and in the Dutch Flower Auction (DFA) we have $Q_s=1.0$. Thus, there are at least three values of Q_s (*0.0, 0.5*, and *1.0*) that correspond to three types of auction familiar from centuries of human economic affairs. Although the ZIP-trader case of $Q_s=0.5$ is indeed a good approximation to the CDA, the fact that any ZIP trader j will accept a quote whenever $q(t)$ and $p_j(t)$ cross means that the one-sided extreme cases $Q_s=0.0$ and $Q_s=1.0$ are not *exact* analogues of the EA and DFA.

The inventive step introduced in [6] was to consider the Q_s values of 0.0, 0.5, and 1.0 not as three distinct market mechanisms, but rather as the two endpoints and the midpoint on a *continuum* of mechanisms. For values other than these, there is a straightforward implementation. For example, Q_s=0.1 can be interpreted as specifying an auction mechanism where, on the average, for every nine quotes by buyers, there will be one quote from a seller. Yet the history of human economic affairs offers no examples of such markets: why would anyone suggest such a bizarre way of operating? And who would go to the trouble of setting themselves up to act as an auctioneer for such a mechanism? Certainly, it is perfectly possible for a human auctioneer to run an auction using a value of Q_s other than 0.0, 0.5, or 1.0. For any given value of Q_s, all that the auctioneer needs is an unbiased roulette-wheel partitioned into two segments: one marked “Seller” and measuring Q_s*360 degrees of arc; the other marked “Buyer” and measuring (1.0-Q_s)*360 degrees. To determine the source of each successive new quote in the auction, the auctioneer would spin the wheel and then, depending on whether the ball ends up in the “Seller” or the “Buyer” segment, would take the next quote either from a seller or a buyer. Clearly, an online version of such an auction mechanism can be implemented in only a few lines of code, so long as an appropriate method for generating random numbers is available. But (to the best of my knowledge) neither the manual roulette-wheel version nor the online implementation of such auction mechanisms have ever been implemented before for any value of Q_s other than 0.0, 0.5, or 1.0.

Nevertheless, there is no *a priori* reason to argue that these three previously-known points on this Q_s continuum are the only loci of useful auction types. Maybe there are circumstances in which values such as $Q_s=0.25$ (say) are preferred. Given the infinite nature of this real continuum it seems appealing to use an automatic exploration process, such as the GA, to identify useful values of Q_s.

Thus, in [6] a ninth dimension was added to the search space, and thus the genotype in the GA became the eight real values for ZIP-trader initialization, plus a real value for Q_s. As before, no “NYSE” quote-improvement rule [3] was used in the experiments reported in this paper.

2.3. The Genetic Algorithm

The simple GA used in [5] is also used here, with one difference. In [5] a population of size 30, evolving for 1000 generations, was used. Each experiment was repeated 50 times, and it was found that several of the experiments yielded multi-modal results. However, in all the experiments reported on in that paper, the qualitative nature of the outcome of the experiment was very clear by generation 500: all runs settled to a particular mode by generation 300, and the improvement in performance (i.e., fitness) between generation 500 and generation 1000 was always very small. Thus the experiments reported here were ended after 500 generations. All other GA control parameters are unchanged. For an introduction to GAs, see [17] or [18].

In each generation, all individuals were evaluated and assigned a fitness value; and the next generation's population was then generated via mutation and crossover on parents identified using rank-based tournament selection. Elitism (where, on each generation, an unadulterated version of the fittest individual from the evaluated population is copied into the new successor population) was also used.

The genome of each individual was simply a vector of nine real values. In each experiment, the initial random population was created by generating random values from $\boldsymbol{U}[0,1]$ for each locus on each individual's genotype. Crossover points were between the real values, and crossover was governed by a Poisson random process with an average of between one and two crosses per reproduction event. Mutation was implemented by adding random values from $\boldsymbol{U}[-m(g),+m(g)]$ where $m(g)$ is the mutation limit at generation g (starting the count at $g=0$). Mutation was applied to each locus in each genotype on each individual generated from a reproduction event, but the mutation limit $m(g)$ was gradually reduced via an exponential-decay annealing function of the form: $\log_{10}(m(g))=\log_{10}(m_s)-((g/(n_g-1)).\log_{10}(m_s/m_e))$ where n_g is the number of generations (here $n_g=1000$ for consistency with [6], despite the fact that all experiments are now terminated after 500 generations) and m_s is the "start" mutation limit (i.e., for $m(0)$) and m_e is the "end" mutation limit (i.e., for $m(n_g-1)$). In all the experiments reported here, as in [6], $m_s=0.05$ and $m_e=0.0005$.

If ever mutation caused the value at a locus to fall outside the range $[0.0,1.0]$ it was simply clipped to stay within that range. This clip-to-fit approach to dealing with out-of-range mutations has been shown [1] to bias evolution toward extreme values (i.e. the upper and lower bounds of the clipping), and so Q_s values of 0.0 or 1.0 are, if anything, more likely than values within those bounds. Moreover, initial and mutated genome values of μ_Δ, β_Δ, and γ_Δ were clipped where necessary to satisfy the constraints $(\mu_{min}+\mu_\Delta)<=1.0$, $(\beta_{min}+\beta_\Delta)<=1.0$, and $(\gamma_{min}+\gamma_\Delta)<=1.0$.

The fitness of genotypes was evaluated using the methods described in [4, 5, 6]. One *trial* of a particular genome was performed by initializing a ZIP-trader market from the genome, and then allowing the ZIP traders to operate within the market for a fixed number of trading periods, with allocations of stock and currency being replenished between each trading period. During each trading period, Smith's [25] α measure (root mean square deviation of transaction prices from the theoretical market equilibrium price) was monitored, and a weighted average of α was calculated across the trading periods in the trial, using the method described in Section 2.5 below. As the outcome of any one such trial is influenced by stochasticity in the system, the final fitness value for an individual was calculated as the arithmetic mean of 100 such trials. Note that as *minimal* deviation of transaction prices from the theoretical equilibrium price is desirable, lower scores are better: we aim here to *minimize* fitness scores.

2.4. Previous ZIP8 Results

In [11], results from 32 sets of experiments were published, and are republished here in Tables 1 to 6. For consistency with those earlier results, all the new experiments whose results are tabulated in Section 3 involve evaluating the performance of the evolving auction-market mechanisms on one or more of four market supply and demand schedules. These four schedules are referred to as markets M1, M2, M3, and M4, and are illustrated in Figure 1.

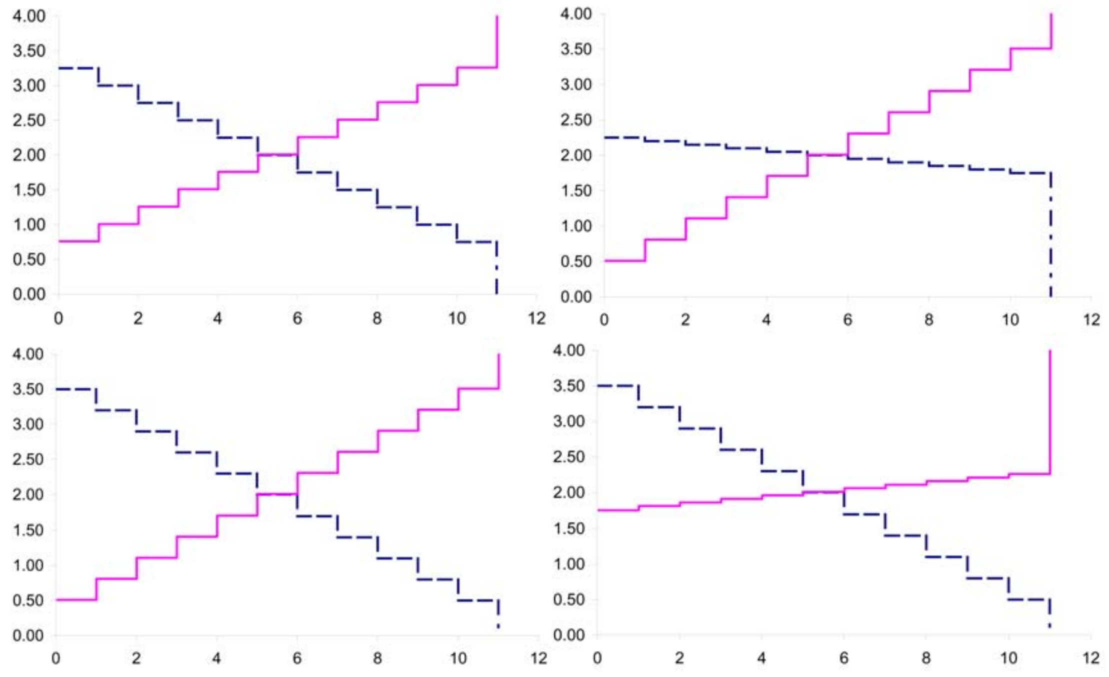


Figure 1: *Supply and demand schedules for markets M1 (top left), M2 (top right), M3 (bottom left) and M4 (bottom right). In all three figures, the horizontal axis is quantity (from 0 to 12) and the vertical axis is price (from 0.00 to 4.00). The upward-sloping supply curve is shown by the solid line, and the downward-sloping demand curve is shown by the broken line.*

In all four schedules there are 11 buyers and 11 sellers, each empowered to buy/sell one unit of commodity: these relatively small numbers of traders are the cause of the stepped supply and demand curves. Market M1 is taken from [25]. The remaining three markets are minor variations on M1. In M2 the slope of the demand curve has been greatly reduced while the slope of the supply curve has been increased only slightly; and in M4 the slope of the supply curve has been greatly reduced while the slope of the demand curve has been increased only slightly. In M3 the slopes of both the supply and demand curves are only slightly steeper than the slopes in M1. Despite the apparent similarity between M1 and M3, a detailed empirical study presented in [8] demonstrated that the minor differences between the supply and demand curves in M1 and M3 can lead to significant differences in the final best evolved solutions.

In the so-called "single-schedule" experiments, only one of the market schedules was used throughout the evolutionary process. Results from the four single-schedule experiments are summarized in Table 1 in Section 3. The key qualitative issue is that in all four experiments, the best evolved mechanisms all differed from the CDA, and in two cases the best evolved mechanism was not even a one-sided auction like the EA or DFA mechanisms; rather, the best evolved auction-mechanism was a peculiar hybrid, partway between the CDA and a pure one-sided auction.

However, because for each trial in all four of these experiments a single fixed market schedule was used in evaluating the evolving solutions, there is a manifest possibility that the GA tailored the final evolved solutions to peculiarities of the specific market supply and demand schedules employed – i.e., that it "over-fitted". To test this hypothesis, a new suite of experiments was run, where "shock changes" were inflicted on the market by swapping from one schedule to another partway through the evaluation process. The results from 19 of these experiments are presented in Section 3. Initially, dual-schedule experiments were run, where the supply and demand schedules were suddenly changed halfway through the evaluation process. Some early results from these experiments were presented in [7]: these showed that when M1 was used for the first half of the evaluation, followed by M2 for the second half (which we refer to here as M1-2), the results evolved by the GA were order-dependent. That is, when the order of the schedules was reversed, so that in the evaluation process M2 was followed by M1 (which we refer to here as M2-1), the results differed from the M1-2 case. Furthermore, for both M1-2 and M2-1, the optimal evolved values of Q_S differed from the values that were found to be optimal when evaluation involved either M1 or M2 alone. The M1-2 results are presented in detail in the next section, as illustration of the process used to compare the results from evolving-mechanism (EM) experiments with the results from fixed-mechanism (FM) experiments. In all the FM experiments, the value of Q_S is not evolved, but the remaining eight ZIP-trader parameter-values on the genotype are still optimized by the GA. The M2-1 results are presented in summary form, along with all other dual-schedule results in Table 2 (Section 3).

The order-dependence shown by the M1-2 and M2-1 results could again potentially be a consequence of the GA over-fitting: a "dual schedule" experiment could also reasonably be described as a "single-shock" experiment; and perhaps the GA had evolved solutions that were over-fitted to each particular shock. For instance, in the M1-2 case the GA might be over-fitting the evolved parameter-values and market-mechanism to the *specific* market-shock of suddenly transitioning from M1 to M2. To explore this possibility, additional sets of experiments were run where two shocks occurred during each evaluation process (i.e., switching between three schedules). Results from four such sets of triple-schedule experiments were presented in [9], all involving schedules M1, M2, and M3. In one experiment, referred to here as M1-2-1, the evaluation involved six trading periods with supply and demand determined by M1, then a sudden change to M2, then six periods later a reversion to M1 for a final six periods. The other sets of experiments are referred to here as M2-1-2, M1-2-3, and M3-2-1 (and so on), the meaning of which should be obvious. The results from these four sets of experiments are presented in summary form in Table 3 (Section 3).

For ease of comparison with the single-schedule results presented in [6], a six-period duration was used for each market schedule, meaning that a dual-schedule trial lasts for 12 periods: 6 periods with the ZIP trading agents adapting to trade under the first schedule, then at the end of the 6th period a sudden “shock change” of the market supply and demand to the second schedule (without altering any of the traders’ parameters or variable values), followed by 6 periods of the traders adapting to trade and under that new schedule. Similarly, the triple-schedule experiments each lasted for 18 trading periods.

In [6], the evaluation function was a weighted average of Smith’s α measure: in each trading period p the value α_p was calculated, and the fitness score was computed as $(1/\Sigma w_p).\Sigma(\alpha_p.w_p)$ for p=1…6 with weights w_1=1.75, w_2=1.5, w_3=1.25, and $w_{p>3}$=1.0. In the dual-schedule experiments reported here, this was extended so that p=1…12 and $w_{p>6}=w_{p-6}$. Similarly, in the triple-schedule experiments, p=1…18 and $w_{p>12}=w_{p-12}$.

Results from the 32 sets of experiments presented in [11] are reproduced here in Tables 1 to 6: one set for each sequence of schedules explored. Each set involves 100 individual experiments: 50 repetitions of the GA experiment for the evolving-mechanism (EM) case where the value of Q_s is under evolutionary control, and (for comparison) a further 50 repetitions for the same sequence in the fixed-mechanism (FM) case, where Q_s is fixed at the CDA value of 0.5. Of the 32 sets, 4 are single-schedule, 10 are dual-schedule and the remaining 18 are triple-schedule. Given that each of the 100 experiments performed for any one schedule involves evaluating each of 30 individuals over 500 generations, where evaluating any one individual requires calculating its average score over 100 trials and each trial is either 6, 12, or 18 trading days long, the data presented in Tables 1 to 6 summarize results from a little over seventy billion simulated trading days.[5]

Section 2.4.1 gives a detailed presentation of results from the M1-2 case, for illustration of the process used to compare the EM and FM cases. Section 2.4.2 then presents and discusses the tables summarizing the results from all the ZIP8 experiments reported in [11].

2.4.1 Detailed Dual-Schedule Results: M1-2

Figure 2a shows the fitness of all 30 genotypes in the population at each generation from 1 to 500 in a single run of the M1-2 evolving-market (EM) experiment. In each generation the elite (best-scoring) individual is of most interest, and Figure 2b shows the

[5] 100*30*500*100*(4*6+10*12+18*18)=70,020,000,000.

trajectory of the elite fitness score for the population shown in Figure 2a. The results shown in Figure 2 are non-deterministic: different runs of the GA (with different seed values for its random number generator) will yield different elite trajectories.

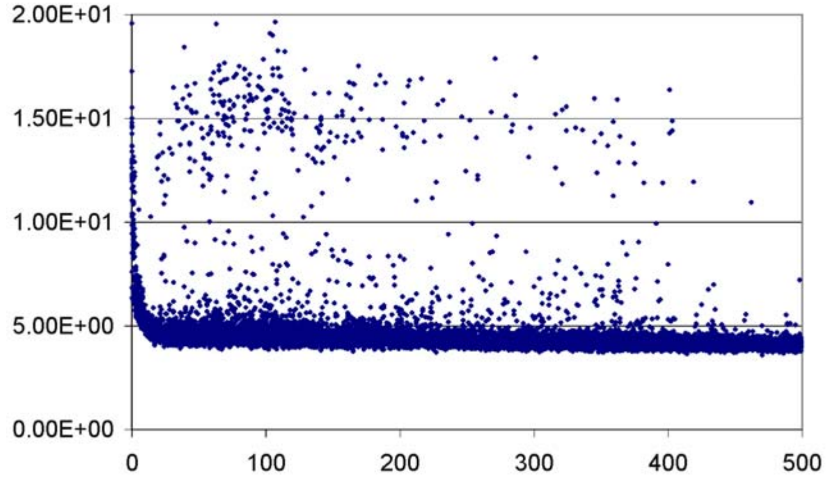
Examining the results from 50 repetitions of this experiment (with a different random seed used in each experiment), the results are clearly bimodal. Of the 50 repetitions, in 36 the elite ends up on fitness minima of about 3.85, while the other two elite fitness mode involves less-good minima around 4.2 to 4.3. Figure 3 shows the evolutionary trajectory of the mean and standard deviation (s.d.) of the Q_s values on the genomes of the 36 members of the best elite mode. Clearly, the elite mode uses a hybrid auction mechanism partway between the one-sided Q_s=0.0 market and the Q_s=0.5 CDA.

For comparison, similar trajectories of fitness values were recorded from 50 repetitions of the M1-2 experiment in fixed-market (FM) conditions (i.e., where the value of Q_s was *not* evolved) for Q_s=0.0, Q_s=0.5, and Q_s=1.0 respectively. Using Q_s=0.0 is plausible because in [6] separate experiments evolving on M1 and on M2 alone both converged on optima at Q_s=0.0. Moreover, using Q_s=0.5 gives a CDA, which is often celebrated as an auction mechanism in which transaction-price equilibration is rapid and stable, so we could plausibly expect the best fitness from using that market type. Fixed-market Q_s=1.0 results were generated for completeness, as this is analogous to the human-designed DFA mechanism.

With Q_s fixed at zero, the mean best-mode elite score is around 4.1; and with Q_s=1.0 the results are worse, by a factor of more than two [7]. With the fixed CDA Q_s=0.5 mechanism, an average elite fitness of around 4.05 is settled on by almost all experiments. To ease the comparison between the EM and FM-CDA results, Figure 4 shows the mean and standard deviation of the best-mode elite scores on the same graph. The EM results are clearly lower (and hence better) than those for the FM CDA.

As our fitness values are effectively measures of market efficiency, from Figure 4 it appears that using Q_s values of around 0.23 give more efficient markets than using the previously “known” Q_s values such as 0.0, 0.5, or 1.0 for the M1-2 schedule sequence.

Noting that the evolved value of Q_s=~0.23 is close to ¼, we can informally claim that a close approximation of this evolved auction mechanism could easily be implemented in an electronic marketplace by allowing, on the average, roughly one quote in four to come from a seller while the remaining three quotes in four come from buyers.



