Monte-Carlo Evaluation of Trading Systems

There are few things that a trader enjoys more than designing an automated trading system, backtesting it, and watching it perform well on the backtest. The dreams of riches to come are delightful. Unfortunately, it is always possible that luck played more of a role in the system's performance than inherent quality. Even a truly worthless system, one that makes its position decisions based on rules that are little more than rolls of celestial dice, can experience fortuitous pairing of positions with market moves. This is especially true if the trader has experimented with several competing systems and chosen the best. It is vital that the trader estimate the probability that a worthless system could perform as well as the candidate performed. Unless one finds that this probability is very small, one should be suspicious of the system.

One method of testing the performance is to assume that the hypothetical population from which the historical returns were drawn has a true mean of zero, and then compute the probability that results as good as those observed could have arisen by luck. If one is willing to assume that the returns are independent draws from a normal distribution, the ordinary single-sample *t-test* is appropriate. If, as is more reasonable, normality cannot be safely assumed, a bootstrap test will do a fairly decent job at estimating this probability.

There is a second, often superior approach to handling the concept of worthlessness in a trading system. Rather than defining worthlessness as the returns being drawn from a population having a mean return of zero, we define worthlessness as the system's position decisions being randomly placed, unrelated to subsequent market moves. This leads to an entirely different test, a useful alternative to the bootstrap.

The fundamental weakness of the bootstrap is its reliance on the assumption that the empirical distribution of the obtained sample is representative of the population distribution. This is what allows us to assert that the bootstrap distribution of the test statistic mirrors its population distribution. As is well known, this assumption is sometimes intrinsically flawed. Even when it is a reasonable assumption on average, sometimes the experimenter will be unlucky and obtain a sample whose dispersion seriously misrepresents that of the population. For this reason, bootstrap tests always lie under a cloud of suspicion.

Many automated market-trading systems are amenable to analysis by *Monte-Carlo permutation* simulation, an excellent though limited alternative to the bootstrap. The beauty of a Monte-Carlo permutation test is that it can reveal the permutation-null-hypothesis distribution of nearly any reasonable test statistic, and it can do so to whatever accuracy is desired, limited only by available computational resources. The test's dependence on the distribution of the obtained sample is greatly reduced compared to the bootstrap.

The weakness of Monte-Carlo permutation is that the trading model must fulfill strict requirements in its design. This document discusses a common, easily performed Monte-Carlo simulation that has fairly broad applicability.

The Permutation Principle

We begin with a brief mathematical introduction to the theory underlying the technique. Suppose we have a scalar-valued function of a vector. We'll call this $g(v)$. In a market-trading scenario, v would be the vector of market changes and $g(.)$ would be a performance measure for our trading system.

For example, suppose that every day, at the close of the market, we compute some indicator variables, submit them to a model, and decide whether we want to be long, short, or neutral the next day. The following morning we set our position accordingly, and we close the position the morning after we opened. Thus, the position is in effect for exactly one day. If we are neutral, our equity does not change. If we are long, our equity changes by the amount that the market changed. If we are short, our equity changes by the negative of the market's change. Then v is the vector of next-day market changes that we encountered, and $g(v)$ is a measure of our performance over the entire test period. This may be our total return, percent return, Sharpe Ratio, or any other reasonable performance figure.

Let $\Phi(.)$ be a permutation. In other words, $\Phi(v)$ is the vector v rearranged to a different order. Suppose v has n elements. Then there are $n!$ possible permutations. We can index these as Φ_i where i ranges from 1 through $n!$. For the moment, assume that the function value of every permutation is different: $g(\Phi_i(v)) \neq g(\Phi_j(v))$ when $i \neq j$. We'll discuss ties later.

Let K be a random integer uniformly distributed over the range 1 through $n!$, and let k be an instance of this random variable. Define Φ as Φ_k. Later we will refer to this as the *original* permutation because it is the permutation of v that is observed in an experiment. Now draw from this population m more times and similarly define Φ_1 through Φ_m. Again for the moment assume that we force these m+1 draws to be unique, perhaps by doing the draws without replacement. We'll handle ties later.

Compute $g(\Phi(v))$ and $g(\Phi_i(v))$ for i from 1 through m. Define the statistic Θ as the fraction of these m+1 values that are less than or equal to $g(\Phi(v))$. Then the distribution of Θ does not depend on the labeling of the permutations, or on $g(.)$, or on v. In fact, Θ follows a uniform distribution over the values $1/(m+1)$, $2/(m+1)$, ..., 1. This is easy to see. Sort these m+1 values in increasing order. Because each of the draws that index the permutations has equal probability, and because we are (temporarily) assuming that there will be no ties, the order is unique. Therefore, $g(\Phi(v))$ may occupy any of the m+1 ordered positions with equal probability.

Let $F(\Theta)$ be the cumulative distribution function of Θ. As m increases, $F(\Theta)$ converges to a continuous uniform distribution on (0,1). In other words, the probability that Θ will be less than or equal to, say, 0.05 will equal 0.05, and the probability that Θ will exceed, say, 0.99 will be 0.01, and so forth.

We can use this fact to define a statistical test of the null hypothesis that Φ, our original permutation, is indeed a random draw from among the n! possible permutations, as opposed to being a special permutation that was chosen by virtue of its having an unusually large or small value of $g(\Phi(v))$. To perform a left-tail test, set a threshold equal to the desired p-value, and reject the null hypothesis if the observed Θ is below the threshold. To perform a right-tail test, set a threshold equal to one minus the desired p-value, and reject the null hypothesis if the observed Θ is above the threshold.

We have conveniently assumed that every permutation gives rise to a unique function value, and that every randomly chosen permutation is unique. This precludes ties. However, the experimental situation may prevent us from avoiding tied function values, and selecting unique permutations is tedious. We are best off simply taking possible ties into account. First, observe that when comparing $g(\Phi(v))$ to its m compatriots, tied values that are strictly above or below $g(\Phi(v))$ are irrelevant. We only need to worry about ties at $g(\Phi(v)$. A left-tail test will become conservative in this case. Unfortunately, a right-tail test will become anti-conservative. The solution is simple: Shift the count boundary to the low end of the set of ties. Note that the code shown later actually computes conservative p-values directly, and it slightly modifies the counting procedure accordingly.

Remember that an utterly crucial assumption for this test is that when the null hypothesis is true, all of the n! possible permutations, including of course the original one, have an equal chance of appearing, both in real life and in the process of randomly selecting m of them to perform the test. Violations of this assumption can creep into an application in subtle ways. Some of these ways will be addressed in this document, but ***the user is ultimately responsible for verifying the veracity of this assumption.***

We end this section by bringing the discussion back down to the application level. Look back to the day-trading example that began this section. The null hypothesis is that the order in which the daily market returns lined up with positions chosen by the trading algorithm is random.

The alternative is that they lined up in a way that improved performance beyond what could be expected by random chance. To test this null hypothesis, we compute the net return of the system. Then we randomly permute the daily market returns and compute the net return for the permutation. Note this result. Repeat the permute-and-evaluate procedure several thousand times. If only a tiny fraction of the random results exceed the performance of the original system, we conclude that our system provides significantly better returns than chance would provide, and we rejoice.