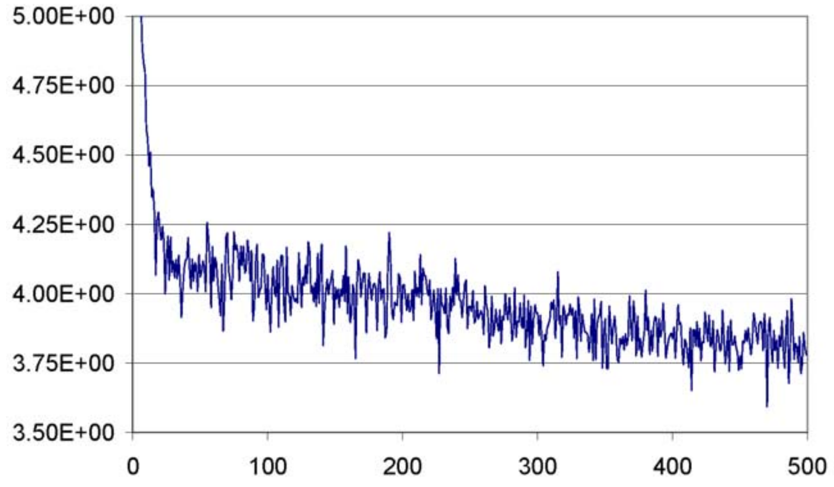


Figure 2a (upper graph): *fitness scores of all 30 members of the population for each generation. Horizontal axis is generation number (0 to 500); vertical axis is fitness score (0 to 20).* ***Figure 2b (lower graph):*** *Fitness score of the elite individual (i.e., the best genotype, with the lowest score) in each generation for the experiment shown in Figure 2a. Horizontal axis is generation number (0 to 500); vertical axis is fitness score (3.5 to 5.0).*

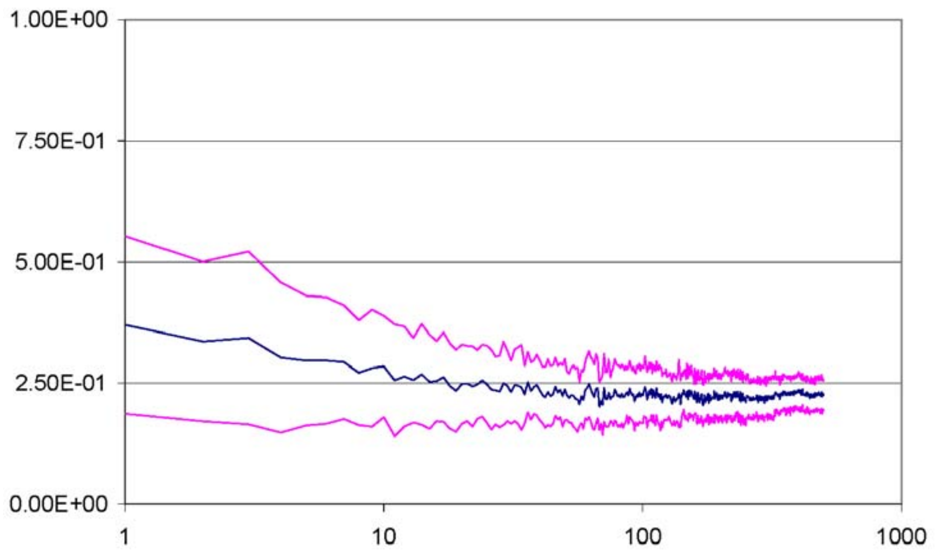


Figure 3: *Evolutionary trajectory of mean (plus and minus one s.d.; n=36) value of Q_s in the best elite mode of 50 repetitions (with different random seeds) of the experiment shown in Figure 2b. Mean Q_s settles to ~0.2*

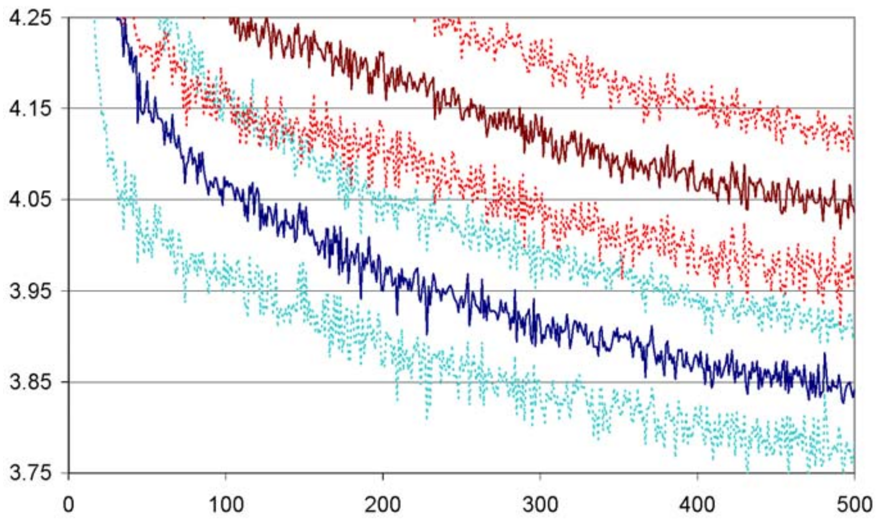


Figure 4: *Average elite fitnesses from 50 EM and 50 FM(Q_s=0.5) M1-2 experiments; data is plotted for mean fitness, plus and minus one s.d.: lower three traces shows best EM fitness mode settling to a mean of approx 3.85 with a s.d. of approx 0.06 (n=36); upper three traces show FM values settling to a mean of around 4.05 with a s.d. of approximately 0.1 (n=49).*

2.4.2 Summary Statistics

Having discussed the M1-2 results in detail, the tables in this section show summary data for a further 31 sets of experiments (each set consisting of 50 EM experiments and 50 FM experiments). As was stated earlier, results for M1, M2, and M3 were presented in [6]; Table 1 summarizes those results and presents new results from M4. The column labeling for all tables in this paper is as follows. The left-most column indicates the market schedules for each row of data. The column labelled "EM:μ" is the mean fitness at generation 500 in the best elite mode from the 50 repetitions of the EM (evolving-market) experiment, and the column labelled "EM:σ" is the standard deviation for that mean. The column labelled "EM:n" shows the number of repetitions of the EM experiment that settled on the best elite fitness mode. The columns marked "FM:μ", "FM:σ", and "FM:n" show the mean fitness, standard deviation of the mean fitness, and number of repetitions (from a total of 50) for the best elite fitness mode at generation 500 in the FM (fixed-market) experiments for each schedule. The column labelled "1%?" shows whether the Wilcoxon-Mann-Whitney test [24] indicates a statistically significant difference at the 1% confidence level between the EM and FM data. Finally, the columns labelled "Q_s:μ" and "Q_s:σ" respectively show the mean Q_s value at generation 500, and the standard deviation on that mean, for the best elite mode from the EM experiments. Rows typeset in italics are those for which there is a statistically significant difference at the 1% level between the EM and FM best elite mode data.

Results for M1-2 and M2-1 were previously presented in [7]; Table 2 summarizes those results and presents new results from an additional 8 single-shock experiments. Results for M1-2-1, M2-1-2, M3-2-1 and M1-2-3 were first presented in [9]; results from an additional 14 sets of dual-shock experiments were first presented in [11].

Schedule	**EM:μ**	**EM:σ**	**EM:n**	**FM:μ**	**FM:σ**	**FM:n**	**1%?**	**Q_s:μ**	**Q_s:σ**
M1	*3.22*	*0.024*	*5*	*4.45*	*0.155*	*48*	*Y*	*0.000*	*0.0002*
M2	*2.16*	*0.103*	*45*	*3.13*	*0.141*	*50*	*Y*	*0.069*	*0.0426*
M3	*5.19*	*0.127*	*50*	*5.52*	*0.168*	*50*	*Y*	*0.158*	*0.0312*
M4	*0.60*	*0.045*	*50*	*0.72*	*0.045*	*50*	*Y*	*0.686*	*0.0433*

Table 1: *Summary of results from dual-schedule (single-shock) experiments. The column labelling is explained in the text.*

Schedule	EM:μ	EM:σ	EM:n	FM:μ	FM:σ	FM:n	1%?	Q_s:μ	Q_s:σ
M1-2	*3.85*	*0.058*	*36*	*4.04*	*0.078*	*49*	*Y*	*0.226*	*0.0309*
M2-1	4.18	0.102	46	4.18	0.092	50	N	0.456	0.0312
M2-3	3.94	0.138	49	3.98	0.128	48	N	0.561	0.0264
M3-2	*3.05*	*0.056*	*49*	*3.46*	*0.082*	*50*	*Y*	*0.137*	*0.0254*
M1-4	*2.78*	*0.061*	*36*	*3.08*	*0.069*	*50*	*Y*	*0.211*	*0.0263*
M4-1	*2.79*	*0.094*	*50*	*2.97*	*0.093*	*50*	*Y*	*0.380*	*0.0237*
M4-3	*3.01*	*0.131*	*50*	*3.25*	*0.118*	*50*	*Y*	*0.364*	*0.0184*
M3-4	*3.17*	*0.078*	*50*	*3.47*	*0.083*	*50*	*Y*	*0.212*	*0.0294*
M2-4	3.57	0.128	49	3.59	0.117	49	N	0.405	0.0394
M4-2	*2.69*	*0.079*	*50*	*2.76*	*0.075*	*50*	*Y*	*0.276*	*0.0754*

Table 2: *Summary of results from dual-schedule (single-shock) experiments. The column labelling and formatting is the same as for Table 1.*

Tables 3 to 6 all involve dual-shock (triple-schedule) evaluations, but they are grouped by the nature of the shocks. Table 3 shows results from experiments where only the demand curve undergoes a major change on each shock. Table 5 shows results from experiments where only the supply curve undergoes a major change on each shock. In Table 4, one of the two shocks involves a major change only to the demand curve while the other shock involves a major change only to the supply curve; and in Table 6 each shock involves a major change to both the supply curve and the demand curve.

Schedule	EM:μ	EM:σ	EM:n	FM:μ	FM:σ	FM:n	1%?	Q_s:μ	Q_s:σ
M1-2-1	4.35	0.084	46	4.32	0.076	49	N	0.509	0.0231
M2-1-2	3.92	0.073	50	3.91	0.076	49	N	0.497	0.0263
M2-3-2	2.99	0.075	49	3.00	0.097	49	N	0.584	0.0266
M3-2-3	3.87	0.070	50	3.86	0.087	50	N	0.528	0.0200
M1-2-3	*4.24*	*0.066*	*50*	*4.28*	*0.076*	*50*	*Y*	*0.564*	*0.0238*
M3-2-1	3.98	0.050	50	3.98	0.067	50	N	0.473	0.0218

Table 3: *Summary of results from dual-shock experiments where each shock involves a major change only to the demand curve. The column labelling and formatting is the same as for Table 1.*

Schedule	**EM:μ**	**EM:σ**	**EM:n**	**FM:μ**	**FM:σ**	**FM:n**	**1%?**	**Q_s:μ**	**Q_s:σ**
M1-4-1	*3.25*	*0.083*	*40*	*3.75*	*0.083*	*50*	*Y*	*0.187*	*0.0225*
M4-1-4	*2.30*	*0.077*	*50*	*2.49*	*0.052*	*50*	*Y*	*0.368*	*0.0205*
M4-3-4	2.83	0.056	50	2.85	0.057	50	N	0.448	0.0181
M3-4-3	*3.52*	*0.083*	*50*	*4.21*	*0.083*	*50*	*Y*	*0.146*	*0.0213*
M1-4-3	*3.25*	*0.101*	*39*	*3.90*	*0.084*	*50*	*Y*	*0.165*	*0.0199*
M3-4-1	*3.56*	*0.082*	*49*	*4.07*	*0.086*	*50*	*Y*	*0.173*	*0.0230*

***Table 4:** Summary of results from dual-shock experiments where each shock involves a major change only to the supply curve. The column labelling and formatting is the same as for Table 1.*

Schedule	**EM:μ**	**EM:σ**	**EM:n**	**FM:μ**	**FM:σ**	**FM:n**	**1%?**	**Q_s:μ**	**Q_s:σ**
M4-3-2	*2.25*	*0.096*	*50*	*2.53*	*0.091*	*50*	*Y*	*0.348*	*0.0226*
M2-3-4	*3.00*	*0.087*	*49*	*3.09*	*0.098*	*50*	*Y*	*0.575*	*0.0238*
M4-1-2	*2.97*	*0.078*	*50*	*3.11*	*0.067*	*50*	*Y*	*0.379*	*0.0188*
M2-1-4	3.29	0.082	48	3.31	0.074	50	N	0.492	0.0300

***Table 5:** Summary of results from dual-shock experiments where one shock involves a major change only to the demand curve and the other involves a major change only to the supply curve. The column labelling and formatting is the same as for Table 1.*

Schedule	**EM:μ**	**EM:σ**	**EM:n**	**FM:μ**	**FM:σ**	**FM:n**	**1%?**	**Q_s:μ**	**Q_s:σ**
M2-4-2	*3.83*	*0.088*	*50*	*3.95*	*0.096*	*50*	*Y*	*0.332*	*0.0276*
M2-4-2	3.14	0.068	50	3.14	0.084	50	N	0.496	0.0271

***Table 6:** Summary of results from dual-shock experiments where each shock involves major changes to both the supply curve and the demand curve. The column labelling and formatting is the same as for Table 1.*

Comparing the data in Tables 1 to 6, three points stand out. First, it is noticeable that in some cases, the elite evolved value of Q_s may differ quite markedly from the CDA value of 0.5, without there being a statistically significant effect on the market dynamics (i.e. on the fitness scores) in comparison to the FM Q_s=0.5 case. For example, in both M2-1 and in M2-3-2 the EM Q_s values have a mean that is over two standard deviations away from the CDA value of 0.5, which on face value could lead one to expect that the mean EM and FM fitness scores would be significantly different; yet they are not. This is a consequence of the optimum Q_s value lying on a shallow plateau-like surface in the fitness landscape, such that apparently quite different values of Q_s yield very similar fitness values: a point explored and illustrated in detail in [8].

The second notable point it that the no-shock and single-shock data are not obviously useful in predicting the results of the dual-shock experiments, despite the fact that each of the dual-shock sequences explored in Tables 3 to 6 can be considered as the concatenation of two of the single-shock sequences explored in Table 2. For instance, both M1-2-1 and M2-1-2 involve an M1-2 and an M2-1 transition. In isolation, the mean best-mode Q_s for M1-2 is 0.226 and for M2-1 is 0.456; yet for M1-2-1 the mean best-mode Q_s is 0.509 and for M2-1-2 it is 0.497.

Finally, it is clear that in the single-schedule (no-shock) experiments of Table 1, 100% of the optimum Q_s values are non-CDA; while in the dual-schedule (single-shock) experiments of Table 2, 70% are non-CDA; and in the triple-schedule (dual-shock) experiments of Tables 3 to 6, the proportion of non-CDA optima drops again to 56%. Thus, the data in Tables 3 to 6 added further weight to the conjecture (first made in [9]) that the more changes to the market supply and demand schedules during evaluation of a genotype, the more likely it is that the CDA Q_s=0.5 value is the optimal mechanism. That is, *in the limit* when nothing is predictable in advance about the market supply and demand curves, the CDA is likely to be the optimal mechanism. A corollary to this is that if there is *some* regularity in the market supply and demand, then those regularities may be exploitable, and a hybrid auction mechanism *might* exhibit better dynamics than a CDA.

2.4.3 Critique: could do better next time

The previously-published experiments and analysis of results discussed in recap thus far can be criticised on three points, all of which are addressed and remedied in the experiments and analysis introduced in Section 3.

The first criticism is one first made implicitly in [8], where the underlying fitness landscapes for the single-schedule ZIP experiments were visualised via brute-force calculation of the fitnesses. The initial reporting of results relied on monitoring the score of the best (elite) individual in the final generation (e.g. at g=500). It was noted in [8] that the stochasticity in the ZIP markets means that there will be some variation in the scores resulting from subjecting the same individual genotype to repeated testing, even when (as here) each test involves 100 trials and each trial involves at least six trading days. The practice adopted in [8] was to calculate the outcome of an experiment by taking the arithmetic mean of the elite individual scores for each of the last *k* generations, for some small *k*. In the experiments reported here (as in [8]) we would expect that by, the end of an experiment, the combination of annealing reduction in mutation rate coupled with 500 generations of selection pressure, would result in a high degree of genetic convergence, so if the elite scores from the last *k* generations do not actually come from the same individual, then those scores will have come from some very close cousins (i.e., almost identical genotypes). Thus taking the mean of the last k generations can informally be considered as taking a more precise measure of the final elite genotype's performance. So, in the new analysis of the old results presented here, and also in the analysis of the new results presented for the first time in this paper, the final result of an individual experiment will be calculated by taking the mean of the scores of the best individual from each of the last *k* generations (i.e. from *g=500-k* to *g=500*), for *k*=10.

The second criticism is somewhat technical: subsequent to the publication of [11], Feltovitch published two papers, the first of which [13] pointed out some potential problems with the Wilcoxon-Mann-Whitney nonparametric significance test that was used throughout [11], arguing that the Robust Rank-Order test is preferable; and the second of which [14] gave a set of exactly-computed values for use in the Robust Rank-Order (RRO) test. For the reasons given in [13], it seems prudent to switch to the use of RRO testing hereafter.

The third criticism is methodological. The results presented thus far rely on manual "eyeballing" identification of the elite mode, and this presents an opportunity for subjectivism and human error that could distort or bias the results. In particular, although in many of the multi-modal outcomes the variance around each mode is so small that deciding which mode each experiment has settled on is trivial and unequivocal; there are nevertheless several cases where what appears on first sight to be a single *n=50* mode can, with a little tweaking of the vertical axis scale, be revealed to be possibly two or more closely-positioned but arguably distinct modes. Of course, the weasel word here is the "arguably" in "arguably distinct" – one person's uni-modal data could be another person's closely-packed multi-modal data. To avoid this criticism, in the analysis of the ZIP60 results that follows, rather than try to manually identify the best elite mode, or to use some complicated form of automated elite-mode-detection (e.g. using cluster analysis, the application of which has itself often been criticised as somewhat more of an art than a science), we will simply compute summary statistics (e.g. mean and standard deviation, or RRO test for significance of difference) for the results from the top decile, i.e. the best 10% of the experiments. As we will be keeping here to the *n=50* repetitions method established in the previous work, we'll be comparing the results from the best-performing (i.e. lowest-scoring) five experiments in each "treatment". As illustration (and for comparison with the ZIP60 results set out below), Table 7 shows the entire *n=50* data from the multi-shock experiments reported on in Tables 3, 4, 5, and 6 re-analysed using RRO as a significance test; while Table 8 shows the same analysis, but operating only on the upper decile (n=5) data sets.

M	**EM:μ**	**EM:σ**	**FM:μ**	**FM:σ**	**Sig?**	**Qs:μ**	**Qs:σ**	**EM/FM?**
1-2-1	4.420	0.259	4.344	0.077	Y	0.503	0.035	FM
2-1-2	3.919	0.081	3.924	0.167	N	0.498	0.026	=
2-3-2	2.990	0.152	3.036	0.170	Y	0.585	0.026	EM
3-2-3	3.875	0.085	3.877	0.087	N	0.527	0.020	=
1-2-3	4.238	0.091	4.278	0.088	Y	0.565	0.024	EM
3-2-1	3.990	0.069	3.988	0.070	N	0.473	0.023	=
1-4-1	3.319	0.192	3.739	0.089	Y	0.187	0.030	EM
4-1-4	2.307	0.070	2.488	0.055	Y	0.367	0.020	EM
4-3-4	2.822	0.058	2.840	0.056	Y	0.447	0.018	EM
3-4-3	3.509	0.089	4.214	0.093	Y	0.147	0.021	EM
1-4-3	3.338	0.224	3.892	0.095	Y	0.162	0.026	EM
3-4-1	3.568	0.105	4.068	0.075	Y	0.173	0.029	EM
4-3-2	2.251	0.093	2.535	0.080	Y	0.346	0.024	EM
2-3-4	3.023	0.161	3.075	0.093	Y	0.577	0.026	EM
4-1-2	2.983	0.075	3.116	0.065	Y	0.380	0.019	EM
2-1-4	3.342	0.217	3.294	0.069	N	0.495	0.035	=
2-4-2	3.834	0.082	3.951	0.095	Y	0.333	0.027	EM
4-2-4	3.154	0.072	3.141	0.075	N	0.493	0.027	=

Table 7: *Results from the ZIP8 experiments analyzed in Tables 3, 4, 5, and 6, re-analyzed on basis of each experiment's mean elite score over last ten generations (rather than simply the score on the final generation) and using Robust Rank-Order (RRO) as a test of significance, for all n=50 experiments (i.e., ignoring any multimodality). Right-most* ***EM/FM?*** *column indicates a tri-valued summary of whether the best scores come from the EM or the FM experiments: this is calculated from the binary contents of the corresponding* ***Sig?*** *(outcome of the RRO test) column: if the* ***Sig?*** *value is "N" then there is no significant difference and hence the EM/FM shows "=" to indicate a tied result. Othewrise, the EM/FM column shows the better-scoring treatment.*

M	EM:μ	EM:σ	FM:μ	FM:σ	Sig?	Qs:μ	Qs:σ	EM/FM?
1-2-1	4.301	0.051	4.288	0.067	Y	0.504	0.014	FM
2-1-2	3.873	0.072	3.853	0.083	N	0.494	0.014	=
2-3-2	2.905	0.074	2.938	0.063	Y	0.573	0.024	EM
3-2-3	3.806	0.067	3.788	0.071	N	0.531	0.012	=
1-2-3	4.166	0.054	4.220	0.062	Y	0.564	0.014	EM
3-2-1	3.939	0.048	3.948	0.056	N	0.491	0.017	=
1-4-1	3.169	0.064	3.660	0.065	Y	0.170	0.021	EM
4-1-4	2.245	0.050	2.443	0.040	Y	0.352	0.011	EM
4-3-4	2.774	0.060	2.804	0.036	Y	0.447	0.015	EM
3-4-3	3.443	0.071	4.145	0.086	Y	0.136	0.013	EM
1-4-3	3.170	0.081	3.807	0.074	Y	0.167	0.015	EM
3-4-1	3.490	0.070	4.016	0.072	Y	0.156	0.027	EM
4-3-2	2.156	0.077	2.451	0.068	Y	0.336	0.018	EM
2-3-4	2.911	0.065	2.992	0.066	Y	0.580	0.021	EM
4-1-2	2.899	0.060	3.055	0.059	Y	0.368	0.025	EM
2-1-4	3.247	0.047	3.221	0.066	N	0.495	0.017	=
2-4-2	3.777	0.069	3.963	0.086	Y	0.348	0.021	EM
4-2-4	3.100	0.069	3.125	0.076	N	0.500	0.025	=

Table 8: *Data from the ZIP8 experiment results in Table 7, with analysis of only those experiments whose final mean elite score was in the top decile (n=5). Column descriptions as for Table 7.*

The most striking aspect of the re-analysed data is that in both the n=50 and n=5 cases, the M1-2-1 results show better results for the FM (fixed mechanism) rather than the EM (evolved mechanism) case; whereas under the previous analysis, it appeared that in all non-tied cases, the EM result was superior. Although only a 1-in-18 result, it nevertheless tarnishes the previous results somewhat, in that there is now the first evidence that the methods followed here do not *always* find either equivalents to or improvements on the CDA: the new analysis of the M1-2-1 results show that we would actually have been better-off fixing *Qs=0.5* at the outset of the experiment, rather than allowing it to be under evolutionary control.

Nevertheless, the new results from ZIP60, presented in Section 3, give us renewed cause for enthusiasm.

3. ZIP60

3.1 From 8 to 60 in five paragraphs

Despite the problem with M1-2-1 revealed by the new analysis in the previous section, the results from using a GA to fine-tune the ZIP trader parameters remain generally encouraging. In fact, they are sufficiently encouraging to prompt speculation that perhaps new variants of ZIP can be developed to take advantage of the fact that we can now (generally, at least) rely on the GA to set appropriate values for the numeric parameters affecting the ZIP market, so there is no need to try to keep the number of such parameters to sufficiently small to render them easily manageable or comprehensible by a human.

To this end, note that in ZIP8 the genome specifies the same vector of eight real values $\{\mu_{min}, \mu_{\Delta}, \beta_{min}, \beta_{\Delta}, \gamma_{min}, \gamma_{\Delta}, c_a, c_r\}$ whether the trader is a buyer or a seller. But in some situations it's perfectly plausible that the overall market dynamics might be better if the buyers were using different parameter-values to the sellers, so we could in principle have a GA-ZIP system dealing with these two cases (i.e. where *Case 1* is the trader is a buyer; *Case 2* is that the trader is a seller) and hence optimizing sixteen real parameters (i.e., "ZIP16"), where the first eight values are the vector used to initialise the buyers and the second eight are the vector used to initialise the sellers.

Furthermore, note also that in some situations a ZIP trader (whether it is a buyer or a seller) has to increase its margin, and in others it has to decrease its margin, and that it might be useful to have different parameter-values depending on which of the four cases we are in, i.e. whether the trader is a buyer raising its margin, a buyer lowering its margin, a seller raising, or a seller lowering. That would give us four times eight values, or "ZIP32".

But we can then note that, in the original specification of the ZIP algorithm, for both buyers and sellers there are actually *three* different cases or circumstances in which the trader alters its margin [3, p.22, Fig.27]. For example, a seller's margin is *raised* if *one* condition holds true (i.e., if the last quote was accepted and the seller's current price is less than the price of the current quote); but the margin is *lowered* if either of *two* other possible conditions are true (i.e., if the last quote was an accepted bid and the seller is active and the seller's price is greater than the price of the last quote, *OR* if the last quote was an offer that was accepted and the seller is active and its precise is greater than the price of the last quote). So we could have the genome specify *three* corresponding parameter-value vectors for the buyers and also *three* such vectors for the sellers, i.e. a total of six different vectors for six different cases, which at eight values per vector gives us "ZIP48".

And in a final flourish of parameter-count inflation, let's abandon the use of a mere pair of system-wide global constants c_a and c_r and in place initialise each trader i with its own corresponding "personal" values $c_{a,i}$ and $c_{r,i}$, generated at initialization from the uniform distributions $\boldsymbol{U}[c_{a:min}, c_{a:min} + c_{a:\Delta}]$ and $\boldsymbol{U}[c_{r:min}, c_{r:min} + c_{r:\Delta}]$. This addition of extra parameters still allows solutions involving the old system-wide constant c_a and c_r values to be “discovered” by the GA -- that will happen if increased fitness values are associated with (near-)zero values of $c_{a:\Delta}$ and $c_{r:\Delta}$. So, each of the six parameter-value vectors needs now to specify values not only the six previous system parameters (μ_{min} , μ_{Δ} , β_{min} , β_{Δ}, γ_{min}, and γ_{Δ}) but also the values for the four newly-introduced system parameters $c_{a:min}$, $c_{a:\Delta}$, $c_{r:min}$, and $c_{r:\Delta}$ –– that gives six vectors, each with ten values per vector, hence sixty values for ZIP60. Note, however, that in some of the six vectors the two parameters governing the initial margin value $\mu_i(0)$ do not always contribute to the individual’s fitness: see the source-code in Appendix A for further details.

3.2 Initial (“c6c6”) ZIP60 results

In testing the performance of ZIP60 thus far, all effort has been devoted to exploring the performance of ZIP60 on the dual-shock tests: it is assumed (but not yet empirically verified) that if ZIP60 does better than ZIP8 on the multi-shock tests, then it will also do better in those cases where there are fewer or no market shocks (i.e. in the sort of experiments discussed in Tables 1 and 2).

The same experiment methods were used as described in Section 2, except that the initial population was now composed of randomly generated ZIP60 individuals, rather than ZIP8s. For reasons that will become clear shortly, we will refer to these results as the “c6c6” results. Summary results are shown in Tables 9 and 10 (for all 50 runs, and for the top decile, respectively); and Table 11 shows a summary of how the top-decile ZIP60 scores rate, as percentage improvements over the corresponding top-decile ZIP8 scores. As is clear in Table 11, although the ZIP60 scores are generally better than ZIP8, sometimes over 20% better, they can also occasionally be significantly worse. This is reflected in the standard deviation of the percentage improvement, which is pretty much identical to the mean percentage improvement itself; not a particularly encouraging result.

M	EM:μ	EM:σ	FM:μ	FM:σ	Sig?	Qs:μ	Qs:σ	EM/FM?
1-2-1	3.948	0.209	3.908	0.197	N	0.566	0.097	=
2-1-2	3.915	0.185	3.903	0.173	N	0.510	0.061	=
2-3-2	2.811	0.213	2.890	0.217	N	0.539	0.095	=
3-2-3	3.567	0.299	3.508	0.267	N	0.573	0.106	=
1-2-3	3.812	0.261	3.748	0.233	N	0.637	0.086	=
3-2-1	3.726	0.271	3.734	0.312	N	0.533	0.076	=
1-4-1	2.966	0.323	3.018	0.216	N	0.336	0.110	=
4-1-4	2.593	0.253	2.628	0.228	N	0.339	0.086	=
4-3-4	3.231	0.149	3.151	0.142	Y	0.475	0.129	FM
3-4-3	3.317	0.414	3.348	0.325	N	0.359	0.113	=
1-4-3	2.966	0.307	3.109	0.243	N	0.324	0.094	=
3-4-1	3.190	0.296	3.284	0.276	N	0.377	0.112	=
4-3-2	2.188	0.267	2.359	0.308	N	0.318	0.130	=
2-3-4	2.953	0.250	2.946	0.234	N	0.568	0.072	=
4-1-2	2.696	0.341	2.654	0.289	N	0.446	0.126	=
2-1-4	3.168	0.230	3.140	0.222	N	0.476	0.093	=
2-4-2	4.461	0.291	4.478	0.297	N	0.469	0.055	=
4-2-4	4.066	0.207	3.990	0.198	N	0.470	0.061	=

Table 9: *Results from the "c6c6" ZIP60 experiment, n=50, as in Table 7.*

M	**EM:μ**	**EM:σ**	**FM:μ**	**FM:σ**	**Sig?**	**Qs:μ**	**Qs:σ**	**EM/FM?**
1-2-1	3.669	0.046	3.615	0.078	N	0.597	0.048	=
2-1-2	3.607	0.055	3.736	0.102	N	0.492	0.056	=
2-3-2	2.456	0.148	2.587	0.092	N	0.410	0.149	=
3-2-3	3.155	0.091	3.117	0.078	N	0.546	0.122	=
1-2-3	3.450	0.097	3.450	0.049	N	0.655	0.021	=
3-2-1	3.316	0.098	3.212	0.141	N	0.497	0.041	=
1-4-1	2.382	0.073	2.695	0.053	Y	0.247	0.050	EM
4-1-4	2.199	0.159	2.361	0.040	Y	0.314	0.028	EM
4-3-4	2.978	0.058	2.939	0.051	N	0.336	0.084	=
3-4-3	2.629	0.100	2.788	0.070	N	0.300	0.081	=
1-4-3	2.419	0.110	2.686	0.103	Y	0.253	0.085	EM
3-4-1	2.671	0.078	2.821	0.126	N	0.302	0.147	=
4-3-2	1.578	0.077	1.881	0.094	Y	0.504	0.014	EM
2-3-4	2.573	0.076	2.610	0.064	N	0.569	0.054	=
4-1-2	2.123	0.242	2.158	0.249	N	0.496	0.078	=
2-1-4	2.853	0.056	2.833	0.063	N	0.441	0.086	=
2-4-2	3.854	0.127	3.922	0.087	N	0.437	0.040	=
4-2-4	3.731	0.058	3.656	0.057	N	0.478	0.052	=

Table 10: *Results from the "c6c6" ZIP60 experiment, top decile n=5, as in Table 8.*

M1-2-1	14.696
M2-1-2	6.855
M2-3-2	15.461
M3-2-3	17.093
M1-2-3	17.189
M3-2-1	15.820
M1-4-1	24.816
M4-1-4	2.046
M4-3-4	-7.368
M3-4-3	23.628
M1-4-3	23.680
M3-4-1	23.461
M4-3-2	26.775
M2-3-4	11.626
M4-1-2	26.759
M2-1-4	12.159
M2-4-2	-2.037
M4-2-4	-20.350
Average	**12.906**
Stdev	**12.879**

Table 11: *Percentage score of top-decile "c6c6" ZIP60 over top-decile ZIP8. Although the mean improvement is approx 13%, the standard deviation on that improvement is almost the same value. Average of the improvements (positive scores only) is 17.5%, but average of the negative scores (declines) is -9.9%.*

3.3 Evolutionary Control of Dimensionality ("c1c6" and "c1c1") ZIP60 results

Examination of the output from the "c6c6" experiments indicated that there was no obvious flaw in the code (i.e., no bug) that was preventing the ZIP60 system from reaching solutions that correspond to ZIP8 solutions. But there are certainly points within the ZIP60 genome-space that correspond perfectly to ZIP8 solutions: where the vectors for the six cases in a ZIP60 genome are all the same, that ZIP60 genome is functionally equivalent to the single-case ZIP8 system. Thus, in those "c6c6" experiments where the average ZIP60 score is significantly worse than the corresponding ZIP8 score, it seems likely that evolutionary search is somehow failing to find the successful ZIP8-style genomes. That can only be regarded as a weakness in the system, but the system component that is fault is the GA search, not the ZIP traders.

To remedy this weakness, the ZIP genome encoding was extended, allowing the number of cases (1, 2, 4, or 6, as discussed in Section 3.1) to be specified on the genome. The genome is still a set of six vectors, with three of the vectors being used for buyers, and the other three being used for sellers. If the number of cases is set to one, then all six parameter-vectors are set to be identical, by copying the values from the first vector into the remaining five. If the number of cases is set to two, then the three buyer vectors are set to be identical copies of each other, as are the three seller vectors; and if the number of cases is set to be six, then the three buyer and three seller vectors can all hold different numeric values. The code in Appendix A illustrates how this is achieved. The motivating hypothesis was that the GA's evolutionary search would be more successful if it could start by first simply optimizing the 1-case genome, and then only once all the values are approximately correct, successive refinements could be introduced by the GA if, for example, a 1-case individual mutated to become a high-case individual, thereby decoupling its genome-values across the different cases; but such mutants would only be retained in the population if the mutation that increases the number of cases is associated with higher fitness.

Two new sets of ZIP60 experiments were performed. In the first, the population was initialised with individuals that had a randomly-assigned value for the number of cases on their genome, with the values 1, 2, 4, and 6 being equally probable. This is the initialization we refer to here as "c1c6". In the second, all individuals in the initial population were set to have 1-case genomes. These are referred to here as the "c1c1" initializations. And so now it should be clear why the results presented in the previous section were referred to as "c6c6" – in those experiments, each individual in the initial population is a 6-case genome.

Tables 12 and 13 show summary comparisons of the c1c1 and c1c6 results, and Table 14 expresses these results as percentage relative improvements. As can be seen from Table 14 the c1c1 initialization method gives only a small average improvement over c1c6, and in fact gives relative average improvement results very slightly worse than the original c6c6, but (crucially) when compared to the original ZIP8 scores, the c1c1 ZIP60 results show an average relative improvement that is higher than the c6c6 score, and also with a much lower standard deviation – in fact, the c1c1 ZIP60 results improve on ZIP8 in *every* experiment. From this, we conclude that the c1c1 initialization method is to be preferred when using a simple GA to optimize a ZIP60 market.

	C1C6				**C1C1**					
M	**EM:μ**	**EM:σ**	**Qs:μ**	**Qs:σ**	**EM:μ**	**EM:σ**	**Qs:μ**	**Qs:σ**	**Sig?**	**Best**
1-2-1	4.366	0.534	0.621	0.112	4.055	0.462	0.567	0.079	N	c6c6
2-1-2	4.302	0.456	0.572	0.091	4.185	0.521	0.574	0.096	N	c6c6
2-3-2	3.486	0.645	0.689	0.070	3.439	0.580	0.696	0.084	N	c6c6
3-2-3	3.758	0.486	0.698	0.064	3.667	0.487	0.684	0.082	N	c6c6
1-2-3	3.827	0.423	0.764	0.073	3.753	0.394	0.765	0.073	N	c1c1
3-2-1	4.057	0.403	0.583	0.087	4.055	0.462	0.567	0.079	N	c6c6
1-4-1	3.456	0.451	0.178	0.071	3.342	0.422	0.162	0.060	N	c6c6
4-1-4	2.645	0.305	0.242	0.069	2.561	0.277	0.258	0.083	N	c1c1
4-3-4	3.092	0.366	0.490	0.087	2.924	0.322	0.469	0.069	N	c1c1
3-4-3	4.623	0.137	0.153	0.042	3.727	0.570	0.151	0.040	N	c6c6
1-4-3	3.397	0.506	0.146	0.059	3.234	0.506	0.136	0.058	N	c6c6
3-4-1	3.675	0.605	0.179	0.037	3.592	0.485	0.189	0.042	N	c6c6
4-3-2	2.585	0.385	0.241	0.077	2.484	0.344	0.232	0.089	N	c6c6
2-3-4	3.441	0.608	0.685	0.073	3.286	0.460	0.690	0.067	N	c6c6
4-1-2	3.152	0.263	0.261	0.065	3.042	0.262	0.266	0.069	N	c6c6
2-1-4	3.737	0.556	0.534	0.092	3.541	0.484	0.536	0.074	N	c6c6
2-4-2	4.684	0.368	0.405	0.065	4.338	0.524	0.399	0.059	N	c1c1
4-2-4	3.849	0.533	0.459	0.052	3.896	0.477	0.472	0.045	N	c1c6

Table 12: *Results from EM experiments with ZIP60 initialized in styles "c1c6" and "c1c1"; results from all n=50 experiments. From the leftmost, the columns are: the market supply/demand sequence identifier label; the mean score for the c1c6 experiments; the standard deviation for the c1c6 scores; the mean Qs value for the c1c6 population; the standard deviation on that mean Qs; the mean fitness for the c1c1 experiments; the standard deviation for the c1c1 fitness; the mean Qs value for the c1c1 population; the standard deviation on that mean Qs; outcome of the RRO significance test comparing the EM scores of the c1c6 and c1c1 experiments ("N" signifies no significant difference detected at 1% level); and the final column shows the best-scoring method when compared across the n=50 results.*

	C1C6				**C1C1**					
M	**EM:μ**	**EM:σ**	**Qs:μ**	**Qs:σ**	**EM:μ**	**EM:σ**	**Qs:μ**	**Qs:σ**	**Sig?**	**Best**
1-2-1	3.486	0.154	0.688	0.073	3.411	0.039	0.552	0.060	N	c1c1
2-1-2	3.613	0.111	0.458	0.087	3.420	0.043	0.445	0.142	N	c6c6
2-3-2	2.556	0.105	0.624	0.072	2.518	0.062	0.634	0.106	N	c1c1
3-2-3	3.100	0.092	0.683	0.061	3.055	0.077	0.716	0.061	N	c1c1
1-2-3	3.202	0.103	0.798	0.030	3.136	0.084	0.798	0.030	N	c1c1
3-2-1	3.405	0.138	0.562	0.062	3.411	0.039	0.552	0.060	N	c6c6
1-4-1	2.786	0.248	0.133	0.077	2.624	0.063	0.167	0.023	N	c6c6
4-1-4	2.095	0.144	0.357	0.031	2.136	0.078	0.262	0.120	N	c1c6
4-3-4	2.527	0.074	0.447	0.014	2.501	0.037	0.434	0.058	N	c1c1
3-4-3	2.863	0.149	0.117	0.021	2.835	0.150	0.148	0.016	N	c6c6
1-4-3	2.560	0.091	0.137	0.036	2.470	0.063	0.092	0.024	N	c1c1
3-4-1	2.778	0.135	0.183	0.025	2.879	0.105	0.159	0.013	N	c6c6
4-3-2	1.905	0.125	0.295	0.092	1.972	0.114	0.266	0.107	N	c1c6
2-3-4	2.547	0.051	0.662	0.115	2.503	0.106	0.697	0.097	N	c6c6
4-1-2	2.634	0.158	0.275	0.059	2.633	0.034	0.242	0.018	N	c1c1
2-1-4	2.828	0.089	0.462	0.063	2.809	0.098	0.502	0.073	N	c1c1
2-4-2	4.045	0.215	0.439	0.064	3.461	0.025	0.326	0.029	Y	c1c1
4-2-4	2.924	0.131	0.472	0.025	2.905	0.061	0.468	0.009	N	c6c6

Table 13: *Top-decile (n=5) results, column labels as per Table 12.*

M	c1c1 on c1c6	c1c1 on c6c6	c1c1 on Zip8
1-2-1	2.161	7.032	20.695
2-1-2	5.337	5.186	11.686
2-3-2	1.487	-2.533	13.320
3-2-3	1.450	3.170	19.721
1-2-3	2.051	9.078	24.707
3-2-1	-0.169	-2.859	13.413
1-4-1	5.813	-10.135	17.196
4-1-4	-1.954	2.859	4.847
4-3-4	1.009	16.009	9.820
3-4-3	0.969	-7.818	17.657
1-4-3	3.516	-2.119	22.063
3-4-1	-3.639	-7.772	17.512
4-3-2	-3.535	-24.933	8.518
2-3-4	1.716	2.685	13.999
4-1-2	0.003	-24.032	9.157
2-1-4	0.684	1.522	13.495
2-4-2	14.424	10.200	8.371
4-2-4	0.665	22.147	6.305
Average	**1.777**	**-0.128**	**14.027**
Stdev	**4.018**	**12.077**	**5.663**

Table 14: *Summary comparison data of c1c1 ZIP60 with initialization methods c1c6, c6c6, and the original ZIP8: see text for discussion.*

Figure 5 illustrates the underlying evolutionary dynamics in one set of 50 c1c1-ZIP60 experiments. Each of the four lines on the graph indicates, for each generation, the number of experiments in which the elite individual's genotype is either 1-case, 2-case, 4-case, or 6-case. As can be seen, there is an initial rapid fall in the number of 1-case elite genotypes, as the number of elite genotypes that are 4-case and 6-cases progressively rises. At approximately generation *g*=50, the number of 4-case elite genotypes levels off and then starts a steady decline which lasts until it has decayed to a noise-level at approximately *g*=300, but the number of 6-case genotypes continues to rise, and by g=300 pretty much all (49 of the 50) experiment have an elite 6-case genotype.