The Trading Scenario

In an automated trading scenario that is amenable to Monte-Carlo simulation, the trader is presented with a large number of trading opportunities. Each time an opportunity presents itself, the trader may choose to take a long position, take a short position, or remain neutral. Each opportunity is associated with a raw return (market change), which may be positive or negative. If the trader takes a long position, the return is added to the account. If a short position is taken, the return is subtracted from the account. If the trader remains neutral, the account remains constant. Note that since the raw market return may be negative, a long position can result in a loss and a short position can result in a gain.

We will remain deliberately vague about the nature of the return. It may be measured in dollars or market points, either of which implies that we are not compounding the returns. If we wish to assume compounding, each return would be the log of the ratio of market value at the end of the trade to the market value at the beginning of the trade. The returns may or may not be annualized. Your choice of how to measure each return is irrelevant to the subject at hand.

What is a trading opportunity? There are many possibilities. Here are a few thoughts regarding possible entry and exit strategies:

1. At the end of each day you perform some calculations and decide what position you will take the next day. If you choose to enter the market, you open the position when the market opens the next morning.

2. Every time a defined event occurs, such as the market retreating two percent from its ten-day high, you perform your entry calculations and act accordingly.

3. You might choose to close the position after a predefined time period (such as one day) elapses.

4. You might choose to close the position after a predefined market move (such as two points up or down) occurs. Deliberately unbalanced moves are potentially troublesome, as discussed on Page 15.

5. In order to avoid compromising the results, the decision to open a position ideally should not depend on whether a position is already open. The implication of this fact is that you need to either be ready to have multiple positions open, or the duration of each position must be short enough that there is a negligible probability of opening another position while one is still open. This can be easily satisfied by making sure that trading opportunities are defined so as to never overlap. Reversal systems automatically fulfill this requirement.

The Monte-Carlo simulation presented here does not test the impact of the definition of a trading opportunity. Rather, it tests the quality of the model that chooses to be long, short, or neutral when the trading opportunity arises. A trading opportunity is an arbitrary designation, and any decision about opening a position is based on information that is obtained when the trading opportunity presents itself. The obvious example of this situation is when calculations are done at the end of a day to make a decision about the position to take the next morning. In this case, each morning is a trading opportunity, and the long/short/neutral decision is based on information gathered from recent history.

One must be careful to preserve this characteristic when more complex trading opportunities are designed. For example, suppose we were to define a trading opportunity as a morning in which yesterday's range was at least ten percent higher than the prior day's range. It may be that this indicator alone has predictive power. If so, Monte-Carlo simulation would not detect this power. So we should instead define a trading opportunity as the morning, and let the range information be a component of the position decision. In other words, the definition of a trading opportunity should be as innocent as possible, since any predictive power in the definition will be ignored. Also, trading opportunities should be as frequent as possible so as to produce a large number of them. Of course, they must not be so frequent as to produce an impractical number of overlapped trades. Recall that in order for Monte-Carlo simulation to be valid we must always be able to open a trade when the trading rule calls for it, regardless of whether a position is already open, and overlapped positions compromise the statistical validity of results.

The fruit of a candidate trading rule, whose quality we wish to evaluate, can be represented by a collection of pairs of information: a raw market return and a long/short/neutral indicator. For every long position, the obtained return will be equal to the raw return. For every short position, the obtained return will be the raw return times minus one. For every neutral position, the obtained return will be zero. The performance of a trading system may look like this:

Opportunity	Position	Raw	Obtained
1	L	3.2	3.2
2	N	2.7	0.0
3	L	–4.8	–4.8
4	S	1.6	–1.6
...			

If the rule that determines the position to take each time a trading opportunity arises is an intelligent rule, the sum of the obtained returns will be larger than the sum that could be expected from a rule that assigns positions randomly. An intelligent rule will tend to associate long positions with positive raw returns and short positions with negative raw returns.

This effective pairing is precisely what is tested by Monte-Carlo simulation. If we could try every possible pairing of a position with a raw market return, we could tabulate the total return from each arrangement and compute the fraction of these returns that equal or exceed the return enjoyed by our candidate model. This would be the probability that the candidate model could have done as well as it did by sheer luck. This probability is in the universe of models that have exactly as many long, short, and neutral positions as the candidate model. Some of the implications of this fact will be discussed later. For now, understand that we are concerned with the quality of the candidate model relative to all possible rearrangements of its pairing of trades with raw returns.

If there are *n* trading opportunities, there are *n*-factorial possible arrangements of the pairs, although many of these would be identical due to the fact that the positions are restricted to three values. This is $n(n–1)(n–2)$..., which is a gigantic number when *n* is large. We obviously cannot tabulate every possible ordering, but we can do something that is almost as good. We can try a large number of random pairings and assume that the resulting distribution of total returns is an effective surrogate for the true distribution. As long as *n* is large (many would consider 50 to be a minimum, and 200 or more desirable), and we test at least several thousand random pairings, it is safe to treat the obtained distribution as a good approximation to the real thing.

What We Test Versus What We Want To Test

The Monte-Carlo simulation presented here tests the null hypothesis that the pairing of long/short/neutral positions with raw returns is random. Implicit in this test is the assumption that all pairings are equally likely under the null hypothesis. The alternative hypothesis is that the candidate model is so intelligent that it is able to pair positions with raw returns in such a way that a superior total return is produced. This test is performed by generating a large number of random pairings, computing the total return of each, and finding the fraction of this collection of random pairings whose total return equals or exceeds that of the candidate model. This fraction is the probability that the supposedly superior return of the candidate model could have been obtained by nothing more than luck. If we find that this probability is small, we rejoice.

The null hypothesis tested by this Monte-Carlo simulation may not be the null hypothesis we think we are testing. For example, it does not test the null hypothesis that the expected return of the model is zero, which is what traditional bootstrap algorithms usually test. Suppose the market has a strong upward bias, which implies that the raw returns have a positive mean. Suppose also that the candidate model capitalizes on this by favoring long positions over short. In this situation, all rearranged pairings will tend to have a positive total return. Thus, algorithms such as the traditional bootstrap that test the null hypothesis of zero expected return will be more inclined to reject this hypothesis than the Monte-Carlo simulation.

This is an important issue. Suppose the trading system is intelligent enough to know that it should be long more often than it is short, but not intelligent enough to come up with useful pairings. In other words, at each trading opportunity the model flips a biased coin, taking a long position with higher probability than a short position. Do we call this a good model? The Monte-Carlo permutation test will say no, while most other traditional methods will say yes.

We can be even more specific. Every beginner knows that the S&P 500 has a huge historical upward bias. Is this information something that an intelligent trading model should use? Do we want to count on this upward bias continuing? One school of thought says that the upward bias in this market is the largest, most reliable characteristic of the market, and a trader would be crazy to ignore it. Another school of thought says that the bias can vanish at any moment and remain hidden for a very long time. Trading systems that can weather such reversals need to ignore the upward bias. Which school of thought is correct? If I knew the answers to questions like this, I'd be living in Bermuda right now.