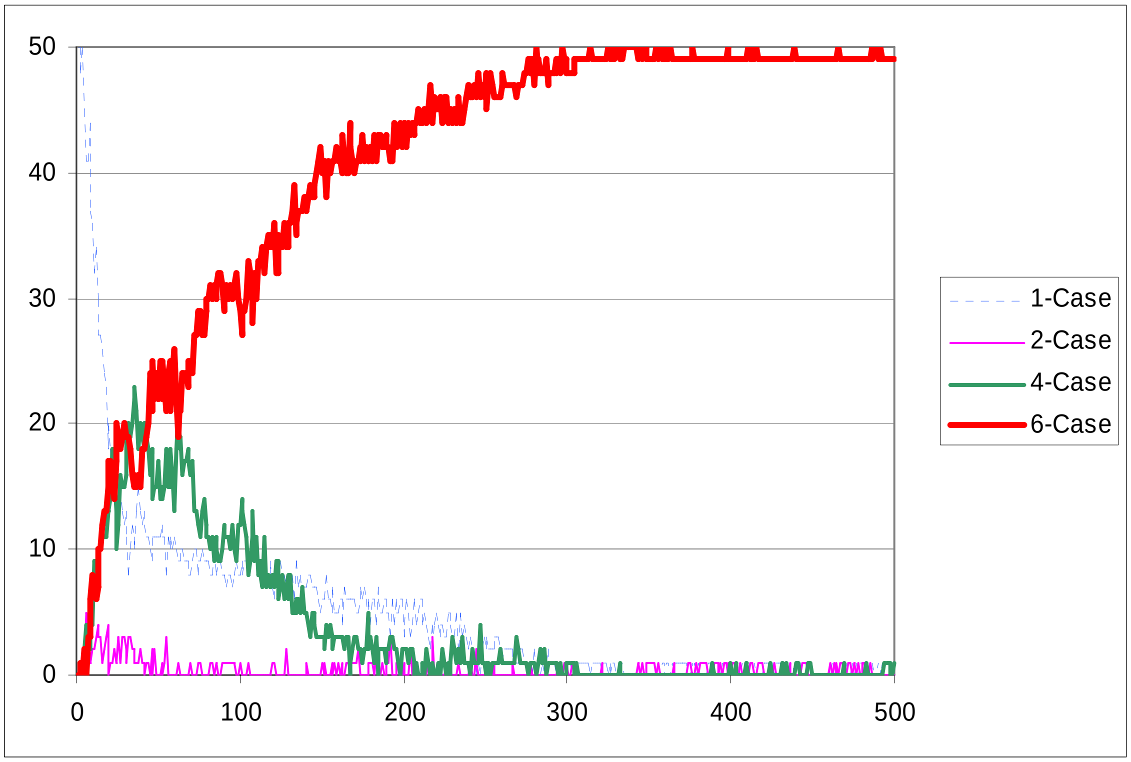


Figure 5: *Evolutionary dynamics in 50 repetitions of the M1-2-1 c1c1-ZIP60 experiment (with different random seeds). The four data series show, for each generation, the number of experiments in which the elite individual genotype has an allele for 1-case, 2-case, 4-case, or 6-case on its n_cases locus. Because this is a c1c1 initialization, the number of 1-case genotypes is initially 50, and for the three other types the number is zero. The number of 4-case and 6-case genotypes rises steadily for the first 50 generations, but thereafter the number of 4-cases goes into decline while the number of 6-cases continues to increase to saturation at generation 300.*

3.4 Discussion

One observation, that should be explored in more depth in future research, is that the ZIP60 results typically only very rarely discover hybrid values of Q_s that yield overall market dynamics that are significantly better than those of the corresponding fixed-market CDA Q_s=0.5 experiments (despite the final evolved Q_s values varying quite widely). It seems likely that this is an indication that the earlier results showing evolved hybrid auction mechanisms are to some extent artefacts of the lack of sophistication in the ZIP8 traders that were used in those studies. A counter to this is that Byde [2] presented results from applying similar GA-search for designs for hybrid sealed-bid auctions, where the GA found hybrid solutions to be preferable to the traditional first-price and second-price sealed bid auctions, and those results were independent of the sophistication of the traders in the market.

6. Conclusions

From the data reported here, the indications are that the ZIP60 variant of ZIP is a genuine improvement on the original ZIP8, so long as care is taken in the construction of the GA. Specifically, the care required is that the initial population be seeded with ZIP8s, and that expansion of the dimensionality of the search-space is allowed only when it increases the fitness of the individuals concerned.

Acknowledgements

Thanks to members of the HP Labs Biologically-Inspired Complex Adaptive Systems (BICAS) research group for valuable discussions. See www.hpl.hp.com/research/bicas.

Endnote

Author's address after May 9th, 2005:
Complex Risk Group, Deutsche Bank Global Markets,
1 Great Winchester Street, London EC2N 2DB. Email david.cliff@db.com

Copies of this report should be available online from the HP Labs Technical Reports Website:
http://www.hpl.hp.com/techreports/2005/HPL-2005-85.html

References

[1] Bullock, S. (1999), "Are artificial mutation biases unnatural?" in: Floreano, D., Nicoud, J.-D. & Mondada, F. (eds) *Advances in Artificial Life: Fifth European Conference (ECAL99)*, pp. 64-73. Springer-Verlag. 1999.

[2] Byde, A. (2002) "Applying Evolutionary Game Theory to Auction Mechanism Design". Accepted for presentation at the *2003 ACM Conference on E-Commerce*. Also available as Hewlett-Packard Laboratories Technical Report HPL-2002-321.

[3] Cliff, D. (1997), "Minimal-intelligence agents for bargaining behaviours in market environments". Hewlett-Packard Laboratories Technical Report HPL-97-91.

[4] Cliff, D. (1998), "Genetic optimization of adaptive trading agents for double-auction markets" in *Proceedings of Computational Intelligence in Financial Engineering (CIFEr) 1998.* IEEE/IAFE/Informs (preprint proceedings), pp.252-258, 1998.

[5] Cliff, D. (2001), "Evolutionary optimization of parameter sets for adaptive software-agent traders in continuous double-auction markets". Presented at the Artificial Societies and Computational Markets (ASCMA98) workshop at the Second International Conference on Autonomous Agents, Minneapolis/St. Paul, May 1998. Also available as HP Labs Technical Report HPL-2001-99.

[6] Cliff, D. (2002), "Evolution of market mechanism through a continuous space of auction-types". Presented at *Computational Intelligence in Financial Engineering* session at *CEC2002*, Hawaii, May 2002. Also available as Hewlett-Packard Laboratories Technical Report HPL-2001-326.

[7] Cliff, D. (2002), "Evolution of market mechanism through a continuous space of auction-types II: Two-sided auction mechanisms evolve in response to market shocks". Presented at *Agents for Business Automation* session at *IC2002*, Las Vegas, June 2002. In: *Proceedings of the International Conference on Internet Computing IC02*, Volume III, edited by H.R. Arabnia and Y. Mun. CSREA Press, pp.682-688. Also available as Hewlett-Packard Laboratories Technical Report HPL-2002-128.

[8] Cliff, D. (2002) "Visualizing Search-Spaces for Evolved Hybrid Auction Mechanisms". Presented at the *Beyond Fitness: Visualizing Evolution* workshop at the 8th International Conference on the Simulation and Synthesis of Living Systems (ALifeVIII) conference, Sydney, December 2002. Also available as Hewlett-Packard Laboratories Technical Report HPL-2002-291.

[9] Cliff, D. (2002) "Evolution of Market Mechanism Through a Continuous Space of Auction-Types III: Multiple Market Shocks Give Convergence Toward CDA". Hewlett-Packard Laboratories Technical Report HPL-2002-312.

[10] Cliff, D., Walia, V., & Byde, A. (2002) "Evolved Hybrid Auction Mechanisms in Non-ZIP Trader Marketplaces". Accepted for presentation at the *International Conference on Computational Intelligence for Financial Engineering (CIFEr03)*, Hong Kong, March 2003. Also available as Hewlett-Packard Laboratories Technical Report HPL-2002-247.

[11] Cliff, D. (2003) Explorations in evolutionary design of online auction market mechanisms. *Electronic Commerce Research and Applications* 2(2):162-175, 2003. Also available as Hewlett-Packard Laboratories Technical Report HPL-2003-80.

[12] Das, R., Hanson, J., Kephart, J., & Tesauro, G. (2001), "Agent-human interactions in the continuous double auction" *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-01)*, 2001

[13] Feltovich, N. (2003), "Nonparametric tests of differences in medians: comparison of the Wilcoxon-Mann-Whitney and robust rank-order tests", *Experimental Economics* 6 (2003), pp. 273-297.

[14] Feltovich, N. (forthcoming) "Critical values for the robust rank-order test," forthcoming, *Communications in Statistics.*

[15] Friedman, D., & Rust, J. (eds) (1993), *The Double Auction Market: Institution, Theories, and Evidence*. Addison Wesley.

[16] Gode, D. & Sunder, S. (1993), "Allocative efficiency of markets with zero-intelligence traders" *Journal of Political Economy* **101**:119-137, 1993.

[17] Goldberg, D. E. (1989) *Genetic Algorithms: In Search, Optimization, and Machine Learning.* Reading, MA: Addison Wesley.

[18] Mitchell, M., (1998). *An Introduction to Genetic Algorithms.* MIT Press.

[19] New York Stock Exchange (2002), *Stock Market Activity* report available at: http://www.nyse.com/pdfs/02_STOCKMARKETACTIVITY.pdf

[20] Phelps, S., McBurney, P., Parsons, S., & Sklar, E., (2002) "Co-evolutionary auction mechanism design: a Preliminary Report", Technical Report ULCS-02-019, Department of Computer Science, University of Liverpool, 2002.

[21] Qin, Z., (2002) *Evolving Marketplace Designs by Artificial Agents*. MSc Thesis, Computer Science Department, Bristol University, September 2002.

[22] Robinson, N., (2002) *Evolutionary Optimization of Market-Based Control Systems for Resource Allocation in Compute Farms.* MSc Thesis, University of Sussex School of Cognitive and Computing Sciences, September 2002.

[23] Rumelhart, D.E., Hinton, G.E., and Williams, R.J. (1986) "Learning internal representations by error propagation", in *Parallel Distributed Processing: Explorations in the Microstructures of Cognition*, Volume 1, edited by D. E. Rumelhart and J. L. McClelland (Cambridge, MA: MIT Press), pp. 318-362, 1986.

[24] Siegel, S., & Castellan, N. J., (1988), *Nonparametric Statistics for the Behavioral Sciences*. McGraw Hill.

[25] Smith, V. (1962), "Experimental study of competitive market behavior" *Journal of Political Economy* **70**:111-137, 1962.

[26] Walia, V., (2002) *Evolving Market Design*, MSc Thesis, School of Computer Science, University of Birmingham, September 2002.

[27] Wilson, S.W., (1995) "Classifier Fitness Based on Accuracy" *Evolutionary Computation*, **3**(2):149-175, 1995.

[28] Wurman, P.R., Wellman, M.P., & Walsh, W.E. (2001) "A Parameterization of the Auction Design Space" *Games and Economic Behavior* **35**:304-338, 2001.

Appendix A: The C Source Code

The following appendices present the C source code for the ZIP60 system, used to generate the experiment results data presented in this report. The code is presented here for illustration/education purposes only, and is issued without any warranties (expressed, implied, or otherwise) whatsoever. It is intended to be read in comparison with the code shown in the appendices to the original ZIP technical report (Cliff, 1997): it is assumed here that the reader is familiar with that original ZIP C Code, and so the revised code is presented here without commentary. In places, comments in the code show the change history (six-digit numbers in those comments represent dates in *yymmdd* format). Files are listed in alphabetical order, with *.h* header files preceding their paired *.c* source code files. The file pairs *random.h/random.c*, *sd.h/sd.c,* and *tdat.h/tdat.c* remain substantially unchanged from the versions published in 1997, and so are not reproduced again here.

A.1 agent.h

```
*agent.h: general global constants and structures
Dave Cliff
August 1996*/

#define NULL_EQ -1 /*signals no equilibrium*/

/*DC added020526:number of different "cases" encoded for on genome*/
#define MAX_N_CASES 6

/*symbolic constants for agent type, shout type, and whether shout is accepted
or rejected*/

#define BUY 1
#define SELL 0
/// DC 020530: BUY and SELL used as mu0[c] indices in initialisation.
#define BID 1
#define OFFER 0
#define DEAL 1
#define NO_DEAL 0
#define END_DAY 2

typedef struct an_agent{
    int     job;     /*BUYing or SELLing*/
    int     active;  /*still in the market?*/
    int     n;       /*number of deals done*/
    int     willing; /*want to make a trade at this price?*/
    int     able;    /*allowed to trade at this price?*/
    Real    limit;   /*the bottom-line price for this agent*/
    Real    profit;  /*profit coefficient in determinining bid/offer price*/
    Real    beta[MAX_N_CASES];    /*coeff for changing profit over time (learning rate)*/
    Real    momntm[MAX_N_CASES];  /*momentum in changing profit*/
    Real    c_a[MAX_N_CASES];     /*absolute offset in target price*/
    Real    c_r[MAX_N_CASES];     /*relative offset in target price*/
    Real    last_d;  /*last change*/
    Real    price;   /*what the agent will actually bid*/
    Real    quant;   /*how much of this commodity*/
    Real    bank;    /*how much money this agent has in the bank*/
    Real    a_gain;  /*actual gain*/
    Real    t_gain;  /*theoretical gain*/
    Real    sum;     /*in determining average reward*/
    Real    avg;     /*average reward*/
} Agent;

void set_price(Agent *);
void shout_update(int deal_type,int status,
                 int n_sell,Agent sellers[],int n_buy,Agent buyers[],
                 Real price,int verbose);
void buy_init(Agent b[],Real bmin[],Real bspr[],Real gmin[],Real gspr[],
              Real pmin[],Real pspr[],Real camn[],Real casp[],Real crmn[],
              Real crsp[],int verbose);
void sell_init(Agent s[],Real bmin[],Real bspr[],Real gmin[],Real gspr[],
               Real pmin[],Real pspr[],Real camn[],Real casp[],Real crmn[],
               Real crsp[],int verbose);
int willing_trade(Agent *a,Real price);
void profit_alter(Agent *a,Real price,int c,int verbose);
//c is the case-i.d. DC added 020530
```

A.2 agent.c

```
/* agent.c: defines an agent, how it adapts, etc.\\
Dave Cliff\\
Aug 1996*/

///updated May 2002 to allow for different params in different cases.

#include <math.h>
#include <stdio.h>
#include "random.h"
#include "max.h"
#include "agent.h"

#define BONUS 0.00


/*set-price: set the price of an agent from its limit and profit values*/
void set_price(Agent *a)
{ a->price=(a->limit)*(1+a->profit);
  /*normalise to one-cent precision*/
  a->price=(floor((a->price*100)+0.5))/100;
}


/*agent-init: initialise the common elements of an agent (buyer or seller)*/
void agent_init(Agent *a,
                Real betamin[MAX_N_CASES],Real betaspread[MAX_N_CASES],
                Real gammamin[MAX_N_CASES],Real gammaspread[MAX_N_CASES],
                Real camin[MAX_N_CASES],Real caspread[MAX_N_CASES],
                Real crmin[MAX_N_CASES],Real crspread[MAX_N_CASES],int verbose)
{ int c=0;
  Real cspread;
  /* if MAX_N_CASES=1 then c_spead=0; else _spread=1*/

  a->bank=0.0;
  a->n=0;
  a->sum=0.0;
  a->last_d=0.0;

  for(c=0;c<MAX_N_CASES;c++)
  {
    a->beta[c]=betamin[c]+randval(betaspread[c]);
    if(a->beta[c]>1.0) a->beta[c]=1.0;

    a->momntm[c]=gammamin[c]+randval(gammaspread[c]);
    if(a->momntm[c]>1.0) a->momntm[c]=1.0;

    if(MAX_N_CASES==1) cspread=0.0; else cspread=1.0;

    a->c_a[c]=camin[c]+cspread*randval(caspread[c]);
    a->c_r[c]=crmin[c]+cspread*randval(crspread[c]);


    if(verbose)
    { fprintf(stdout,"prof=%+5.3f beta[%1d]=%5.3f mom[%1d]=%5.3f ca[%1d]=%5.3f
                  cr[%1d]=%5.3f bank=%5.2f\n",
              a->profit,c,a->beta[c],c,a->momntm[c],a->c_a[c],a->c_r[c],a->bank);
    }
  }

  a->active=1;

}


/*buy-init: initialize the buyers*/
void buy_init(Agent b[MAX_AGENTS],
              Real bmin[MAX_N_CASES],Real bspr[MAX_N_CASES],
              Real gmin[MAX_N_CASES],Real gspr[MAX_N_CASES],
              Real pmin[MAX_N_CASES],Real pspr[MAX_N_CASES],
              Real camn[MAX_N_CASES],Real casp[MAX_N_CASES],
              Real crmn[MAX_N_CASES],Real crsp[MAX_N_CASES],
              int verbose)
{ int a;

  for(a=0;a<MAX_AGENTS;a++)
  { b[a].job=BUY;
    if(MAX_N_CASES>1)
    { b[a].profit=-1.0*(pmin[BUY]+randval(pspr[BUY])); }
    else //MAX_N_CASES==1
    { b[a].profit=-1.0*(pmin[0]+randval(pspr[0])); }
    if(b[a].profit<-1.0) b[a].profit=-1.0;
    if(verbose) fprintf(stdout,"B%2d ",a);
    agent_init(b+a,bmin,bspr,gmin,gspr,camn,casp,crmn,crsp,verbose);
  }
}

/*sell-init: initialize the sellers*/
void sell_init(Agent s[MAX_AGENTS],
               Real bmin[MAX_N_CASES],Real bspr[MAX_N_CASES],
               Real gmin[MAX_N_CASES],Real gspr[MAX_N_CASES],
               Real pmin[MAX_N_CASES],Real pspr[MAX_N_CASES],
               Real camn[MAX_N_CASES],Real casp[MAX_N_CASES],
               Real crmn[MAX_N_CASES],Real crsp[MAX_N_CASES],
               int verbose)
{ int a;

  for(a=0;a<MAX_AGENTS;a++)
  { s[a].job=SELL;
    if(MAX_N_CASES>1)
    { s[a].profit=pmin[SELL]+randval(pspr[SELL]); }
    else //MAX_N_CASES==1
    { s[a].profit=pmin[0]+randval(pspr[0]); }
    if(verbose) fprintf(stdout,"S%2d ",a);
    agent_init(s+a,bmin,bspr,gmin,gspr,camn,casp,crmn,crsp,verbose);
  }
}


/*willing-trade: is an agent willing to trade at given price?*/
int willing_trade(Agent *a,Real price)
{ if(a->job==BUY)
  { /*willing to buy at this price?*/
    if((a->active)&&(a->price>=price))
    { a->willing=1; }
    else
    { a->willing=0; }
  }

  else
  { /*willing to sell at this price?*/
    if((a->active)&&(a->price<=price))
    { a->willing=1; }
    else
    { a->willing=0; }
  }
  return(a->willing);
}


/*profit-alter: update profit margin on basis of sale price*/
/*using Widrow-Hoff style update with learning rate $\beta$.*/
void profit_alter(Agent *a,Real price,int c,int verbose)
{ Real diff,change,newprofit,gamma;

  if(c>=MAX_N_CASES)
  { fprintf(stderr,
            "FAIL; MAX_N_CASES=%d but profit_alter called with c=%d\n",
            MAX_N_CASES,c);
    exit(-1);
  }

  if(verbose) fprintf(stdout,"lim=%5.3f prof=%5.3f price=%5.2f",
                      a->limit,a->profit,a->price);

  gamma=a->momntm[c];
  diff=(price-(a->price));
  change=((1.0-gamma)*(a->beta[c])*diff)+(gamma*(a->last_d));

  if(verbose) fprintf(stdout," last_d=%5.3f diff=%5.2f chng=%+5.3f",
                               a->last_d,diff,change);

  a->last_d=change;

  /*set new prices by altering profit margin*/
  newprofit=((a->price+change)/a->limit)-1.0;

  if(a->job==SELL)
  { if(newprofit>0.0) a->profit=newprofit; }
  else
  { if(newprofit<0.0) a->profit=newprofit; }

  set_price(a);

  if(verbose)
  { fprintf(stdout," nu_prof=%5.3f nu_price=%5.2f",a->profit,a->price);}
}



/*shout-update: update strategies of buyers and sellers after a shout*/
void shout_update(int deal_type,int status,int n_sell,
                 Agent sellers[],int n_buy,Agent buyers[],Real price,
                 int verbose)
{ int b,s,c;
  Real target_price;
  /*any seller whose price is less than or equal to the deal price raises
  profit margin*/
  /*(this is an attempt to increase profits next time around)*/

  for(s=0;s<n_sell;s++)
  { if(verbose) fprintf(stdout,"S%02d(%d) ",s,sellers[s].active);

    if(status==DEAL)
    { if(sellers[s].price<=price)
      { /*could get more? -- try raising margin*/
        c=0; /*ALWAYS case 0*/
        target_price=(price*(1.0+randval(sellers[s].c_r[c])))+randval(sellers[s].c_a[c]);
        profit_alter(sellers+s,target_price,c,verbose);
      }

      else
      { /*wouldn't have got this deal, so mark the price down*/
        if( (deal_type==BID) &&
            (!willing_trade(sellers+s,price)) &&
            (sellers[s].active)
          )
        {
          c=1;
          target_price=(price*(1.0-randval(sellers[s].c_r[c])))-
                       randval(sellers[s].c_a[c]);
          profit_alter(sellers+s,target_price,c,verbose);
        }
      }
    }

    else /*NO DEAL*/
    { if(deal_type==OFFER)
      if((sellers[s].price>=price)&&(sellers[s].active))
      { /*would have asked for more and lost the deal, so reduce profit*/
        c=2;
        target_price=(price*(1.0-randval(sellers[s].c_r[c])))-randval(sellers[s].c_a[c]);
        profit_alter(sellers+s,target_price,c,verbose);
      }
    }
    if(verbose)fprintf(stdout,"\n");
  }

  for(b=0;b<n_buy;b++)
  { if(verbose) fprintf(stdout,"B%02d(%d) ",b,buyers[b].active);

    if(status==DEAL)
    { if(buyers[b].price>=price)
      { /*could get lower price? -- try raising margin (i.e. cutting price)*/
        switch(MAX_N_CASES)
        c=3;
        target_price=(price*(1.0-randval(buyers[b].c_r[c])))-randval(buyers[b].c_a[c]);
        profit_alter(buyers+b,target_price,c,verbose);
      }

      else
      { /*wouldn't have got this deal, so mark the price up (reduce profit)*/
        if( (deal_type==OFFER) &&
            (!willing_trade(buyers+b,price)) &&
            (buyers[b].active)
          )
        { c=4;
          target_price=(price*(1.0+randval(buyers[b].c_r[c])))+randval(buyers[b].c_a[c]);
          profit_alter(buyers+b,target_price,c,verbose);
        }
      }
    }

    else /*NO-DEAL*/
    { if(deal_type==BID)
      if((buyers[b].price<=price)&&(buyers[b].active))
      { /*would have bid less and also lost the deal, so reduce profit*/
        c=5;
        target_price=(price*(1.0+randval(buyers[b].c_r[c])))+randval(buyers[b].c_a[c]);
        profit_alter(buyers+b,target_price,c,verbose);
      }
    }
    if(verbose)fprintf(stdout,"\n");
  }
}
```