Probably the closest we can come to answering this question is by noting that if we ignore market bias and judge a model only on its ability to find a superior pairing, we can always incorporate the bias later, and if we are correct in assuming that the bias will continue, improve the model further by doing so. In other words, we can separate performance into a bias component and a beat-the-odds component. Thus, it seems reasonable to remove long-short prejudice from consideration as an element of quality. This automatically happens with Monte-Carlo simulation when the measure of quality is the total return. But some other measures fail to achieve this immunity. Moreover, it can be instructive to compare Monte-Carlo techniques with bootstrap techniques, which do not enjoy this immunity when the market is biased. Hence we need to devise a universal method for immunizing quality measures against long/short prejudice.

An excellent solution is to compute the expected return of a random trading system that has the same number of long, short and neutral positions as the candidate system. This is the portion of the total return of the candidate system that is attributable to long/short imbalance interacting with market bias. When this is subtracted from the total return, the remainder is the portion of the total that is attributable to beating the odds.

The expected value of a single raw return drawn from the population is, by definition, the mean of these returns. This is expressed in Equation (1), in which the raw market returns are R_i for i=1, ..., n.

$$E_{Raw} = \frac{1}{n} \sum_{i=1}^{n} R_i \tag{1}$$

Let P_i signify the position of a trading system at opportunity i. This will be long, short, or neutral. The expected return of a trading system in which raw returns are randomly paired with positions is shown in Equation (2).

$$E_{Random} = \sum_{P_i = Long} E_{Raw} - \sum_{P_i = Short} E_{Raw} \tag{2}$$

The total return of the candidate system is the sum of the raw returns at those times when the candidate system is long, minus the sum of the raw returns when the system is short. This is expressed in Equation (3).

$$Total = \sum_{P_i = Long} R_i - \sum_{P_i = Short} R_i \tag{3}$$

The total return of the candidate system is corrected for long/short prejudice by subtracting the expectation of a similarly prejudiced random system. This is shown in Equation (4), which is Equation (3) minus Equation (2).

$$Corrected = \sum_{P_i = Long} \left(R_i - E_{Raw} \right) - \sum_{P_i = Short} \left(R_i - E_{Raw} \right) \tag{4}$$

Notice that the corrected return of Equation (4) is identical to the uncorrected return of Equation (3) except that the mean of the raw returns is subtracted from each individual raw return. In other words, to remove the effect of long/short imbalance in a biased market, all you need to do is center the raw returns. This is intuitive; to remove the impact of bias you simply remove the bias. But it's nice to see solid theoretical support for doing so, rather than relying on intuition. By removing bias this way, which is implicitly done by the Monte-Carlo permutation test, other statistical tests can be made to behave more like the permutation test.

Serial Dependence

Another consideration is the impact of serial dependence within the raw returns and the positions taken by the candidate system. If either the raw returns or the positions are serially independent, random rearrangement of the pairs is a reasonable way to generate a null distribution. But what if both the positions and the raw returns are serially dependent? This can easily happen. For example, moving-average crossover systems applied to serially dependent raw returns will produce positions that are serially dependent themselves. In such situations, fully random reordering of the pairings will produce trial systems that would not normally appear in the context of the trading rule. So is randomizing pairs a valid way to test the null hypothesis of worthless pairing against the alternative that the candidate system is intelligent? This is an arguable point, but I claim that the answer is usually yes, tinged with only a hint of unease.

The argument hinges on the implied nature of the universe in which the null hypothesis dwells. Suppose that the experimenter is interested in only models that exhibit substantial serial dependence within both the raw returns and the positions taken. He or she would be inclined to ask the following question: What is the probability that, among all such serially dependent but otherwise random (worthless) models, a model that was as good as the candidate would have been encountered by virtue of pure luck? In this situation, the Monte-Carlo technique of this chapter would be incorrect. Instead, the experimenter would need to devise a way to simulate random pairings that all exhibit the desired degree of serial dependence. This would be difficult at best, and perhaps impossible.

But I claim that this is often the wrong question. Why limit the null universe to serial dependence? It may be that the serial dependence within the positions is a critical component of the intelligence of the trading system. Limiting the null universe to such serial dependence would discount its importance, leading to an overly conservative test. If the intelligent response to serial dependence in the raw returns is serial dependence in the positions, so be it. I would like the testing algorithm to credit such intelligence instead of discounting it. By basing the null distribution on fully random pairing, we give credit where, I believe, credit is usually due. Feel free to disagree if you wish, because there are good arguments to the contrary. But at least understand the situation. Figures 1 and 2 shed more light on the issue.

Figure 1 shows the attained rejection rate versus the presumed rejection rate for 100 trading opportunities with raw returns having no serial correlation. Half of the positions are long and half are short. The short dashed line is for the percentile bootstrap, the long dashed line is for the basic bootstrap, and the dotted line is for the Monte-Carlo method. This latter method is the closest to optimality, a solid line. In fact, its departures from optimality are due only to random sampling error. Monte-Carlo simulation would approach perfection as the number of trials is increased. As is often the case, the bootstrap methods are slightly anti-conservative.

Figure 2 is the same scenario except that the raw returns have a lag-one serial correlation of 0.1, which is large for a commodity. Also, all long positions are contiguous, as are all short positions. This, of course, is extreme serial correlation, far beyond what one would normally encounter in practice. Observe that Monte-Carlo simulation has become somewhat anti-conservative, though not terribly so. The two bootstrap algorithms have suffered even more, although a dependent bootstrap would probably help their performance.

This illustrates the heart of the matter. The rules tested in Figure 2 are random by design in that they choose their positions without looking at the associated raw returns. But they are also serially correlated, as are the raw returns. Monte-Carlo simulation rejects the null hypothesis of worthlessness more often than would be expected for totally random rules. This is undoubtedly a flaw, at least in this correlated-by-design simulation. But serial correlation can be argued to be a type of non-randomness that can arise intelligently. Since the raw returns themselves exhibit serial correlation, it is reasonable to expect that profitable models would tend to be serially correlated in their positions. So by limiting the rules that went into generating Figure 2 to only serially correlated rules, we are working with a population of candidate rules that, although otherwise random, are inclined to be the types of rules that would succeed in a market whose raw returns are serially correlated. Thus we would expect that an above-average number of these rules would have superior returns. In other words, the Monte-Carlo simulation has detected the fact that in a serially correlated market, rules that perform well tend to have serially correlated positions. It is my opinion that this modest degree of anti-conservative behavior is more tolerable than the extreme conservatism we would suffer by limiting the null distribution to serial correlation (if we even could!).

Actually, this whole discussion probably blows the issue far out of proportion. The simulation shown in Figure 2 uses data that is considerably more correlated than would be encountered in most markets, and the positions are the ultimate in correlated: one clump of contiguous longs, and one clump of contiguous shorts. No realistic trading system would take this extreme approach. Yet even so, at a significance level of 0.1 we get rejection at a rate of about 0.12. This does not demonstrate a serious problem until we get far out into the tail.

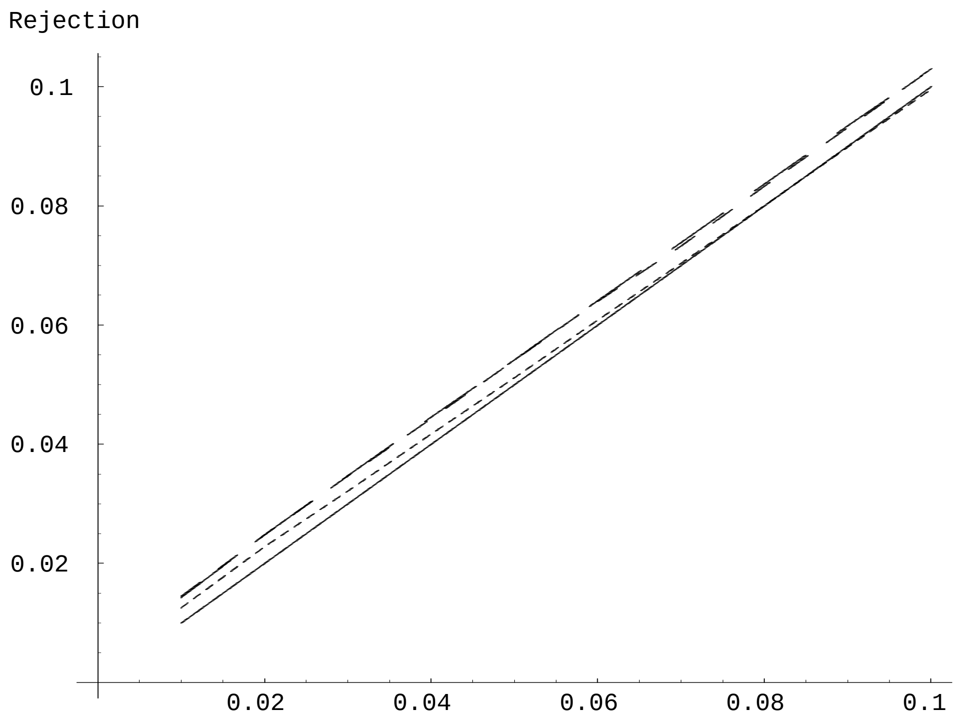

Figure 1 Rejection rates with no serial correlation.

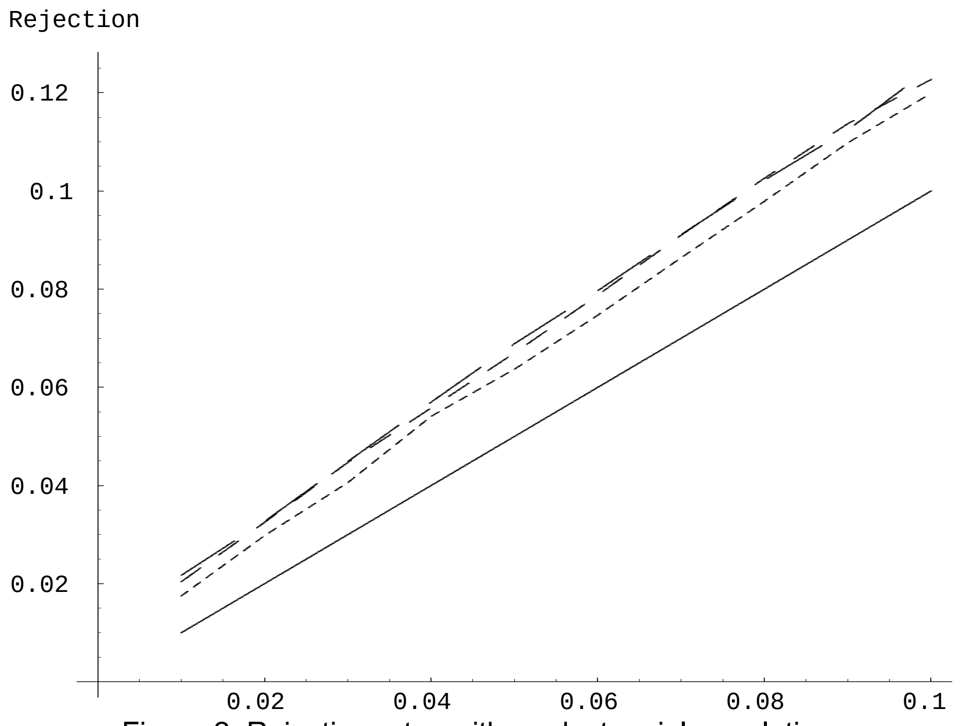

Figure 2 Rejection rates with modest serial correlation.

Nonstationarity

Nonstationarity of the raw returns and positions might be an issue. When nonstationarity is obvious, one might be tempted to break the series of pairs into several subseries, each of which we assume to be stationary, and permute each separately. But in my opinion, the argument against this is the same as the argument against limiting the null distribution in the case of serial dependence. Nonstationarity of the positions may well be an intelligent response of the model to nonstationarity of the raw returns. In this case we want to reward the model.

For example, suppose we apply an effective trend-following model to a market that has several extended trends. The model will favor long positions during upward trends and short positions during downward trends. If we were to break up the series into subseries according to trend and permute each separately, the null distribution would be centered around a positive number because the return from each subseries will be decent regardless of how it is permuted. Every upward trending subseries will enjoy mostly long positions, so every permutation will favor a positive return. The same applies to downward trending series with mostly short positions. This will reduce the significance of the candidate system. Only full permutation will reveal the quality of the candidate system by comparing it to random systems that fail to make the intelligent connection between trend and effective position.

Thin Position Vectors

The final consideration can be confusing, although it is the least troublesome in practice. Suppose for the moment that the raw returns have zero mean (which we should generally ensure by centering, although this has no effect on the Monte-Carlo permutation test) and are independent with a symmetric distribution. This is usually fairly close to reality for most markets, though it is almost never satisfied completely. As shown in Equation (2), the expected return of any random system, regardless of its quantity of long, short, and neutral positions, will be zero because the mean of every raw return is zero.

The standard deviation of the total return is another matter entirely. One extreme situation is that the candidate system may have an open position (long or short) exactly once out of *n* trading opportunities. The standard deviation of the total return among all permutations will equal that of the raw returns. More generally, we may have exactly *k* open positions in the candidate system. The standard deviation of the total return among the permutations will be approximately sqrt(*k*) times the standard deviation of the raw returns, with the exact factor depending on the position vector. In other words, the dispersion of the permuted returns depends on the number of open positions. Working with mean or annualized returns will not help because such alternatives will just be rescaled values of the total return. There is no getting around the fact that the number of open positions in the candidate system affects the dispersion of the Monte-Carlo null distribution, and hence the computed probability associated with a candidate.

Why does this at first seem to be a problem? Recall that our Monte-Carlo simulation tests the null hypothesis that the order in which the trading model placed its trades is random. The alternative is that the order of the trades was intelligently designed so as to produce a superior return. We are able to compute the probability that, among random systems *that have exactly as many long, short, and neutral positions as the candidate model*, a return as good as that obtained by the candidate model could have been obtained by sheer luck. This is a conditional probability.