A.3 ddat.h

```
/*ddat.h: header for ddat.c routines
Dave Cliff, Sept 1996*/


/*datatype for Real sum and sum of squares, used in calculuating mean and s.d.*/
typedef struct real_stat{
  Real sum,sumsq;
  int n;
} Real_stat;

/*data and stats for a day's trading*/
typedef struct day_data{
  Real_stat alpha; /*Smith's alpha*/
  Real_stat quant; /*Quantity*/
  Real_stat effic; /*Efficiency*/
  Real_stat price; /*price*/
  Real_stat pdisp; /*profit dispersal*/
  Real_stat volty; /*transaction price volatility*/
} Day_data;


/*ddat-init: initialise daily data*/
void ddat_init(Day_data *);

/*ddat-update: update daily data*/
void ddat_update(Day_data *,int,Real,Real,Real,Real,Real,int);

/*ddat-xgraph: plot the daily stats in xgraph format*/
void xg_daily_graph(Day_data dd[],int,int,char *);
```

A.4 ddat.c

```
/*ddat.c: code for handling data/stats compiled at end of each trading day
Dave Cliff, Sept 1996*/

#include <math.h>
#include <stdio.h>
#include <string.h>

#include "random.h"
#include "ddat.h"
#include "max.h"

#define SMALLREAL 0.0000001 /*used to dodge rounding errors on sqrt */

#define DD_ALPHA 0
#define DD_QUANT 1
#define DD_EFFIC 2
#define DD_PRICE 3
#define DD_PDISP 4
#define DD_VOLTY 5

/*rstat-zero: set everything to zero in one Real-stat structure*/
void rstat_zero(Real_stat *r)
{ r->sum=0.0;
  r->sumsq=0.0;
  r->n=0;
}

/*ddat-init: initialise day data*/
void ddat_init(Day_data *ddat)
{
  rstat_zero(&(ddat->alpha));
  rstat_zero(&(ddat->quant));
  rstat_zero(&(ddat->effic));
  rstat_zero(&(ddat->price));
  rstat_zero(&(ddat->pdisp));
  rstat_zero(&(ddat->volty));
}


/*ddat-update: update day data. */
void ddat_update(Day_data *dd,int n_deals,Real sum_price,
                 Real alpha,Real pdisp,Real effic,Real pdiff,int verbose)
{ Real v;
  if(verbose) fprintf(stdout,"ddat_update n=%d s_p=%f a=%f pdis=%f e=%f pdif=%f",
                      n_deals,sum_price,alpha,pdisp,effic,pdiff);
  if(n_deals>0)
  { (dd->price.sum)+=(sum_price/n_deals);
    (dd->price.sumsq)+=((sum_price/n_deals)*(sum_price/n_deals));
    (dd->price.n)++;

     v=sqrt(pdiff/n_deals); /*root mean square difference*/
    (dd->volty.sum)+=v;
    (dd->volty.sumsq)+=(v*v);
    (dd->volty.n)++;

    (dd->alpha.sum)+=alpha;
    (dd->alpha.sumsq)+=(alpha*alpha);
    (dd->alpha.n)++;

    (dd->effic.sum)+=effic;
    (dd->effic.sumsq)+=(effic*effic);
    (dd->effic.n)++;

    (dd->quant.sum)+=n_deals;
    (dd->quant.sumsq)+=(n_deals*n_deals);
    (dd->quant.n)++;

    (dd->pdisp.sum)+=pdisp;
    (dd->pdisp.sumsq)+=(pdisp*pdisp);
    (dd->pdisp.n)++;
  }
}


/*ddat-meanpmsd: plot mean plus and minus one standard deviation*/
void ddat_meanpmsd(FILE *fp,int field,int n_days,int n_exps,Day_data dd[])
{ int d,n[MAX_N_DAYS];
  Real mean,meansq,diff,sum[MAX_N_DAYS],sumsq[MAX_N_DAYS];
  char fieldstr[30];

  if(n_days>MAX_N_DAYS)
  { fprintf(stderr,"\nFAIL: MAX_N_DAYS too small in ddat.c: recompile\n");
    exit(0);
  }

  switch(field)
  { case DD_ALPHA: strcpy(fieldstr,"Alpha");
                   for(d=0;d<n_days;d++)
                   { sum[d]=dd[d].alpha.sum;
                     sumsq[d]=dd[d].alpha.sumsq;
                     n[d]=dd[d].alpha.n;
                   }
                   break;

    case DD_QUANT: strcpy(fieldstr,"Quantity");
                   for(d=0;d<n_days;d++)
                   { sum[d]=dd[d].quant.sum;
                     sumsq[d]=dd[d].quant.sumsq;
                     n[d]=dd[d].quant.n;
                   }
                   break;

    case DD_EFFIC: strcpy(fieldstr,"Efficiency");
                   for(d=0;d<n_days;d++)
                   { sum[d]=dd[d].effic.sum;
                     sumsq[d]=dd[d].effic.sumsq;
                     n[d]=dd[d].effic.n;
                   }
                   break;

    case DD_PRICE: strcpy(fieldstr,"Price");
                   for(d=0;d<n_days;d++)
                   { sum[d]=dd[d].price.sum;
                     sumsq[d]=dd[d].price.sumsq;
                     n[d]=dd[d].price.n;
                   }
                   break;

    case DD_PDISP: strcpy(fieldstr,"Dispersion");
                   for(d=0;d<n_days;d++)
                   { sum[d]=dd[d].pdisp.sum;
                     sumsq[d]=dd[d].pdisp.sumsq;
                     n[d]=dd[d].pdisp.n;
                   }
                   break;

    case DD_VOLTY: strcpy(fieldstr,"Volatility");
                   for(d=0;d<n_days;d++)
                   { sum[d]=dd[d].volty.sum;
                     sumsq[d]=dd[d].volty.sumsq;
                     n[d]=dd[d].volty.n;
                   }
                   break;

    default: fprintf(stderr,"\nFAIL: bad field in ddat_meanpmsd (%d)\n",field);
             exit(0);
  }

  fprintf(fp,"\" %s (mean)\n",fieldstr);
  for(d=0;d<n_days;d++) fprintf(fp,"%d %f\n",d+1,sum[d]/n[d]);
  fprintf(fp,"\n");

  fprintf(fp,"\" %s (-1s.d.)\n",fieldstr);
  for(d=0;d<n_days;d++)
  { mean=sum[d]/n[d];
    meansq=mean*mean;
    diff=(sumsq[d]/n[d])-meansq;
    if(diff<SMALLREAL) diff=0.0;
    fprintf(fp,"%d %f\n",d+1,mean-sqrt(diff));
  }
  fprintf(fp,"\n");

  fprintf(fp,"\" %s (+1s.d.)\n",fieldstr);
  for(d=0;d<n_days;d++)
  { mean=sum[d]/n[d];
    meansq=mean*mean;
    diff=(sumsq[d]/n[d])-meansq;
    if(diff<SMALLREAL) diff=0.0;
    fprintf(fp,"%d %f\n",d+1,mean+sqrt(diff));
  }
  fprintf(fp,"\n");
}


/*ddat-xgraph: plot the daily stats in xgraph format*/
void xg_daily_graph(Day_data dd[],int n_days,int n_exps,char *fname)
{ int d;
  FILE *fp;

  fp=fopen(fname,"w");

  fprintf(fp,"TitleText: %s: n=%d\n\n",fname,n_exps);

  if(n_exps<2)
  { /*no sense in calculating SD*/

    fprintf(fp,"\" Alpha\n");
    for(d=0;d<n_days;d++) fprintf(fp,"%d %f\n",d+1,dd[d].alpha.sum);
    fprintf(fp,"\n");

    fprintf(fp,"\" Efficiency\n");
    for(d=0;d<n_days;d++) fprintf(fp,"%d %f\n",d+1,dd[d].effic.sum);
    fprintf(fp,"\n");

    fprintf(fp,"\" Quantity\n");
    for(d=0;d<n_days;d++) fprintf(fp,"%d %f\n",d+1,dd[d].quant.sum);
    fprintf(fp,"\n");

    fprintf(fp,"\" Dispersion\n");
    for(d=0;d<n_days;d++) fprintf(fp,"%d %f\n",d+1,dd[d].pdisp.sum);
    fprintf(fp,"\n");
  }

  else
  { /*plot mean and s.d. for the daily stats*/
    ddat_meanpmsd(fp,DD_PRICE,n_days,n_exps,dd);
    ddat_meanpmsd(fp,DD_ALPHA,n_days,n_exps,dd);
    ddat_meanpmsd(fp,DD_EFFIC,n_days,n_exps,dd);
    ddat_meanpmsd(fp,DD_QUANT,n_days,n_exps,dd);
    ddat_meanpmsd(fp,DD_PDISP,n_days,n_exps,dd);
    ddat_meanpmsd(fp,DD_VOLTY,n_days,n_exps,dd);
  }
  fclose(fp);
}
```

A.5 expctl.h

```
/*expctl.h: header file for experiment control data struct and i/o etc
  Dave Cliff,  Aug 1996*/


/*Agent-sched: data associated with one agent's buy/sell limits etc*/
typedef struct an_agent_sched{
  int n_units;           /*how many units the agent has/wants*/
  Real limit[MAX_UNITS]; /*limit price of each unit*/
} Agent_sched;


/*SD-sched: data associated with a supply or demand schedule*/
typedef struct sd_sched{
  int n_agents;                   /*how many agents involved*/
  int first_day;                  /*first day this schedule applies to*/
  int last_day;                   /*last day this schedule applies to*/
  int can_shout;                  /*boolean: 0=>silent traders; 1=>can shout*/
  Agent_sched agents[MAX_AGENTS]; /*details of individual agents*/
} SD_sched;


/*Expctl: experiment control parameters*/
typedef struct a_expctl{
  char id[MAX_ID];               /*id characters for output files*/
  int n_days;                    /*number of trading periods to run for*/
  int min_trades;                /*minimum number of trades per day*/
  int max_trades;                /*maximum number of trades per day*/
  int random;                    /*boolean: 0=> ZIP; 1=>ZI-C*/
  int nyse;                      /*boolean: 0=>NYSE off; 1=>NYSE on*/
  int n_dem_sched;               /*number of demand schedules*/
  SD_sched dem_sched[MAX_SCHED]; /*details of demand schedules*/
  int d_sched;                   /*index of currently active demand schedule*/
  int n_sup_sched;               /*number of supply schedules*/
  SD_sched sup_sched[MAX_SCHED]; /*details of supply schedules*/
  int s_sched;                   /*index of currently active supply schedule*/
} Expctl;

/*Initctl: genome initialization control parameters*/
typedef struct a_initctl{
    Real pr_shout;
    Real beta_min;
    Real beta_del;
    Real gamma_min;
    Real gamma_del;
    Real mu_min;
    Real mu_del;
    Real ca_min;
    Real ca_del;
    Real cr_min;
    Real cr_del;
    int  n_cases;
} A_initctl;

typedef struct an_initctl{
    A_initctl min;
    A_initctl max;
} Initctl;

typedef struct a_gactl{
    int   pop_size;   // population size
    Real  mut_start;  // mutation rate at start (anneals to mut_end)
    Real  mut_end;    // mutation rate at end
    Real  pr_casemut; // pr of flip-mutating the number of cases.
    Real  pr_xover;   // pr of xover -- should be ~prop to 1/GENOME_LEN.
    int   n_kids;     // number of kids per reproduction event
    Real  evo_Qs;     // Qs: if <0.0 then freely evolving, otherwise fixed.
} GActl;
```

```
void expctl_in(char [],Expctl *,int);
void initctl_in(char [],Initctl *,int);
void gactl_in(char[],GActl *,int);
```