Now suppose your model has 1000 trading opportunities, and it decides to be long 187 times, short 49 times, and neutral the rest of the time. The Monte-Carlo randomization test will tell you the probability that a worthlessly random model having precisely this number of long, short, and neutral positions could have done as well as your candidate model by luck alone. However, your real null universe is not systems that have these exact position quantities. Rather, you are dwelling in a more general universe in which models will be long perhaps 10 to 20 percent of the time, short another 10 to 20 percent of the time, and neutral otherwise. If you had consumed three cups of coffee this morning instead of two, perhaps your candidate model would have been long 103 times, short 107 times, and neutral the remainder of the time. So the quantity that you really want to compute is the probability that a worthless model from this more general universe could have done as well as your candidate by nothing more than luck. Not surprisingly, this quantity is difficult or impossible to compute. Of course you could specify probability distributions for the number of long and short positions and perform a huge Monte-Carlo simulation in which you generate a vast number of random models having a variety of position counts. This would give you a probability in the larger universe. But is this probability any more useful than that obtained by basic randomization? No.

Imagine that a vast number of experimenters in a vast number of locations are devising candidate models having 1000 trading opportunities. Segregate their results into bins according to their number of long, short, and neutral positions. As we already know, within each bin the Monte-Carlo test will work correctly. In other words, within any particular bin, worthless models will reject the null hypothesis at a p=0.1 level ten percent of the time on average, at a p=0.05 level five percent of the time, et cetera. But if the rejection rate in each bin is correct, the grand rejection rate in the larger universe is also correct. This is admittedly a very heuristic argument, but it captures the essence of the situation.

The bottom line is that each randomization test is valid for its candidate model only. You cannot generate a Monte-Carlo null distribution using a candidate system having long/short/neutral positions of 187/49/764 and then use this distribution to test a system with positions of 103/107/790.

These two candidate rules will have null distributions with different dispersions. But in practice this would never be a problem anyway. Use each candidate rule to generate its own null distribution and everything will be fine.

Avoid Using a Generic Null Distribution

There is one situation in which the experimenter may be tempted to interchange null distributions and tests. It would be convenient to generate just one null distribution in advance from a generic candidate model, and then judge all subsequent models based on this distribution. This would seem to be reasonable for trading systems that are, by design, always in the market, regularly switching between long and short. In this case, all null distributions would have same number of neutral positions: none. Hence, the spread of the dispersion would seem to be unaffected by varying the position vector.

Nonetheless, this is dangerous. If the distribution of the raw returns is not symmetric, the distribution of total returns under permutation will depend on the relative number of long and short positions. As an extreme example, suppose that there are 1000 centered raw returns, and that 999 of them are +1 and one of them is –999. Also suppose that the candidate system has 999 long positions and one short position. The Monte-Carlo null distribution will have a long right tail due to the rare alignment of the –999 return with the single short position. Conversely, if the candidate system has 999 short positions and one long, the null distribution will have a long left tail due to the single long position rarely being matched with the –999 raw return. This is admittedly an extreme example. Nonetheless, it is important to be aware that the long/short ratio can have an impact on the null distribution when the distribution of raw returns is significantly asymmetric. Once again, the admonition is to let a candidate model generate its own null distribution.

When Are We Unable To Use a Monte-Carlo Test?

It should be clear by now that the Monte-Carlo permutation test is an excellent way to test the null hypothesis that a trading system's matching of positions to raw returns is random as opposed to intelligent. Why don't we do it all the time, instead of relying on parametric tests that have possibly unrealistic assumptions, or bootstrap tests that are notoriously unstable? The answer is that sometimes we simply can't do it. We may not have the requisite information, or the trading system may have been designed in a way that is incompatible with Monte-Carlo randomization tests. Most other tests require only a set of trade results. For example, if someone hands you a set of 100 profit/loss figures from the 100 trades that he did last year, you could perform a bootstrap test of the null hypothesis that the mean of the population from which these trades were taken is zero. But in order to perform a Monte-Carlo randomization test, you would need far more information than just these 100 numbers. At a minimum, you would need the raw market return and his position for each trade. In addition, if he is ever out of the market, you ideally should have the complete set of raw potential profits and losses for every trading opportunity he had, whether that opportunity was taken or not. This is because you need to know how much profit was foregone with every opportunity not taken, and how much profit would have been foregone had opportunities taken been passed by. If the system has numerous long and short positions, the Monte-Carlo permutation test can be performed on just times when a position is open. But this shortcut sacrifices considerable power and stability.

This is not an impossible requirement, though, especially if the trading system is designed with Monte-Carlo testing in mind. You need to be able to rigorously define every trading opportunity and associate a raw return with each opportunity. This is the most important requirement.

There are several more considerations. In order to avoid possible trade interactions, it is best if the trading opportunities do not overlap. However, in practice this is not strictly necessary as long as any overlap is not extensive. Finally, you need to remember that the hypothesis being tested concerns the order in which positions and raw returns are paired. The hypothesis does not concern the actual return of the system, which is usually the more pertinent question. It is vital that these alternative viewpoints be compatible in your experimental situation. In particular, the randomizations performed in the Monte-Carlo simulation must be a reasonable representation of the possibilities that exist in real life. If the set of Monte-Carlo randomizations includes a significant number of pairings that, due to the design of the experiment, would be unlikely in real life, the test will not be valid.

We can elaborate on these issues with a situation that may present itself occasionally. Suppose someone gives us an extended trading history defined by entry and exit dates and returns for each period in the market. Ideally, all entries and exits occurred at the open or close of a market so that no day was ever split by having an open position for just part of the day. Partial days would require approximations to full day returns. The best way to handle a set of dates and returns would be to break it into positions and returns for each individual day and apply Monte-Carlo randomization to the complete set of days, including all days in which the trader was out of the market. This forces the trades into the ideal structure for the test.

But it may be that such a detailed breakdown is difficult and we want a shortcut. We would still need the raw return of each neutral period. If we cannot procure this information, Monte-Carlo randomization is compromised, perhaps severely. So what we possess for our simplified method is a net return for each long period, a net return for each short period, and a hypothetical net return for each neutral period. This would be far fewer numbers than if we broke down the long/short/neutral periods into individual days. Can we perform Monte-Carlo randomization on this data?

The answer is that we can, but it is dangerous. Suppose, for example, that the trading system is in the market for short periods of time. Perhaps when it takes a long position it is in the market for just one day, and when it takes a short position it is in the market for roughly five days. Between these trades the system is neutral for extended periods of many days. Suppose also that the system is completely worthless, taking its positions randomly. The raw return associated with each long position will have a relatively small variance because each long return is the raw return of a single day. At the other extreme, the raw return associated with each neutral position will have a relatively large variance because it is the sum of the raw returns of many days.

Now think about what happens when we randomly permute the pairings of positions with raw returns in order to generate the supposed null hypothesis distribution of total returns of worthless systems. Many of these random permutations will pair long positions with raw returns having unnaturally large variance, and simultaneously pair neutral positions with raw returns having unnaturally small variance. Such pairings would rarely, if ever, occur in actual random realizations of the trading system. Therefore, the Monte-Carlo distribution would not be an accurate representation of the real-world distribution of returns from similar worthless systems.

Note that the problem is *not* that the raw returns have different variances. The Monte-Carlo test described in this chapter does not require that the raw returns have equal variances, although this goal is usually easily satisfied and is great insurance against problems like the one being described here. The problem in this example is that there is an inherent relationship between variances and positions. Even when the trading system is worthlessly random, different positions will tend to be associated with raw returns having different variances, with these associations *having nothing to do with intelligence of the system*. They are inherent in the design of the model. The implication of this association is that randomization will produce numerous unlikely pairings. This is a direct violation of the

Monte-Carlo assumption that in real life all possible pairings are equally likely under the null hypothesis.

There is a crude but often effective way to get around the problem just described if you must do so. It does bear repeating that the best approach is to break down extended positions into strings of individual (probably daily) positions and returns. But if for some reason we cannot do this, we should not simply apply Monte-Carlo randomization to the positions and net raw returns of the candidate system. The impact of unequal variances as just described could be severe. Instead, test only the sign of each obtained return, rather than its magnitude. In other words, count as a success every pairing that produces a positive return, and count as a failure every pairing that produces a negative return, regardless of the magnitude of the return. The total score of any system, candidate or randomized, is the number of successes minus the number of failures.

This is obviously not a perfect solution. A candidate system that has a large number of small wins and a small number of enormous losses, resulting in a net loss, will nevertheless be endorsed by this algorithm. Thus, systems that are inherently unbalanced need not apply. Nevertheless, beggars cannot be choosers. If the system being evaluated has no *a priori* imbalance, this method may perform well. The examples shown later will demonstrate the surprising power of testing nothing more than the ability of a trading system to match long positions with upward market moves and short positions with downward market moves.

Another situation in which the experimental design may inadvertently produce position/return pairings that are not well simulated by Monte-Carlo permutation is when the exit strategy involves unbalanced market moves. For example, suppose our trading system involves both long and short positions, and we decide that an open position will be closed if the market either rises four points or drops two points from when the trade was opened. Then in real life, long positions will always be associated with either a four-point win or a two-point loss, while the opposite will be true for short positions. But Monte-Carlo randomization of the pairings will associate either size win or loss with either position, and thus produce a null distribution that does not reflect reality. Note that the problem is *not* that the wins and losses are unbalanced. This is common and legitimate. The problem is that the trading system *by design* limits pairing of positions with returns in a way that is not reflected by random perturbation. The key question is this: If the trading system is truly worthless, are all possible pairing of returns with positions equally likely in real life? The answer is clearly no in this example.

Code for Monte-Carlo Randomization

At the center of an efficient Monte-Carlo randomization test is an algorithm that can quickly shuffle an array in such a way that every possible permutation is equally likely. The following algorithm fits the bill. Let the array be $X_1, X_2, ..., X_n$. Then:

1) Set $i=n$.
2) Randomly choose j from 1, ..., i with all choices being equally likely.
3) Exchange X_i with X_j.
4) Set $i=i-1$.
5) If $i>1$ go to Step 2.

The prior section discussed the possibility of scoring systems based on only the sign of the obtained return, ignoring the magnitude of the returns. This test might loosely be termed *nonparametric* because it is fairly free from the influence of inherent relationships between distribution parameters and positions, although it is certainly not completely free from these influences. This sign-only test is potentially useful when such damaging relationships are possible. Unfortunately, the randomized null distribution for this test can be problematic when you do not have a large number of position changes. Only three obtained returns are possible: success (usually coded as +1), failure (usually coded as –1) and zero. The result is that the null distribution is plagued with many ties, a fact which artificially inflates the area of the right tail (as well as the left tail, although we don't care about that one). Computed significance levels are overly conservative. A tempting but dangerous fix is to add a tiny random number to the score obtained by the candidate system and each randomized system. These random numbers should be small enough that they have no impact other than breaking ties. This technique converts a discrete null distribution with a large number of ties to a continuous distribution with no ties. The resulting test does not suffer from tie-induced conservative bias, and hence it rejects a true null hypothesis with the expected probability. Moreover, the apparent power of the test is increased. Nonetheless, strictly speaking, this modification is incorrect in that it does alter the null distribution from its theoretically correct shape. I prefer to be correct and conservative. Code for Monte-Carlo randomization is as follows:

```
double monte (
   int n ,                    // Number of trading opportunities
   double *raw ,              // Raw return of each
   int *pos ,                 // Position of each; 1=long; -1=short; 0=neutral
   int nreps ,                // Number of replications, generally at least 1000
   double *nonpar ,           // If not NULL, only signs used for nonparametric test
   int *work                  // Work vector n long avoids shuffling input data
   )
```

```
{
   int i, irep, k1, k2, count, nonpar_count, temp ;
   double cand_return, nonpar_cand_return, trial_return, prod ;

   memcpy ( work , pos , n * sizeof(int) ) ;  // Preserve input positions

/*
   Compute the return of the candidate model
   If requested, do the same for the nonparametric version
*/

   cand_return = 0.0 ;
   for (i=0 ; i<n ; i++)
      cand_return += pos[i] * raw[i] ;

   if (nonpar != NULL) {                    // Do the same using only signs if requested
      nonpar_cand_return = 0.0 ;
      for (i=0 ; i<n ; i++) {
         prod = pos[i] * raw[i] ;
         if (prod > 0.0)
            nonpar_cand_return += 1.0 ;
         else if (prod < 0.0)
            nonpar_cand_return -= 1.0 ;
         }
      }

/*
   Do the Monte-Carlo replications
*/
   count = 0 ;                  // Counts how many at least as good as candidate
   nonpar_count = 0 ;           // Ditto for ignoring signs (if requested)


   for (irep=0 ; irep<nreps ; irep++) {

      k1 = n ;                  // Shuffle the positions, which are in 'work'
      while (k1 > 1) {          // While at least 2 left to shuffle
         k2 = (int) (unifrand () * k1) ; // Pick an int from 0 through k1-1
         if (k2 >= k1)          // Should never happen as long as unifrand()<1
            k2 = k1 - 1 ;       // But this is cheap insurance against disaster
         temp = work[--k1] ;    // Count down k1 and swap k1, k2 entries
         work[k1] = work[k2] ;
         work[k2] = temp ;
         }   // Shuffling is complete when this loop exits
      trial_return = 0.0 ;
      for (i=0 ; i<n ; i++)        // Compute return for this randomly shuffled system
         trial_return += work[i] * raw[i] ;

      if (trial_return >= cand_return) // If this random system beat candidate
         ++count ;                 // Count it

      if (nonpar != NULL) {    // Do the same using only signs if requested
         trial_return = 0.0 ;
         for (i=0 ; i<n ; i++) {   // Compute return for this randomly shuffled system
            prod = work[i] * raw[i] ;
            if (prod > 0.0)
               trial_return += 1.0 ;
            else if (prod < 0.0)
               trial_return -= 1.0 ;
            }
         if (trial_return >= nonpar_cand_return) // If this random system beat candidate
            ++nonpar_count ;          // Count it
         }
      }

   if (nonpar != NULL)
      *nonpar = (double) nonpar_count / (double) nreps ;

   return (double) count / (double) nreps ;
}
```

The calling parameter list includes *nonpar, which should be input as NULL if only the ordinary test is to be performed. If a pointer to a real variable is input, the so-called nonparametric test, in which magnitudes are ignored, will also be performed, with the resulting p-value returned to this variable. Once again, note that this test is not strictly nonparametric, because the distribution of the raw returns can have some impact in some trading scenarios. This will be discussed more in the next section.

A work vector is used to preserve the original order of the input position vector. If preservation is not required, this vector can be eliminated.