A.6 expctl.c

```
/*expctl.c: read experiment control parameters from a file\\
  Dave Cliff\\
  Aug 1996\\*/

/* This does {\em some} validity checks but still need to be careful that the
   data-file it reads from is structured correctly. When there is more
   than one schedule for supply or demand, they must be listed in the data-file
   in the order they are to become active. The first-day for the 0th schedule
   is set to zero, whatever value is given in the data-file
*/

#include <math.h>
#include <stdio.h>
#include <ctype.h>

#include "random.h"
#include "max.h"
#include "expctl.h"

#define LLEN 1024 /*max. no. of characters in a line*/

/*get-non-comment-line: read to start of next line that doesn't start with
`\verb-#-'*/
int get_non_comment_line(FILE *fp)
{ int c,reading=1;
  char s[LLEN];

  while(reading)
  { /*get to first non-whitespace char*/
    c=' ';
    while(isspace(c))
    { c=fgetc(fp);
      if(c==EOF) return(EOF);
    }

    /*is this a comment line?*/
    if(c=='#')
    { /*yes: read the rest of this line*/
      ungetc(c,fp);
      fgets(s,LLEN,fp);
    }
    else
    { /*no: put the char back and exit*/
      ungetc(c,fp);
      return(EOF-1); /*i.e. something that isn't EOF*/
    }
  }
}


/*read-sched: read a supply or demand schedule*/
int read_sched(FILE *fp,SD_sched *sched,int verbose)
{ int i,*pi,a,u;
  float f,*pf;

  pi=&i; pf=&f;

  /*read number of agents*/
  if(get_non_comment_line(fp)!=EOF)
  { if(fscanf(fp,"%d",pi)!=EOF)
    { sched->n_agents=(*pi);
      if((sched->n_agents<1)||(sched->n_agents>MAX_AGENTS))
      { fprintf(stderr,"\nFail: # agents must be in range {1,...,%d}\n",
                MAX_AGENTS);
        exit(-1);
      }
      if(verbose)
      { fprintf(stdout,"  %d agents: ",sched->n_agents); fflush(stdout); }
    }
    else {fprintf(stderr,"\nFail: can't read # agents \n"); exit(0);}
  }
  else {fprintf(stderr,"\nFail: EOF reading # agents\n"); exit(0);}


  /*read start day*/
  if(get_non_comment_line(fp)!=EOF)
  { if(fscanf(fp,"%d",pi)!=EOF)
    { sched->first_day=(*pi);
      if(verbose)
      { fprintf(stdout,"from day %d ",sched->first_day); fflush(stdout); }
    }
  }

  /*read end day*/
  if(get_non_comment_line(fp)!=EOF)
  { if(fscanf(fp,"%d",pi)!=EOF)
    { sched->last_day=(*pi);
      if(sched->last_day<sched->first_day)
      { fprintf(stderr,"\nFail: last_day(%d)<first_day(%d)\n",
                sched->last_day,sched->first_day);
        exit(0);
      }
      if(verbose)
      { fprintf(stdout,"to day %d\n",sched->last_day); fflush(stdout); }
    }
  }

  /*read shout flag*/
  if(get_non_comment_line(fp)!=EOF)
  { if(fscanf(fp,"%d",pi)!=EOF)
    { sched->can_shout=(*pi);
      if((sched->can_shout<0)||(sched->can_shout>1))
      { fprintf(stderr,"\nFail: can_shout not Boolean (%d)\n",
                sched->can_shout);
        exit(0);
      }
      if(verbose)
      { if(sched->can_shout)
        { fprintf(stdout,"(These traders CAN SHOUT)\n"); fflush(stdout); }
        else
        { fprintf(stdout,"(These traders are SILENT)\n"); fflush(stdout); }
      }
    }
  }

  /*read agent pricing specs*/
  for(a=0;a<sched->n_agents;a++)
  { if(get_non_comment_line(fp)!=EOF)
    { if(fscanf(fp,"%d",pi)!=EOF)
      { sched->agents[a].n_units=(*pi);
        if((sched->agents[a].n_units<1)||(sched->agents[a].n_units>MAX_UNITS))
        { fprintf(stderr,"\nFail: # units must be inrange {1,...,%d}\n",
                   MAX_UNITS);
          exit(0);
        }

        if(verbose)
        { fprintf(stdout,"  Agent %2d, %d units: ",a,sched->agents[a].n_units);
          fflush(stdout);
        }

        for(u=0;u<sched->agents[a].n_units;u++)
        { if(fscanf(fp,"%f",pf)!=EOF)
          { sched->agents[a].limit[u]=(Real)(*pf);
            if(sched->agents[a].limit[u]<0.0)
            { fprintf(stderr,"\nFail: negative price (%f)\n",*pf);
              exit(0);
            }
            if(verbose)
```

```
            { fprintf(stdout,"%f ",sched->agents[a].limit[u]);
              fflush(stdout);
            }
          }
        }

        if(verbose) {fprintf(stdout,"\n"); fflush(stdout); }
      }
    }
  } /*end of reading the agent data*/
  return(1);
}


/*expctl-in: read expctl data from a specified file*/
void expctl_in(char filename[],Expctl *ec,int verbose)
{ int *pi,i,sched;
  float f,*pf;
  FILE *fp;

  fp=fopen(filename,"r");
  if(fp==NULL)
  { fprintf(stderr,"\nFAIL: can't open \"%s\" as expctl input file\n",filename);
    exit(0);
  }

  pi=&i; pf=&f;

  /*read id string*/
  if(get_non_comment_line(fp)!=EOF)
  { /*copy id string up to but not including the newline*/
    fscanf(fp,"%s\n",&(ec->id));
    if(verbose)
    { fprintf(stdout,"ID: %s\n",ec->id); fflush(stdout); }
  }

  /*read number of days*/
  if(get_non_comment_line(fp)!=EOF)
  { if(fscanf(fp,"%d",pi)!=EOF)
    { ec->n_days=(*pi);
      if((ec->n_days<1)||(ec->n_days>MAX_N_DAYS))
      { fprintf(stderr,"\nFail: # trading days must be in range {1,...,%d}\n",
                MAX_N_DAYS);
        exit(0);
      }
      if(verbose)
      { fprintf(stdout,"%d days: ",ec->n_days); fflush(stdout); }
    }
    else { fprintf(stderr,"\nFail: can't read number of days\n"); exit(0); }
  }
  else { fprintf(stderr,"\nFail: EOF reading number of days\n"); exit(0); }

  /*read min number of trades per day*/
  if(get_non_comment_line(fp)!=EOF)
  { if(fscanf(fp,"%d",pi)!=EOF)
    { ec->min_trades=(*pi);
      if((ec->min_trades<1)||(ec->min_trades>MAX_TRADES))
      { fprintf(stderr,"\nFail: min # trades must be in range {1,...,%d}\n",
                MAX_TRADES);
        exit(0);
      }
      if(verbose)
      { fprintf(stdout,"min_trades=%d ",ec->min_trades); fflush(stdout); }
    }
    else { fprintf(stderr,"\nFail: can't read min_trades\n"); exit(0); }
  }
  else { fprintf(stderr,"\nFail: EOF reading min_trades\n"); exit(0); }

  /*read max number of trades per day*/
  if(get_non_comment_line(fp)!=EOF)
  { if(fscanf(fp,"%d",pi)!=EOF)
    { ec->max_trades=(*pi);
      if((ec->max_trades<ec->min_trades)||(ec->max_trades>MAX_TRADES))
      { fprintf(stderr,"\nFail: max # trades muts be in range {%d,...,%d}\n",
                ec->min_trades,MAX_TRADES);
        exit(0);
      }
      if(verbose)
      { fprintf(stdout,"max_trades=%d\n",ec->max_trades); fflush(stdout); }
    }
    else { fprintf(stderr,"\nFail: can't read max_trades\n"); exit(0); }
  }
  else { fprintf(stderr,"\nFail: EOF reading max_trades\n"); exit(0); }


  /*read random flag*/
  if(get_non_comment_line(fp)!=EOF)
  { if(fscanf(fp,"%d",pi)!=EOF)
    { ec->random=(*pi);
      switch(ec->random)
      { case 1: if(verbose) fprintf(stdout,"Random (ZI-C) traders; ");
                break;

        case 0: if(verbose) fprintf(stdout,"Intelligent traders; ");
                break;

        default: fprintf(stderr,"\nFail: random flag must be boolean\n");
                 exit(0);
      }
      if(verbose) fflush(stdout);
    }
    else { fprintf(stderr,"\nFail: can't read random flag\n"); exit(0); }
  }
  else { fprintf(stderr,"\nFail: EOF reading random flag\n"); exit(0); }


  /*read nyse flag*/
  if(get_non_comment_line(fp)!=EOF)
  { if(fscanf(fp,"%d",pi)!=EOF)
    { ec->nyse=(*pi);
      switch(ec->nyse)
      { case 1: if(verbose) fprintf(stdout,"NYSE trading rules\n");
                break;

        case 0: if(verbose) fprintf(stdout,"no NYSE rules\n");
                break;

        default: fprintf(stderr,"\nFail: NYSE flag must be boolean\n");
                 exit(0);
      }
      if(verbose) fflush(stdout);
    }
    else { fprintf(stderr,"\nFail: can't read nyse flag\n"); exit(0); }
  }
  else { fprintf(stderr,"\nFail: EOF reading nyse flag\n"); exit(0); }

  /*read number of demand schedules*/
  if(get_non_comment_line(fp)!=EOF)
  { if(fscanf(fp,"%d",pi)!=EOF)
    { ec->n_dem_sched=(*pi);
      if((ec->n_dem_sched<1)||(ec->n_dem_sched>MAX_SCHED))
      { fprintf(stderr,"\nFail: # demand scheds must be in range {1,...,%d}\n",
                MAX_SCHED);
        exit(0);
      }

      if(verbose)
      { fprintf(stdout,"%d demand schedules:\n",ec->n_dem_sched);
        fflush(stdout);
      }
    }
```

```
    else {fprintf(stderr,"\nFail: can't read # demand schedules\n"); exit(0);}
  }
  else {fprintf(stderr,"\nFail: EOF reading # demand schedules\n"); exit(0);}


  /*read the schedules*/
  for(sched=0;sched<ec->n_dem_sched;sched++)
  { if(verbose) fprintf(stdout,"  Demand schedule %d:\n",sched);
    if(read_sched(fp,&(ec->dem_sched[sched]),verbose)==EOF)
    { fprintf(stderr,"\nFail: no more demand schedules\n");
      exit(0);
    }
  }

  ec->d_sched=0;
  ec->dem_sched[ec->d_sched].first_day=0;


  /*read number of supply schedules*/
  if(get_non_comment_line(fp)!=EOF)
  { if(fscanf(fp,"%d",pi)!=EOF)
    { ec->n_sup_sched=(*pi);
      if((ec->n_sup_sched<1)||(ec->n_sup_sched>MAX_SCHED))
      { fprintf(stderr,"\nFail: # supply scheds must be in range {1,...,%d}\n",
                 MAX_SCHED);
        exit(0);
      }

      if(verbose)
      { fprintf(stdout,"%d supply schedules:\n",ec->n_sup_sched);
        fflush(stdout);
      }
    }
    else {fprintf(stderr,"\nFail: can't read # supply schedules\n"); exit(0);}
  }
  else {fprintf(stderr,"\nFail: EOF reading # supply schedules\n"); exit(0);}


  /*read the schedules*/
  for(sched=0;sched<ec->n_sup_sched;sched++)
  { if(verbose) fprintf(stdout,"  Supply schedule %d:\n",sched);
    if(read_sched(fp,&(ec->sup_sched[sched]),verbose)==EOF)
    { fprintf(stderr,"\nFail: no more supply schedules\n");
      exit(0);
    }
  }

  ec->s_sched=0;
  ec->sup_sched[ec->s_sched].first_day=0;

  fclose(fp);
}


/*initctl_in: read initialization control data from a specified file*/
void initctl_in(char filename[],Initctl *ic,int verbose)
{ int i,i2;
  float f;
  FILE *fp;

  fp=fopen(filename,"r");
  if(fp==NULL)
  { fprintf(stderr,"\nFAIL: can't open \"%s\" as initctl input file\n",filename);
    exit(-1);
  }

  i=fscanf(fp,"%f",&f);
  if(i!=EOF)
  { ic->min.pr_shout=(Real)f; }
  else
  { fprintf(stderr,"\nFail: can't read min.pr_shout from %s\n",filename);
    exit(-1);
  }
  i=fscanf(fp,"%f",&f);
  if(i!=EOF)
  { ic->max.pr_shout=(Real)f; }
  else
  { fprintf(stderr,"\nFail: can't read max.pr_shout from %s\n",filename);
    exit(-1);
  }
  i=fscanf(fp,"%f",&f);
  if(i!=EOF)
  { ic->min.beta_min=(Real)f; }
  else
  { fprintf(stderr,"\nFail: can't read min.beta_min from %s\n",filename);
    exit(-1);
  }
  i=fscanf(fp,"%f",&f);
  if(i!=EOF)
  { ic->max.beta_min=(Real)f; }
  else
  { fprintf(stderr,"\nFail: can't read max.beta_min from %s\n",filename);
    exit(-1);
  }
  i=fscanf(fp,"%f",&f);
  if(i!=EOF)
  { ic->min.beta_del=(Real)f; }
  else
  { fprintf(stderr,"\nFail: can't read min.beta_del from %s\n",filename);
    exit(-1);
  }
  i=fscanf(fp,"%f",&f);
  if(i!=EOF)
  { ic->max.beta_del=(Real)f; }
  else
  { fprintf(stderr,"\nFail: can't read max.beta_del from %s\n",filename);
    exit(-1);
  }
  i=fscanf(fp,"%f",&f);
  if(i!=EOF)
  { ic->min.gamma_min=(Real)f; }
  else
  { fprintf(stderr,"\nFail: can't read min.gamma_min from %s\n",filename);
    exit(-1);
  }
  i=fscanf(fp,"%f",&f);
  if(i!=EOF)
  { ic->max.gamma_min=(Real)f; }
  else
  { fprintf(stderr,"\nFail: can't read max.gamma_min from %s\n",filename);
    exit(-1);
  }
  i=fscanf(fp,"%f",&f);
  if(i!=EOF)
  { ic->min.gamma_del=(Real)f; }
  else
  { fprintf(stderr,"\nFail: can't read min.gamma_del from %s\n",filename);
    exit(-1);
  }
  i=fscanf(fp,"%f",&f);
  if(i!=EOF)
  { ic->max.gamma_del=(Real)f; }
  else
  { fprintf(stderr,"\nFail: can't read max.gamma_del from %s\n",filename);
    exit(-1);
  }
  i=fscanf(fp,"%f",&f);
  if(i!=EOF)
  { ic->min.mu_min=(Real)f; }
  else
  { fprintf(stderr,"\nFail: can't read min.mu_min from %s\n",filename);
    exit(-1);
```

```
}
i=fscanf(fp,"%f",&f);
if(i!=EOF)
{ ic->max.mu_min=(Real)f; }
else
{ fprintf(stderr,"\nFail: can't read max.mu_min from %s\n",filename);
  exit(-1);
}
i=fscanf(fp,"%f",&f);
if(i!=EOF)
{ ic->min.mu_del=(Real)f; }
else
{ fprintf(stderr,"\nFail: can't read min.mu_del from %s\n",filename);
  exit(-1);
}
i=fscanf(fp,"%f",&f);
if(i!=EOF)
{ ic->max.mu_del=(Real)f; }
else
{ fprintf(stderr,"\nFail: can't read max.mu_del from %s\n",filename);
  exit(-1);
}
i=fscanf(fp,"%f",&f);
if(i!=EOF)
{ ic->min.ca_min=(Real)f; }
else
{ fprintf(stderr,"\nFail: can't read min.ca_min from %s\n",filename);
  exit(-1);
}
i=fscanf(fp,"%f",&f);
if(i!=EOF)
{ ic->max.ca_min=(Real)f; }
else
{ fprintf(stderr,"\nFail: can't read max.ca_min from %s\n",filename);
  exit(-1);
}
i=fscanf(fp,"%f",&f);
if(i!=EOF)
{ ic->min.ca_del=(Real)f; }
else
{ fprintf(stderr,"\nFail: can't read min.ca_del from %s\n",filename);
  exit(-1);
}
i=fscanf(fp,"%f",&f);
if(i!=EOF)
{ ic->max.ca_del=(Real)f; }
else
{ fprintf(stderr,"\nFail: can't read max.ca_del from %s\n",filename);
  exit(-1);
}
i=fscanf(fp,"%f",&f);
if(i!=EOF)
{ ic->min.cr_min=(Real)f; }
else
{ fprintf(stderr,"\nFail: can't read min.cr_min from %s\n",filename);
  exit(-1);
}
i=fscanf(fp,"%f",&f);
if(i!=EOF)
{ ic->max.cr_min=(Real)f; }
else
{ fprintf(stderr,"\nFail: can't read max.cr_min from %s\n",filename);
  exit(-1);
}
i=fscanf(fp,"%f",&f);
if(i!=EOF)
{ ic->min.cr_del=(Real)f; }
else
{ fprintf(stderr,"\nFail: can't read min.cr_del from %s\n",filename);
  exit(-1);
}
```

```
  i=fscanf(fp,"%f",&f);
  if(i!=EOF)
  { ic->max.cr_del=(Real)f; }
  else
  { fprintf(stderr,"\nFail: can't read max.cr_del from %s\n",filename);
    exit(-1);
  }
  i=fscanf(fp,"%d",&i2);
  if(i!=EOF)
  { ic->min.n_cases=i2; }
  else
  { fprintf(stderr,"\nFail: can't read min.n_cases from %s\n",filename);
    exit(-1);
  }
  if((i2!=1)&&(i2!=2)&&(i2!=4)&&(i2!=6))
  { fprintf(stderr,"\nFail: min.n_cases=%d not one of (1,2,4,6)\n",i2);
    exit(-1);
  }
  i=fscanf(fp,"%d",&i2);
  if(i!=EOF)
  { ic->max.n_cases=i2; }
  else
  { fprintf(stderr,"\nFail: can't read max.n_cases from %s\n",filename);
    exit(-1);
  }
  if((i2!=1)&&(i2!=2)&&(i2!=4)&&(i2!=6))
  { fprintf(stderr,"\nFail: max.n_cases=%d not one of (1,2,4,6)\n",i2);
    exit(-1);
  }

  if(verbose)
  { fprintf(stdout,"\nInitCtl:\n");
    fprintf(stdout,"Pr_Shout(min)=%f Pr_Shout(max)=%f\n",
                    ic->min.pr_shout,ic->max.pr_shout);
    fprintf(stdout,"Beta min(min)=%f min(max)=%f del(min)=%f del(max)=%f\n",
                    ic->min.beta_min,ic->max.beta_min,ic->min.beta_del,ic->max.beta_del);
    fprintf(stdout,"Gamma min(min)=%f min(max)=%f del(min)=%f del(max)=%f\n",
                    ic->min.gamma_min,ic->max.gamma_min,ic->min.gamma_del,
                    ic->max.gamma_del);
    fprintf(stdout,"Mu min(min)=%f min(max)=%f del(min)=%f del(max)=%f\n",
                    ic->min.mu_min,ic->max.mu_min,ic->min.mu_del,ic->max.mu_del);
    fprintf(stdout,"Ca min(min)=%f min(max)=%f del(min)=%f del(max)=%f\n",
                    ic->min.ca_min,ic->max.ca_min,ic->min.ca_del,ic->max.ca_del);
    fprintf(stdout,"Cr min(min)=%f min(max)=%f del(min)=%f del(max)=%f\n",
                    ic->min.cr_min,ic->max.cr_min,ic->min.cr_del,ic->max.cr_del);
    fprintf(stdout,"N_cases min=%d max=%d\n",
                    ic->min.n_cases,ic->max.n_cases);
  }

  fclose(fp);
}


/*gactl_in: read ga control data from a specified file*/
void gactl_in(char filename[],GActl *gac,int verbose)
{ int i,i2;
  float f;
  FILE *fp;

  fp=fopen(filename,"r");
  if(fp==NULL)
  { fprintf(stderr,"\nFAIL: can't open \"%s\" as GActl input file\n",filename);
    exit(-1);
  }

  i=fscanf(fp,"%d",&i2);
  if(i!=EOF)
  { gac->pop_size=i2; }
  else
  { fprintf(stderr,"\nFail: can't read population size from %s\n",filename);
    exit(-1);
  }

  i=fscanf(fp,"%f",&f);
  if(i!=EOF)
  { gac->mut_start=(Real)f; }
  else
  { fprintf(stderr,"\nFail: can't read starting mutation rate from %s\n",filename);
    exit(-1);
  }

  i=fscanf(fp,"%f",&f);
  if(i!=EOF)
  { gac->mut_end=(Real)f; }
  else
  { fprintf(stderr,"\nFail: can't read end mutation rate from %s\n",filename);
    exit(-1);
  }

  i=fscanf(fp,"%f",&f);
  if(i!=EOF)
  { gac->pr_casemut=(Real)f; }
  else
  { fprintf(stderr,"\nFail: can't read n_cases mutation probability from %s\n",filename);
    exit(-1);
  }

  i=fscanf(fp,"%f",&f);
  if(i!=EOF)
  { gac->pr_xover=(Real)f; }
  else
  { fprintf(stderr,"\nFail: can't read real-genome crossover probability from %s\n",
            filename);
    exit(-1);
  }

  i=fscanf(fp,"%d",&i2);
  if(i!=EOF)
  { gac->n_kids=i2; }
  else
  { fprintf(stderr,"\nFail: can't read n_kids from %s\n",filename);
    exit(-1);
  }

  i=fscanf(fp,"%f",&f);
  if(i!=EOF)
  { gac->evo_Qs=(Real)f; }
  else
  { fprintf(stderr,"\nFail: can't read evo_Qs from %s\n",filename);
    exit(-1);
  }

  if(verbose)
  { fprintf(stdout,"\nGActl:\n");
    fprintf(stdout,"Pop-size=%d\n",gac->pop_size);
    fprintf(stdout,"mut_start=%f mut_end=%f\n",gac->mut_start,gac->mut_end);
    fprintf(stdout,"pr_casemut=%f pr_xover=%f\n",gac->pr_casemut,gac->pr_xover);
    fprintf(stdout,"n_kids=%d evo_Qs=%f min(max)=%f\n\n",gac->n_kids,gac->evo_Qs);
  }

  fclose(fp);
}
```

A.7 max.h

```
/*max.h: maxima for array bounds etc*/

#define MAX_N_DAYS 30
#define MAX_TRADES 100
#define TOT_TRADES (MAX_N_DAYS*MAX_TRADES)
#define MAX_FAILS 100  /*maximum numbers of bids/offers allowed to fail before
                       day's trading closes*/

#define MAX_BUYERS 100
#define MAX_SELLERS 100
#define MAX_AGENTS (MAX_BUYERS>MAX_SELLERS?MAX_BUYERS:MAX_SELLERS)
#define MAX_UNITS 2  /*max no. of units an agent can sell/buy*/
#define MAX_SCHED 4  /*max no. of supply or demand schedules in an experiment*/
#define MAX_ID 30    /*max no. of chars in id tag used for output files*/
```