The first step is to compute the return of the candidate system, as well as the sign-only return if requested. Then the Monte-Carlo loop repeatedly shuffles the position vector and computes the return for each trial. Every time a trial return equals or exceeds the candidate's return, a counter is incremented. The computed p-value, which is the area to the right of the candidate's return, is obtained by dividing the count by the number of trial replications. Observe that the shuffling algorithm stated earlier assumes an index origin of one, but C++ uses an origin of zero. This necessitates some minor changes in indexing.

Tests With Simulated Returns

I wrote a program that compares Monte-Carlo randomization with the percentile and basic bootstrap algorithms using a variety of raw return distributions. Positions are serially correlated by design, with positions (long, short, or neutral) tending to be grouped into clumps rather than being randomly distributed. Tests are done two ways, once using individual raw returns, which is the preferred method for Monte-Carlo randomization when possible, and once using the net return for each extended trade. In other words, for this method the sum of raw returns is cumulated for each group of the same position, and the tests are based on the sums only. Real-world trading scenarios may sometimes require this modification. Finally, this summation method is tested using the signs-only randomization test described earlier.

Some tables of results now follow. Each test is performed for three degrees of intelligence in the candidate model, and the percent rejection at the 0.05 level is reported for each of these three powers. For worthless candidates we would expect that the rejection rate at the 0.05 level would be about five percent. Modestly powerful candidates should enjoy a higher rejection rate, and strong candidates should reject at an even higher rate. If worthless candidates are rejected at a rate of less than five percent, the test is conservative, which is annoying but not a serious problem as long as the rejection rate is close to five percent. If the rejection rate for worthless candidates is much higher than the expected five percent, the test is anti-conservative, which is a serious problem in that the test will erroneously reject the null hypothesis too often. For intelligent models, higher rejection rates are good because we want to reject the null hypothesis when the candidate has intelligence.

Each table contains seven rows and three columns. The left column is the rejection rate for worthless models, the center column is the rate for modestly intelligent models, and the right column is that for powerful models. The first row is the set of tests using Monte-Carlo randomization of the individual returns. The next two rows are the percentile and basic bootstrap tests of the same data. The next three rows are the Monte-Carlo, percentile, and basic tests using the grouped returns. The last row is the Monte-Carlo sign-only test of the grouped returns.

Table 1 presents the results for serially independent raw returns that have a normal distribution. There were 500 individual returns, of which approximately 156 were long, 78 were short, and 266 were neutral. (Actually, the number of long, short, and neutral positions is random. The cited numbers are the means.) The length of each open position follows a geometric distribution with a mean length of eight.

This table shows that when the individual obtained returns are used, Monte-Carlo randomization performs excellently. Its 0.05 rejection rate is exactly what would be expected, and its reject rate increases rapidly as the true power of the trading system increases. Both bootstrap algorithms also perform well, though with a slightly conservative bias.

When the trades are grouped, which violates the identical distribution requirement of both randomization and the bootstrap, all three tests become moderately anti-conservative, with the basic bootstrap performing best by a small margin.

What is very interesting is that the sign-only modification of the grouped data performs fabulously. Its reject rate under the null hypothesis is substantially less than what would be expected, which ordinarily would be a warning flag that the test lacks power. But its reject rate when the null hypothesis is false is actually better than the three full-information alternatives! This is probably because of the way effective models are defined in the test program. Effective models have a higher than random probability of correctly matching positions with raw returns, which is precisely what the sign-only test measures.

Test	Worthless	Modest	Powerful
MC Indiv.	5.00	19.43	47.40
PCtile Indiv.	4.69	18.84	46.55
Basic Indiv.	4.72	18.85	46.57
MC grouped	5.71	21.14	48.83
PCtile grouped	6.09	22.86	53.35
Basic grouped	5.54	22.19	53.30
Sign-only	3.36	22.93	63.21

Table 1 Raw returns with independent normal distribution.

Table 2 demonstrates results when the raw returns have a strong positive skew. The bootstrap tests become conservative, with Monte-Carlo randomization being unaffected. Interestingly, the percentile bootstrap, despite having a far lower reject rate when the null hypothesis is true, has the highest reject rate when the null hypothesis is false. This is fantastic!

Test	Worthless	Modest	Powerful
MC Indiv.	5.03	12.26	25.27
PCtile Indiv.	1.49	20.14	47.98
Basic Indiv.	0.31	9.40	28.47
MC grouped	4.37	12.32	26.19
PCtile grouped	2.35	18.28	45.87
Basic grouped	0.61	9.07	29.81
Sign-only	3.53	13.73	36.59

Table 2 Raw returns with right-skewed distribution.

Unfortunately, as Table 3 shows, the situation reverses dramatically when the raw returns have a left skew. Both bootstrap methods become so anti-conservative that the tests are worthless. Again, Monte-Carlo randomization is not affected.

Test	Worthless	Modest	Powerful
MC Indiv.	5.09	13.12	27.01
PCtile Indiv.	21.01	34.70	53.21
Basic Indiv.	13.81	23.91	38.54
MC grouped	5.52	14.15	28.58
PCtile grouped	14.36	31.80	52.99
Basic grouped	7.73	22.06	40.45
Sign-only	2.62	13.21	37.78

Table 3 Raw returns with left-skewed distribution.

Tables 4 and 5 show the results when the raw returns have two different forms of unusual symmetric distributions. The former has a heavy-tailed unimodal distribution, and the latter has a relatively light-tailed bimodal distribution, which would be encountered in practice when the trading system is designed to have preordained wins and losses. There is nothing unusual to note here. All of the tests behave roughly as expected and enjoy decent and similar power. When individual returns are used, Monte-Carlo randomization exhibits a reject rate that is closest to what theory would predict. And once again, the sign-only test for grouped data performs extremely well.

Test	Worthless	Modest	Powerful
MC Indiv.	5.02	16.30	38.18
PCtile Indiv.	5.48	17.35	40.58
Basic Indiv.	4.31	15.01	37.37
MC grouped	5.66	18.09	40.05
PCtile grouped	6.70	20.94	46.28
Basic grouped	5.15	18.12	43.61
Sign-only	3.50	22.30	61.69

Table 4 Raw returns with heavy-tailed symmetric distribution.

Test	Worthless	Modest	Powerful
MC Indiv.	5.02	20.03	49.02
PCtile Indiv.	4.76	19.22	47.91
Basic Indiv.	4.83	19.43	48.29
MC grouped	5.70	21.91	50.22
PCtile grouped	6.17	23.46	54.63
Basic grouped	5.68	22.89	54.74
Sign-only	3.43	23.03	62.76

Table 5 Raw returns with bimodal symmetric distribution.

Table 6 illustrates results when the raw returns have a serial correlation of 0.1, which is far larger than that exhibited by any commodity market I've seen. The tests become anti-conservative, which as has already been discussed may be viewed as a manifestation of the fact that serially correlated positions are the best response to a serially correlated market.

Test	Worthless	Modest	Powerful
MC Indiv.	6.57	23.35	52.67
PCtile Indiv.	6.21	22.58	51.87
Basic Indiv.	6.25	22.55	51.91
MC grouped	5.61	21.23	48.14
PCtile grouped	6.04	22.81	52.93
Basic grouped	5.36	21.92	52.69
Sign-only	3.31	23.02	62.80

Table 6 Raw returns with correlated normal distribution.