A.8 smith.c

```
/*smith.c : the master program\\
Dave Cliff\\
Sept 1996\\*/

/*revised version: allowing long-genotype "multiple case" ZIP traders*/

#include <math.h>
#include <stdio.h>

#include "max.h"
#include "random.h"
#include "agent.h"
#include "sd.h"
#include "ddat.h"
#include "tdat.h"
#include "expctl.h"

#define MAX_POP_SIZE 30

/*reinit the traders when schedules change?*/
#define REINIT_ON_S_SCHED_CHANGE 0
#define REINIT_ON_D_SCHED_CHANGE 0

/*DC added020526: names of genome indices; also Qs is now a proper part of the genome*/
#define G_BM 0 /// beta min
#define G_BD 1 /// beta delta
#define G_GM 2 /// gamma min
#define G_GD 3 /// gamma delta
#define G_MM 4 /// mu min
#define G_MD 5 /// mu delta
#define G_CAM 6 /// ca min
#define G_CAD 7 /// ca delta
#define G_CRM 8 /// cr min
#define G_CRD 9 /// cr delta
#define G_QS 10 /// Qs
#define G_NCASES 11 /// the number of distinct cases this genome codes for
#define G_FIT 12 /// -- bit of a kludge: we store the fitness of the genome in
pop[i][G_FIT][0]

#define GENE_LEN 13 /// this should be consistent with the G_?? #def's above


/*reward: monetary reward for a deal*/
Real reward(Agent *a,Real price)
{ Real r;
  if((a->job)==SELL)
  { r=((price-(a->limit))); }
  else
  { r=(((a->limit)-price)); }

  if(r<0.0) r=0.0;

  return(r);
}


/*get-price: get a price from an agent)*/
Real get_price(Agent *a,int id,int random,int verbose)
{ Real price;
  Real rmin=0.01,rmax=4.0;  /*bounds on random prices*/

  if(random)
  { /*agent price is generated at random*/
    if(rmax<a->limit)
    { fprintf(stderr,"\nFail: rmax too low in get_price()\n");
      exit(0);
    }

    if(a->job==BUY) price=rmin+randval((a->limit)-rmin);
    else price=(a->limit)+randval(rmax-(a->limit));
    price=(floor(0.5+(price*100)))/100;
    a->price=price;
  }
  else price=a->price;

  if(verbose)
  { if(a->job==BUY) fprintf(stdout,"Buyer %d bids at %5.3f (reward=%5.3f)\n",
                            id,price,reward(a,price));
    else fprintf(stdout,"Seller %d offers at %5.3f (reward=%5.3f)\n",
                 id,price,reward(a,price));
  }
  return(price);
}


/*get-willing: form a list of agents willing to deal*/
int get_willing(Real price,Agent agents[],int n,int ilist[],char *s,int random,
                int verbose)
{ int willing=0,a;
  Real r_price,p;

  p=price;

  for(a=0;a<n;a++)
  { if(random)
    { /*agent generates a price at random, compares it to given price*/
      /*and is willing if random price makes a profit*/
      agents[a].willing=0;

      if(agents[a].active)
      { r_price=get_price(agents+a,a,random,verbose);
        if(agents[a].job==BUY)
        { if(r_price>price)
          { agents[a].willing=1; p=r_price; }
        }

        else
        { if(r_price<price)
          { agents[a].willing=1; p=r_price; }
        }
      }
    }

    else
    { /*use some intelligence*/
      willing_trade(agents+a,price);
    }
    if(agents[a].willing)
    { ilist[willing]=a;
      willing++;
      if(verbose)
      { fprintf(stdout,"%s%2d willing (r)price=%5.3f reward=%5.3f\n",
                s,a,p,reward(agents+a,price));
      }
    }
  }
  if(verbose) fprintf(stdout,"%d traders willing to deal\n",willing);

  return(willing);
}


/*get-able: form a list of agents able to deal*/
int get_able(Real price,Agent agents[],int n,int ilist[],char *s,int verbose)
{ int able=0,a;

  for(a=0;a<n;a++)
  { if(agents[a].able)
    { ilist[able]=a;
      able++;
      if(verbose)
      { fprintf(stdout,"%s%2d able (reward=%5.3f)\n",
                  s,a,reward(agents+a,price));
      }
    }
  }
  return(able);
}


/*bank: adjust bank balances of buyer and seller in a deal*/
void bank(Agent *s,Agent *b,Real price,Real *surplus,int verbose)
{ Real r,surp;

  surp=*surplus;

  /*seller*/
  r=reward(s,price);
  (s->bank)+=r;
  (s->a_gain)+=r;
  surp+=(r);

  (s->quant)--;
  if(s->quant<1) s->active=0;
  if(verbose)
  { fflush(stdout);
    fprintf(stdout,"Seller: limit=%.2f price=%.2f reward=%.2f bank=%.2f quant=%d ",
                   s->limit,price,r,s->bank,s->quant);
    fprintf(stdout,"surplus %f\n",surp);
    fflush(stdout);
  }

  /*buyer*/
  r=reward(b,price);
  (b->bank)+=r;
  (b->a_gain)+=r;
  surp+=(r);

  (b->quant)--;
  if(b->quant<1) b->active=0;
  if(verbose)
  { fprintf(stdout,"Buyer: limit=%.2f price=%.2f reward=%.2f bank=%.2f quant=%d ",
                    b->limit,price,r,b->bank,b->quant);
    fprintf(stdout,"surplus %f\n",surp);
    fflush(stdout);
  }
  *surplus=surp;
}


/*day-init: initialise all data structures for start of day*/
void day_init(int exp_number,int day_number,Day_data *ddat,Expctl *ec,
              Agent sellers[],Agent buyers[],
              Real *p_0,Real *max_surplus,int verbose)
{ int b,s,q_0,s_sched,d_sched,n_buy,n_sell;
  Real eq_profit;
  char filename[40];

  /*initialise the buyers*/

  if(day_number==0)
  { /*first day: read the first demand schedule*/
    ec->d_sched=0;
  }
  else
  if((day_number-1)==(ec->dem_sched[ec->d_sched].last_day))
  { /*previous day was last day on that demand schedule: update*/
    (ec->d_sched)++;
    if(ec->d_sched==ec->n_dem_sched)
    { fprintf(stderr,"\nFail: ran out of demand schedules on day %d\n",
              day_number);
      exit(0);
    }
  }
  d_sched=ec->d_sched;
  n_buy=ec->dem_sched[d_sched].n_agents;

  /*mark all buyers active, set quantities and limit prices*/
  for(b=0;b<n_buy;b++)
  { buyers[b].quant=ec->dem_sched[d_sched].agents[b].n_units;
    buyers[b].active=1;
    buyers[b].a_gain=0.0;
    /*NOTE: ONLY ALLOWS FOR ONE LIMIT PRICE*/
    buyers[b].limit=ec->dem_sched[d_sched].agents[b].limit[0];
    set_price(buyers+b);
    if(verbose) fprintf(stdout,"buyer %d price %f\n",b,buyers[b].price);
  }

  /*initialise the sellers*/

  if(day_number==0)
  { /*first day: read the first demand schedule*/
    ec->s_sched=0;
  }
  else
  if((day_number-1)==(ec->sup_sched[ec->s_sched].last_day))
  { /*previous day was last day on that supply schedule: update*/
    (ec->s_sched)++;
    if(ec->s_sched==ec->n_sup_sched)
    { fprintf(stderr,"\nFail: ran out of supply schedules on day %d\n",
              day_number);
      exit(0);
    }
  }
  s_sched=ec->s_sched;
  n_sell=ec->sup_sched[s_sched].n_agents;

  /*mark all sellers active, set quantities and limit prices*/
  for(s=0;s<n_sell;s++)
  { sellers[s].quant=ec->sup_sched[s_sched].agents[s].n_units;
    sellers[s].active=1;
    sellers[s].a_gain=0.0;
    /*NOTE: ONLY ALLOWS FOR ONE LIMIT PRICE*/
    sellers[s].limit=ec->sup_sched[s_sched].agents[s].limit[0];
    set_price(sellers+s);
    if(verbose) fprintf(stdout,"seller %d price %f\n",s,sellers[s].price);
  }


  /*find theoretical equilibrium price*/
  if(exp_number==0) sprintf(filename,"%ssd%02d_000.fig",ec->id,day_number+1);
  else sprintf(filename,"\0");
/***Switched off .fig output at MIT because doesn't work with old Xfig at MIT*/
  sprintf(filename,"\0");
///verbose=1;
  supdem(n_sell,sellers,n_buy,buyers,
         ec->max_trades,p_0,&q_0,max_surplus,EQ_THEORY,filename,
         NULL,verbose);
///verbose=0;
  /*set theoretical gains for buyers and sellers*/
  for(b=0;b<n_buy;b++)
  { eq_profit=buyers[b].quant*(buyers[b].limit-(*p_0));
    if(eq_profit<0.0) eq_profit=0.0;
    buyers[b].t_gain=eq_profit;
  }
  for(s=0;s<n_sell;s++)
  { eq_profit=sellers[s].quant*((*p_0)-sellers[s].limit);
    if(eq_profit<0.0) eq_profit=0.0;
    sellers[s].t_gain=eq_profit;
  }
}



/*trade: see if a buyer and a seller can be found who will enter into a trade*/
void trade(Trade_data *tdat,Agent sellers[],Agent buyers[],Expctl *ec,
           Real max_surplus,Real *surplus,int *stat,
           Real pr_seller_shouts, Real evoQs,
           int verbose)
{ int b,s,        /*buyer and seller indices*/
      dt,         /*deal type*/
      status,     /*what's happening*/
      eq_q,       /*equilibrium quantity*/
      n_willing,  /*number of agents willing to trade at a given price*/
      n_able,     /*number of agents able to trade at a given price*/
      n_fails,    /*number of failed/declined bids/offers*/
      n_buy,      /*number of buyers*/
      n_sell,     /*number of sellers*/
      active_b,   /*number of active buyers*/
      active_s,   /*number of active sellers*/
      sell_shout, /*can sellers shout offers?*/
      buy_shout,  /*can buyers shout bids?*/
      v_whoshout, /*for smooth variation experiments: says who shouts*/
      traders,    /*number of traders to choose from when generating shout*/
      first_offer,/*flag raised until an opening offer is made*/
      first_bid,  /*falg raised unitl an opening bid is made*/
      ilist[MAX_AGENTS]; /*list of indices*/

  Real pss,       /*local pr_seller_shouts*/
       eq_p,      /*equilibrium price*/
       cur_surp,  /*current actual max surplus*/
       best_offer,/*used in NYSE rules*/
       best_bid,  /*used in NYSE rules*/
       price;     /*price of bid/ask*/

  n_buy=ec->dem_sched[ec->d_sched].n_agents;
  n_sell=ec->sup_sched[ec->s_sched].n_agents;
  sell_shout=ec->sup_sched[ec->s_sched].can_shout;
  buy_shout=ec->dem_sched[ec->d_sched].can_shout;
  if((sell_shout==0)&&(buy_shout==0))
  { fprintf(stderr,"\nFAIL: Can't have both buyers AND sellers silent\n");
    exit(0);
  }

  /*find the theoretical equilibrium price*/
  supdem(n_sell,sellers,n_buy,buyers,ec->max_trades,
         &eq_p,&eq_q,&cur_surp,EQ_THEORY,
         "\0",NULL,verbose);
  if(eq_q!=NULL_EQ) { tdat->t_eq_p=eq_p; tdat->t_eq_q=eq_q; }
  else tdat->t_eq_q=NULL_EQ;

  /*find the actual equilibrium price*/
  supdem(n_sell,sellers,n_buy,buyers,ec->max_trades,
         &eq_p,&eq_q,&cur_surp,EQ_ACTUAL,
         "\0",NULL,verbose);
  if(eq_q!=NULL_EQ) { tdat->a_eq_p=eq_p; tdat->a_eq_q=eq_q; }
  else tdat->a_eq_q=NULL_EQ;


  n_fails=0;
  status=NO_DEAL;
  first_offer=1; first_bid=1;
  while((status==NO_DEAL)&&(n_fails<MAX_FAILS))
  {
    /*count active agents and mark them as able to bid*/
    active_b=0;
    for(b=0;b<n_buy;b++)
    { if(buyers[b].active)
      { buyers[b].able=1;
        active_b++;
      }
      else buyers[b].able=0;
    }
    active_s=0;
    for(s=0;s<n_sell;s++)
    { if(sellers[s].active)
      { sellers[s].able=1;
        active_s++;
      }
      else sellers[s].able=0;
    }


    traders=0;
    if(sell_shout) traders+=active_s;
    if(buy_shout) traders+=active_b;
    if(verbose) fprintf(stdout,"%d traders: active_s=%d active_b=%d\n",
                        traders,active_s,active_b);


    pss=pr_seller_shouts;
    if(evoQs>=0.0) pss=evoQs;

    if(randval(1.0)<pss)
    { /*is there a seller able to make an offer?*/
      dt=OFFER;
      if(ec->nyse&&(!first_offer))
      { if(ec->random)
        { /*any seller with a limit price higher than best offer can't deal*/
          for(s=0;s<n_sell;s++)
          { if(sellers[s].limit>best_offer) sellers[s].able=0; }
        }
        else
        { /*any seller with an equal or higher price can't offer*/
          for(s=0;s<n_sell;s++)
          { if(sellers[s].price>=best_offer) sellers[s].able=0; }
        }
      }
      n_able=get_able(0.0,sellers,n_sell,ilist,"S",verbose);

      if(n_able>0)
      { /*an able seller makes an offer*/
        s=ilist[irand(n_able)];
        /*get price for seller*/
        price=get_price(sellers+s,s,ec->random,verbose);
        if(ec->nyse)
        { if(first_offer)
          { best_offer=price; first_offer=0; }
          else
          { if(price<best_offer) best_offer=price; }
        }

        /*get willing buyers*/
        n_willing=get_willing(price,buyers,n_buy,ilist,"B",ec->random,verbose);
        if(n_willing>0) status=DEAL;
      }
      else
      {
        if(verbose) fprintf(stdout,"No sellers able to offer\n");
        n_fails=MAX_FAILS;
        status=END_DAY;
      }
    }

    else
    { /*is there a buyer able to make a bid?*/
      dt=BID;
      if(ec->nyse&&(!first_bid))
      { if(ec->random)
        { /*any buyer with limit lower than best bid can't deal*/
          for(b=0;b<n_buy;b++)
          { if(buyers[b].limit<best_bid) buyers[b].able=0; }
        }
        else
        { /*any buyer with an equal or lower price can't bid*/
          for(b=0;b<n_buy;b++)
          { if(buyers[b].price<=best_bid) buyers[b].able=0; }
        }
      }
      n_able=get_able(0.0,buyers,n_buy,ilist,"B",verbose);

      if(n_able>0)
      { /*an able buyer makes a bid*/
        b=ilist[irand(n_able)];
        /*get price for buyer*/
        price=get_price(buyers+b,b,ec->random,verbose);
        if(ec->nyse)
        { if(first_bid)
          { best_bid=price; first_bid=0; }
          else
          { if(price>best_bid) best_bid=price; }
        }
        /*get willing selllers*/
        n_willing=get_willing(price,sellers,n_sell,ilist,"S",ec->random,verbose);
        if(n_willing>0) status=DEAL;
      }
      else
      { if(verbose) fprintf(stdout,"No buyers able to bid\n");
        n_fails=MAX_FAILS;
        status=END_DAY;
      }
    }

    if(status==DEAL)
    { /*DEAL*/
      if(dt==OFFER)
      { /*select the willing buyer for this offer*/
        b=ilist[irand(n_willing)];
        if(verbose)
        { fprintf(stdout,
                  "Seller %d sells to Buyer %d (reward=%5.3f)\n",
                  s,b,reward(buyers+b,price));
        }
      }
      else
      { /*select the willing seller for this bid*/
        s=ilist[irand(n_willing)];
        if(verbose)
        { fprintf(stdout,
                  "Buyer %d buys from Seller %d (reward=%5.3f)\n",
                  b,s,reward(sellers+s,price));
        }
      }

      /*record what happened*/
      tdat->deal_p=price;
      tdat->deal_t=dt;

      /*update trading strategies of buyers and sellers*/
      shout_update(dt,status,n_sell,sellers,n_buy,buyers,price,verbose);

      /*update bank accounts of buyer and seller*/
      bank(sellers+s,buyers+b,price,surplus,verbose);

    }
    else
    { /*NO DEAL or END DAY*/
      n_fails++;
      if(verbose) fprintf(stdout,"No willing takers (fails=%d)\n",n_fails);
      tdat->deal_p=-1.0; /*negative price => no deal*/

      /*update trading strategies of buyers and sellers*/
```

```
      shout_update(dt,status,n_sell,sellers,n_buy,buyers,price,verbose);
    }
  }
  *stat=status;
}

/*clip a Real to within [0.0,1.0]*/

void Rclip01(Real *pr)
{ if(*pr>1.0) *pr=1.0;
  if(*pr<0.0) *pr=0.0;
}

int main(int argc,char *argv[])
{ int n_trans,    /*number of transactions on a day*/
      d,          /*day*/
      status,     /*how things are going*/
      rs,         /*random seed*/
      s,b,        /*seller, buyer,  loop indices*/
      n_buy,      /*number of buyers*/
      n_sell,     /*number of sellers*/
      n_trades,   /*number of trades done in a day*/
      n_exps,     /*number of experiments to run per fitness eval*/
      n_gens,     /*number of generations*/
      n_reps,     /*how many repetitions to do*/
      max_trades, /*maxmimum number of trades in a session*/
      rep,        /*number of repetitions*/
      verbose=0,  /*how verbose the output is*/
      eliteid,    /*the population index of the current generation's elite*/
      ats_n[MAX_TRADES], /*counts of entries in ats[]*/
      eq,         /*equilibrium quantity*/
      c,          /*case number/index*/
      d_sched_num, s_sched_num,  /*number of the sup/dem schedule currently being used*/
      e,          /*experiment number*/
      i,j,i1,i2,i3,g,nx,        /*individual and generation in GA*/
      elcnt=0,
      t;          /*transaction number within a day*/

  Real price,p_0,sigmasum,alpha,ep,last_price,sum_price_diff,
       sum_price,fitness,elitefit,
       mutlim,
       diff,pd,pdisp,pds,
       alphatrans,      /*alpha over transaction sequence (cf G+S fig6)*/
       ats[MAX_TRADES], /*alpha over transaction sequence (cf G+S fig6)*/
       bounddata[4],   /*can be used to inhibit autoscaling on supdem*/
       *bounds,
       pop1[MAX_POP_SIZE][GENE_LEN+1][MAX_N_CASES], /*population1*/
       pop2[MAX_POP_SIZE][GENE_LEN][MAX_N_CASES], /*population2*/
       fweight[MAX_N_DAYS], /*fitness weightings*/
       fweightsum,          /*sum of weightings*/
       nrv,max_surplus,surplus,efficiency;
  char fname[60];
  Day_data   ddat[MAX_N_DAYS];
  Trade_data tdat[MAX_N_DAYS][MAX_TRADES];
  Real_stat  ats_e[MAX_TRADES]; /*for summarising ats[] over experiments*/
  Agent      buyers[MAX_AGENTS],
                 sellers[MAX_AGENTS];
  Expctl     expctl;
  Initctl    initctl;
  GActl      gactl;
  FILE *fp,*fppopstats,*fpelitestats;


  if(argc<7)
  { fprintf(stderr,
   "\nUsage:\n smith <n_exps> <n_gens> <n_reps> <datafname> <initfname> <gactlfname>\n");
    exit(0);
  }
  sscanf(argv[1],"%d",&n_exps);
  sscanf(argv[2],"%d",&n_gens);
  sscanf(argv[3],"%d",&n_reps);
```

```
fprintf(stdout,
       "%d experiments, %d gens, %d reps; data-file=%s; init-file=%s ga-ctl-file=%s\n",
       n_exps,n_gens,n_reps,argv[4],argv[5],argv[6]);

/*disabled for the GA*/
/*
if(n_exps==1) rs=0;
else rs=999;
*/

/*repeat the same GA experiment N_REPS times, with unique data files for each rep*/

for(rep=0;rep<n_reps;rep++)
{

sprintf(&(fname[0]),"stats_pop_%02d",rep);
fppopstats=fopen(fname,"w");

sprintf(&(fname[0]),"stats_elite_%02d",rep);
fpelitestats=fopen(fname,"w");

rs=0;

rseed(&rs);

/*read the experiment control file*/
expctl_in(argv[4],&expctl,1);


/*read the experiment control file*/
initctl_in(argv[5],&initctl,1);


/*read the experiment control file*/
gactl_in(argv[6],&gactl,1);

if(gactl.pop_size>MAX_POP_SIZE)
{ fprintf(stderr,"\nFAIL: MAX_POP_SIZE too small: recompile\n");
  exit(-1);
}

/*set up the fitness weightings*/
for(d=0;d<MAX_N_DAYS;d++) fweight[d]=0,0;

/*the joys of cut and paste…*/

fweight[0]=1.75;
fweight[1]=1.50;
fweight[2]=1.25;
fweight[3]=1.00;
fweight[4]=1.00;
fweight[5]=1.00;

fweight[6]=1.75;
fweight[7]=1.50;
fweight[8]=1.25;
fweight[9]=1.00;
fweight[10]=1.00;
fweight[11]=1.00;

fweight[12]=1.75;
fweight[13]=1.50;
fweight[14]=1.25;
fweight[15]=1.00;
fweight[16]=1.00;
fweight[17]=1.00;

fweight[18]=1.75;
fweight[19]=1.50;
fweight[20]=1.25;
fweight[21]=1.00;
```

```
  fweight[22]=1.00;
  fweight[23]=1.00;

  fweightsum=0.0;
  for(d=0;d<expctl.n_days;d++) fweightsum+=fweight[d];

  /*generate initial random population*/
  for(i=0;i<gactl.pop_size;i++)
  {
    for(c=0;c<MAX_N_CASES;c++)
    {
      pop1[i][G_BM][c]=initctl.min.beta_min+
                       randval(initctl.max.beta_min-initctl.min.beta_min);
      pop1[i][G_BD][c]=initctl.min.beta_del+
                       randval(initctl.max.beta_del-initctl.min.beta_del);
      if(pop1[i][G_BM][c]+pop1[i][G_BD][c]>1.0) pop1[i][G_BD][c]=1.0-pop1[i][G_BM][c];

      pop1[i][G_GM][c]=initctl.min.gamma_min+
                       randval(initctl.max.gamma_min-initctl.min.gamma_min);
      pop1[i][G_GD][c]=initctl.min.gamma_del+
                       randval(initctl.max.gamma_del-initctl.min.gamma_del);
      if(pop1[i][G_GM][c]+pop1[i][G_GD][c]>1.0) pop1[i][G_GD][c]=1.0-pop1[i][G_GM][c];

      pop1[i][G_MM][c]=initctl.min.mu_min+randval(initctl.max.mu_min-initctl.min.mu_min);
      pop1[i][G_MD][c]=initctl.min.mu_del+randval(initctl.max.mu_del-initctl.min.mu_del);

      if(pop1[i][G_MM][c]+pop1[i][G_MD][c]>1.0) pop1[i][G_MD][c]=1.0-pop1[i][G_MM][c];

      pop1[i][G_CAM][c]=initctl.min.ca_min+
                        randval(initctl.max.ca_min-initctl.min.ca_min);
      pop1[i][G_CAD][c]=initctl.min.ca_del+
                        randval(initctl.max.ca_del-initctl.min.ca_del);
      if(pop1[i][G_CAM][c]+pop1[i][G_CAD][c]>1.0)
                  pop1[i][G_CAD][c]=1.0-pop1[i][G_CAM][c];

      pop1[i][G_CRM][c]=initctl.min.cr_min+
                        randval(initctl.max.cr_min-initctl.min.cr_min);
      pop1[i][G_CRD][c]=initctl.min.cr_del+
                        randval(initctl.max.cr_del-initctl.min.cr_del);
      if(pop1[i][G_CRM][c]+pop1[i][G_CRD][c]>1.0)
                  pop1[i][G_CRD][c]=1.0-pop1[i][G_CRM][c];

      fprintf(stdout,
      "%3d, case %1d: b[%4.2f,%4.2f] g[%4.2f,%4.2f] m[%4.2f,%4.2f] ca[%4.2f,%4.2f]
cr[%4.2f,%4.2f]\n",
               i,c,pop1[i][G_BM][c],pop1[i][G_BM][c]+pop1[i][G_BD][c],
                   pop1[i][G_GM][c],pop1[i][G_GM][c]+pop1[i][G_GD][c],
                   pop1[i][G_MM][c],pop1[i][G_MM][c]+pop1[i][G_MD][c],
                   pop1[i][G_CAM][c],pop1[i][G_CAD][c],
                   pop1[i][G_CRM][c],pop1[i][G_CRD][c]
             );
      fflush(stdout);
    }

    pop1[i][G_QS][0]=initctl.min.pr_shout+
                     randval(initctl.max.pr_shout-initctl.min.pr_shout);

    /* careful...this could loop a lot*/
    c=-1;
    while((c<initctl.min.n_cases)||(c>initctl.max.n_cases))
    { c=2*irand(4);
      if(c<1) c=1;
    }
    pop1[i][G_NCASES][0]=c;


  }

  /*do n generations*/
  for(g=0;g<n_gens;g++)
  {
     elitefit=1000000.00; ///assume this gen's elite fitness is terrible.

  for(i=0;i<gactl.pop_size;i++)
  { /*evaluate one individual*/

    /*adjust this individual's genome to reflect its N_CASES setting*/
    switch((int)pop1[i][G_NCASES][0])
    { case 1: /*only one case so copy first case into all*/
              for(c=1;c<MAX_N_CASES;c++)
              { for(j=0;j<GENE_LEN;j++)
                { pop1[i][j][c]=pop1[i][j][0];
                }
              }
              break;
      case 2: /*buyer and seller are only distinct cases;*/
              for(c=1;c<3;c++)
              { for(j=0;j<GENE_LEN;j++)
                { pop1[i][j][c]=pop1[i][j][0];
                  pop1[i][j][3+c]=pop1[i][j][3];
                }
              }
              break;
      case 4: /*only one case so copy first case into all*/
              for(j=0;j<GENE_LEN;j++)
              { pop1[i][j][2]=pop1[i][j][1];
                pop1[i][j][5]=pop1[i][j][4];
              }
              break;
      case 6: /*no copying needed*/
              break;
      default: fprintf(stderr,
                      "\nFAIL: bad N_cases (=%d) on genome %d\n",
                      (int)pop1[i][G_NCASES][0],i); exit(-1);
    }



  /*initialise daily data records*/
  for(d=0;d<expctl.n_days;d++) ddat_init(ddat+d);

  for(t=0;t<MAX_TRADES;t++)
  { ats_n[t]=0; ats[t]=0.0;
    ats_e[t].n=0; ats_e[t].sum=0.0; ats_e[t].sumsq=0.0;
  }

  for(e=0;e<n_exps;e++)
  { /*do one experiment*/


buy_init(buyers,pop1[i][G_BM],pop1[i][G_BD],pop1[i][G_GM],pop1[i][G_GD],pop1[i][G_MM],

pop1[i][G_MD],pop1[i][G_CAM],pop1[i][G_CAD],pop1[i][G_CRM],pop1[i][G_CRD],verbose);

sell_init(sellers,pop1[i][G_BM],pop1[i][G_BD],pop1[i][G_GM],pop1[i][G_GD],pop1[i][G_MM],

pop1[i][G_MD],pop1[i][G_CAM],pop1[i][G_CAD],pop1[i][G_CRM],pop1[i][G_CRD],verbose);


    d_sched_num=-1;
    s_sched_num=-1;

    for(d=0;d<expctl.n_days;d++)
    { /*one trading period or "day"*/

      /*set maximum number of trades in this day*/
      max_trades=expctl.max_trades;
      if(verbose) { fprintf(stdout,
                           "\nday %d: %d trades\n",d+1,max_trades);
                    fflush(stdout); }
      /*set things up for the start of the day*/
      day_init(e,d,ddat+d,&expctl,sellers,buyers,&p_0,&max_surplus,verbose);

      /*DC added this 020130 to allow for re-init of traders if/when schedules change*/
      if(d_sched_num!=expctl.d_sched)
      { /*Demand schedule has changed*/
        if(REINIT_ON_D_SCHED_CHANGE>0)
        {
buy_init(buyers,pop1[i][G_BM],pop1[i][G_BD],pop1[i][G_GM],pop1[i][G_GD],pop1[i][G_MM],

pop1[i][G_MD],pop1[i][G_CAM],pop1[i][G_CAD],pop1[i][G_CRM],pop1[i][G_CRD],verbose);
        }
        d_sched_num=expctl.d_sched;
      }
      if(s_sched_num!=expctl.s_sched)
      { /*Supply schedule has changed*/
        if(REINIT_ON_S_SCHED_CHANGE>0)
        {
sell_init(buyers,pop1[i][G_BM],pop1[i][G_BD],pop1[i][G_GM],pop1[i][G_GD],pop1[i][G_MM],

pop1[i][G_MD],pop1[i][G_CAM],pop1[i][G_CAD],pop1[i][G_CRM],pop1[i][G_CRD],verbose);
        }
        s_sched_num=expctl.s_sched;
      }

      n_buy=expctl.dem_sched[expctl.d_sched].n_agents;
      n_sell=expctl.sup_sched[expctl.s_sched].n_agents;

      surplus=0.0;
      n_trades=0;
      sigmasum=0.0;
      sum_price=0.0;
      sum_price_diff=0.0;

      bounds=NULL;


      for(t=0;t<max_trades;t++)
      { /*one trading session: either a trade occurs or a fail is recorded*/

        if(verbose) fprintf(stdout,"\nday %d trade %d\n",d,t+1);


        trade(&(tdat[d][t]),sellers,buyers,&expctl,
              max_surplus,&surplus,&status,pop1[i][G_QS][0],gactl.evo_Qs,
              verbose);

        /*calculate stats*/
        if(status==DEAL)
        { if(t>0) last_price=price;
          price=tdat[d][t].deal_p;
          if(t>0) sum_price_diff+=((price-last_price)*(price-last_price));

          pds=((price-p_0)*(price-p_0));
          (ats[n_trades])+=pds;
          (ats_n[n_trades])++;
          n_trades++;
          sum_price+=price;
          sigmasum+=pds;
          alpha=(100*sqrt(sigmasum/n_trades))/p_0;
          efficiency=(surplus/max_surplus)*100;
          if(verbose)
          { fprintf(stdout,"Day %d deal %d price=%f p_0=%f alpha=%f efficiency=%f\n",
                    d,n_trades,price,p_0,alpha,efficiency);
          }
        }
        else
        { if(status==END_DAY) /*give up*/
          { t=max_trades;} /*exit the t loop*/
        }
      } /*end of the trading session*/

      /*update the data for this day*/
      /*profit dispersion*/
      pd=0.0;
      for(b=0;b<n_buy;b++)
      { diff=((buyers[b].a_gain)-(buyers[b].t_gain));
        pd+=(diff*diff);
      }
      for(s=0;s<n_sell;s++)
      { diff=((sellers[s].a_gain)-(sellers[s].t_gain));
        pd+=(diff*diff);
      }
      pdisp=sqrt((1/((Real)(n_buy+n_sell)))*pd);

///fprintf(stdout,"TST >ddat[d].alpha.sum=%f ",ddat[d].alpha.sum);
      ddat_update(ddat+d,n_trades,sum_price,alpha,pdisp,efficiency,
                  sum_price_diff,verbose);
///fprintf(stdout,"TST <ddat[d].alpha.sum=%f \n",ddat[d].alpha.sum);
      if(verbose)
      { fprintf(stdout,"Exp %d end of day %d deal %d alpha=%f efficiency=%f\n",
                        e,d,n_trades,alpha,efficiency);
      }

    } /*end of the day loop*/

///    if(e==0) ///first experiment?
///    { sprintf(fname,"%s_trades.xg",expctl.id);
///      xg_trades_graph(tdat,ddat,expctl.n_days,max_trades,fname,n_exps);
///    }

    if(verbose) fprintf(stdout,"gen=%d i=%d experiment %d done\n",g,i,e);

  } /*end of the experiment loop*/

  /*plot the end-of-day stats in xgraph format*/
/*
  sprintf(fname,"%sres_day.xg",expctl.id);
  xg_daily_graph(ddat,expctl.n_days,n_exps,fname);
*/

  /*assign fitness*/
  fitness=0.0;
 /// fprintf(stdout,"fitness=%9.5e\n",fitness);
  for(d=0;d<expctl.n_days;d++)
  { /*DC introduced DIV0 catch here 010918*/
    if(ddat[d].alpha.n>0)
        { fitness+=fweight[d]*(ddat[d].alpha.sum/ddat[d].alpha.n);
          if(verbose)
          { fprintf(stdout,"fitness=%9.5e fw=%9.5e a.sum=%9.5e a.n=%d\n",
            fitness,fweight[d],ddat[d].alpha.sum,ddat[d].alpha.n);
          }
        }
    else
        { fitness+=fweight[d]*100.0; }

  }

  fitness=fitness/fweightsum;

  pop1[i][G_FIT][0]=fitness;

  if(fitness<elitefit)
  { elitefit=fitness;
    eliteid=i;
  }

  fprintf(stdout,"rep=%2d gen=%04d i=%03d nc=%1d fit=%9.5e Qs=%6.4f\n",
          rep,g,i,(int)pop1[i][G_NCASES][0],pop1[i][G_FIT][0],pop1[i][G_QS][0]);
  /*DC 020526: note new print order: gen#,i#,fitness,QS, #cases*(params)*/
  fprintf(fppopstats,"%04d %03d %9.5e %9.5e %1d ",
          g,i,pop1[i][G_FIT][0],pop1[i][G_QS][0],(int)pop1[i][G_NCASES][0]);
  for(c=0;c<MAX_N_CASES;c++)
  { fprintf(fppopstats,"%9.5e %9.5e %9.5e %9.5e %9.5e %9.5e %9.5e %9.5e %9.5e %9.5e ",
            pop1[i][G_BM][c],
            pop1[i][G_BD][c],
            pop1[i][G_GM][c],
            pop1[i][G_GD][c],
            pop1[i][G_MM][c],
            pop1[i][G_MD][c],
            pop1[i][G_CAM][c],
            pop1[i][G_CAD][c],
            pop1[i][G_CRM][c],
            pop1[i][G_CRD][c]);
  }
  fprintf(fppopstats,"\n");



  }/*end of i=0;i<POPSIZE loop*/



  /*breed new population*/

  /* DC added this, using exponential decay on annealing, 010918*/
  /// mutlim=pow(10.0,(log10(MUT_START)-((g*log10(MUT_START/MUT_END))/n_gens) ));
  /* DC changed this 020129 to effectively set n_gens=1000, giving
     annealing back-compatibility with longer
     experiments but allowing n_gens to be reduced to, say 250 or 500*/

  ///DC put it back to (n_gens-1) 030806 -- we just want to get the best results quickly
  ///mutlim=pow(10.0,(log10(gactl.mut_start)-
((g*log10(gactl.mut_start/gactl.mut_end))/1000) ));

  mutlim=pow(10.0,(log10(gactl.mut_start)-
((g*log10(gactl.mut_start/gactl.mut_end))/(n_gens-1)) ));

  fprintf(stdout,"gen=%d (of %d) mutlim=%f lastelite=%f\n",g,n_gens,mutlim,elitefit);

  /*DC changed 020531 so now we add N_KIDS from every breeding event*/
  if((gactl.n_kids<1)||(gactl.n_kids>2))
  { /*at the moment only 1 or 2 kids allowed*/
    fprintf(stderr,"\nFAIL: N_KIDS=%d illegal value\n",gactl.n_kids);
    exit(-1);
  }

  for(i=0;i<gactl.pop_size;i+=gactl.n_kids)
  {
    /*this is a real hack*/

    /*pick the first parent*/
    i1=irand(gactl.pop_size);
    i2=i1;
    while(i2==i1) i2=irand(gactl.pop_size);
    /*make sure i1 is the fittest one*/
    if(pop1[i2][G_FIT][0]<pop1[i1][G_FIT][0]) i1=i2; /*fitness for MIN score*/
    /*now i1 is an individual known to be fitter than at least one other*/

    /*find the second parent*/
    i2=i1;
    while(i2==i1)
    { /*same again for the second parent*/
      i2=irand(gactl.pop_size);
      i3=i2;
      while(i3==i2) i3=irand(gactl.pop_size);
      if(pop1[i3][G_FIT][0]<pop1[i2][G_FIT][0]) i2=i3; /*fitness for MIN score*/
    }
    /*now i2 is not i1 and is fitter than i3*/
```

```
if(pop1[i2][G_FIT][0]<pop1[i1][G_FIT][0]){i3=i1; i1=i2; i2=i3;}

/*now i1 and i2 are the two distinct parents, and i1 is fittest*/
/*slight preference for i1 as it's the fittest (e.g. if N_KIDS=1)*/

/*now copy with mutation and multipoint stochastic Xover*/

nx=0;

for(c=0;c<MAX_N_CASES;c++)
{
  for(j=0;j<GENE_LEN;j++)
  { /*mutate*/
    pop2[i][j][c]=pop1[i1][j][c]+mutlim-randval(2*mutlim);
    if(gactl.n_kids>1) pop2[i+1][j][c]=pop1[i2][j][c]+mutlim-randval(2*mutlim);

    /*clip: NB THIS COULD BE REFLECT NOT TRUNCATE (CF SETH ECAL PAPER)*/
    Rclip01(&(pop2[i][j][c]));
    if(gactl.n_kids>1) Rclip01(&(pop2[i+1][j][c]));

    /*swap the parent we are copying from?*/
    if(randval(1.0)<gactl.pr_xover) {i3=i1; i1=i2; i2=i3; nx++;} /*swap parent*/
  }

  /*do the shout prob*/
  /*cross?*/
  if(randval(1.0)<gactl.pr_xover) {i3=i1; i1=i2; i2=i3; nx++;} /*swap parent*/
  /*mutate*/
  pop2[i][G_QS][0]=pop1[i1][G_QS][0]+mutlim-randval(2*mutlim);
  if(gactl.n_kids>1) pop2[i+1][G_QS][0]=pop1[i2][G_QS][0]+mutlim-randval(2*mutlim);
  /*clip*/
  Rclip01(&(pop2[i][G_QS][0]));
  if(gactl.n_kids>1) Rclip01(&(pop2[i+1][G_QS][0]));

  /*do the n_cases*/
  /*cross?*/
  if(randval(1.0)<gactl.pr_xover) {i3=i1; i1=i2; i2=i3; nx++;} /*swap parent*/
  /*mutate?*/
  j=(int)pop1[i1][G_NCASES][0];
  if(randval(1.0)<gactl.pr_casemut)
  {
    while(j==pop1[i1][G_NCASES][0])
    { j=irand(4)*2;
      if(j<1) j=1;
    }
  }
  pop2[i][G_NCASES][0]=(Real)j;

  if(gactl.n_kids>1)
  { j=(int)pop1[i2][G_NCASES][0];
    /*mutate?*/
    if(randval(1.0)<gactl.pr_casemut)
    {
      while(j==pop1[i2][G_NCASES][0])
      { j=irand(4)*2;
        if(j<1) j=1;
      }
    }
    pop2[i+1][G_NCASES][0]=(Real)j;
  }



  /*some adjustment to make sure genotype values reflect actual usage*/
  /*this clipping occurs in agent_init anyway, but this keeps genomes meaningful*/
  if(pop2[i][G_BM][c]+pop2[i][G_BD][c]>1.0) pop2[i][G_BD][c]=1.0-pop2[i][G_BM][c];
  if(pop2[i][G_GM][c]+pop2[i][G_GD][c]>1.0) pop2[i][G_GD][c]=1.0-pop2[i][G_GM][c];
  if(pop2[i][G_MM][c]+pop2[i][G_MD][c]>1.0) pop2[i][G_MD][c]=1.0-pop2[i][G_MM][c];

  if(gactl.n_kids>1) if(pop2[i+1][G_BM][c]+pop2[i+1][G_BD][c]>1.0)
                     pop2[i+1][G_BD][c]=1.0-pop2[i+1][G_BM][c];
      if(gactl.n_kids>1) if(pop2[i+1][G_GM][c]+pop2[i+1][G_GD][c]>1.0)
                         pop2[i+1][G_GD][c]=1.0-pop2[i+1][G_GM][c];
      if(gactl.n_kids>1) if(pop2[i+1][G_MM][c]+pop2[i+1][G_MD][c]>1.0)
                         pop2[i+1][G_MD][c]=1.0-pop2[i+1][G_MM][c];

    } /*end c loop*/

/*
    for(i3=0;i3<8;i3++)
      { fprintf(stdout,"%5.3f ",pop2[i][i3][0]); }
    fprintf(stdout," (%d crossovers)\n",nx);
*/
  } /*end i loop*/

  /*ELITISM: copy elite of pop1 into pop2[0]*/

  for(j=0;j<GENE_LEN;j++)
  { for(c=0;c<MAX_N_CASES;c++)
    { pop2[0][j][c]=pop1[eliteid][j][c];
      pop2[0][j][c]=pop1[eliteid][j][c];
    }
  }

  /*DC 020526: note new print order: gen#,i#,fitness,QS, (cases*params)*/
  fprintf(fpelitestats,"%04d %03d %9.5e %9.5e %1d ",
          g,eliteid,pop1[eliteid][G_FIT][0],pop1[eliteid][G_QS][0],
         (int)pop1[eliteid][G_NCASES][0]);
  for(c=0;c<MAX_N_CASES;c++)
  { fprintf(fpelitestats,"%9.5e %9.5e %9.5e %9.5e %9.5e %9.5e %9.5e %9.5e %9.5e %9.5e ",
            pop1[eliteid][G_BM][c],
            pop1[eliteid][G_BD][c],
            pop1[eliteid][G_GM][c],
            pop1[eliteid][G_GD][c],
            pop1[eliteid][G_MM][c],
            pop1[eliteid][G_MD][c],
            pop1[eliteid][G_CAM][c],
            pop1[eliteid][G_CAD][c],
            pop1[eliteid][G_CRM][c],
            pop1[eliteid][G_CRD][c]);
  }
  fprintf(fpelitestats,"\n");
  fflush(fpelitestats);
  fflush(fppopstats);

  /*copy pop2 into pop1*/
  for(i=0;i<gactl.pop_size;i++)
  { for(j=0;j<GENE_LEN;j++)
    { for(c=0;c<MAX_N_CASES;c++)
      { pop1[i][j][c]=pop2[i][j][c];
      }
    }
  }

  /*}*/


  }/*end of generation loop*/


  fprintf(stdout,"Experiment done!");

  fclose(fpelitestats);
  fclose(fppopstats);

  } /*end of reps loop*/

  return(1);
}
```