Finally, Table 7 demonstrates what happens when there are only 40 individual trades in an independent normally distributed market. As often happens, the bootstrap tests become moderately anti-conservative. Monte-Carlo randomization of individual trades manages to stay right on target with its null rejection rate.

Test	Worthless	Modest	Powerful
MC Indiv.	5.00	15.04	33.62
PCtile Indiv.	6.49	18.82	40.35
Basic Indiv.	6.49	19.25	41.60
MC grouped	3.19	10.78	25.86
PCtile grouped	6.49	18.84	40.42
Basic grouped	6.49	19.23	41.59
Sign-only	2.68	12.06	34.74

Table 7 Raw returns with independent normal distribution, n=40.

If there is one thing to be learned from the preceding tables, it is that the Monte-Carlo permutation test of individual returns is stable and reliable compared to bootstrap tests. Table 3 certainly makes this clear! Naturally, the user may not always have a choice. The fact that Monte-Carlo randomization requires more information than the bootstrap may leave the user with only the bootstrap option. But the moral of the story is that, whenever possible, a trading strategy should be designed with this in mind. By enabling use of all individual returns, obtained and hypothetical, the user can employ Monte-Carlo randomization tests, which have many advantages over the bootstrap.

Testing the Best of Many Models

Very often the experimenter will build many models, which may be variations on one theme, or which may represent a variety of ideas. The best model from among a collection of candidates might have obtained that exalted position because of its quality, or because of its good luck. Most often both are involved. It would be terribly unfair to evaluate the quality of the best model by testing it alone. Any good luck that propelled it to the top, commonly called *selection bias*, will be misinterpreted as intelligence. This, of course, has been the downfall of many an aspiring market trader. Fortunately, the Monte-Carlo randomization algorithm just described can easily be modified to take selection bias into account. This modified algorithm computes the probability that the best model from among the collection of candidates could have done as well as it did if *all* of the candidates represent random pairings of positions with raw returns. Note that the null hypothesis tested is *not* that the best performer is worthless. Rather, the null hypothesis is that *all* of the competitors are worthless. In practical applications this distinction is of little or no consequence. But it is a crucial theoretical distinction.

The modification is effected by repeatedly permuting all of the position vectors simultaneously, and for each permutation finding the maximum total return from among the competing rules. The collection of these returns obtained from a large number of permutations defines the null hypothesis distribution.

It is crucial that the competing models' position vectors be permuted simultaneously, rather than permuting each independently. (Equivalently, and perhaps more simply, the market return vector can be permuted.) This is because the competing rules may (and probably will) be correlated. This correlation structure must be maintained. As an extreme example, suppose that all of the competing models are identical; their position vectors are the same. They will all obviously have the same total return, so finding the maximum return among them would be redundant. This situation should obviously reduce to the single-model problem already described. If the position vectors of the competing models were perturbed independently, the resulting null distribution would be quite different from that obtained by treating the problem as if it were a single model, which it really is.

Even though the best-of modification is mostly straightforward, examination of the code clarifies any possible misunderstandings. Here is the routine:

```
double monte_best (
   int n ,                   // Number of trading opportunities for each system
   int nbest ,               // Number of competing systems
   double *raw ,             // Raw return of each system; nbest sets of n
   int *pos ,                // Position of each; 1=long; -1=short; 0=neutral; nbest sets of n
   int normalize ,           // If nonzero, normalize positions for time neutral
   int nreps ,               // Number of replications, generally at least 1000
   double *nonpar ,          // If not NULL, only signs used for nonparametric test
   int *shuffle              // Work vector n long holds shuffle indices
   )
{
   int i, ibest, irep, k1, k2, count, nonpar_count, temp, *posptr, npos ;
   double cand_return, nonpar_cand_return, trial_return, shuffle_return ;
   double factor, prod, *rawptr ;

   factor = sqrt ( n ) ;     // Produces sum when always in market

/*
   Compute the return of the candidate models and keep track of best
   If requested, do the same for the nonparametric version
*/

   posptr = pos ;            // Points to first position of each candidate system
   rawptr = raw ;            // Ditto for raw returns
```

```
for (ibest=0 ; ibest<nbest ; ibest++) {
   trial_return = 0.0 ;         // Will sum total return
   npos = 0 ;                   // Will count times in market
   for (i=0 ; i<n ; i++) {      // Cumulate return for candidate system 'ibest'
      if (posptr[i])            // If we are in the market at this time
         ++npos ;               // Count time in market
      trial_return += posptr[i] * rawptr[i] ; // Sum total return
      }

   if (normalize)
      trial_return *= factor / sqrt ( npos ) ;

   if ((ibest == 0)  ||  (trial_return > cand_return))
      cand_return = trial_return ; // Keep track of the best of the candidates
   posptr += n ;   // Advance to the next candidate system
   rawptr += n ;
   }

if (nonpar != NULL) {           // Do the same using only signs if requested
   posptr = pos ;               // Points to first position of each candidate system
   rawptr = raw ;               // Ditto for raw returns
   for (ibest=0 ; ibest<nbest ; ibest++) {
      trial_return = 0.0 ;
      npos = 0 ;
      for (i=0 ; i<n ; i++) {
         if (posptr[i])         // If we are in the market at this time
            ++npos ;            // Count time in market
         prod = posptr[i] * rawptr[i] ;
         if (prod > 0.0)
            trial_return += 1.0 ;
         else if (prod < 0.0)
            trial_return -= 1.0 ;
         }

      if (normalize)
         trial_return *= factor / sqrt ( npos ) ;

      if ((ibest == 0)  ||  (trial_return > nonpar_cand_return))
         nonpar_cand_return = trial_return ; // Keep track of the best of the candidates
      posptr += n ;             // Advance to the next candidate system
      rawptr += n ;
      }
   }
```

```
/*
   Do the Monte-Carlo replications
*/
   count = 0 ;                    // Counts how many at least as good as candidate
   nonpar_count = 0 ;             // Ditto for ignoring signs (if requested)

   for (irep=0 ; irep<nreps ; irep++) {

      for (i=0 ; i<n ; i++)
         shuffle[i] = i ;         // These will index the position vectors
      k1 = n ;                    // Shuffle the position indices
      while (k1 > 1) {            // While at least 2 left to shuffle
         k2 = (int) (unifrand () * k1) ; // Pick an int from 0 through k1-1
         if (k2 >= k1)            // Should never happen as long as unifrand()<1
            k2 = k1 - 1 ;         // But this is cheap insurance against disaster
         temp = shuffle[--k1] ; // Count down k1 and swap k1, k2 entries
         shuffle[k1] = shuffle[k2] ;
         shuffle[k2] = temp ;
         }                        // Shuffling is complete when this loop exits

      posptr = pos ;              // Points to first position of each candidate system
      rawptr = raw ;              // Ditto for raw returns
      for (ibest=0 ; ibest<nbest ; ibest++) { // Find the best of the shuffled candidates

         trial_return = 0.0 ;     // Will sum total return
         npos = 0 ;               // Will count times in market
         for (i=0 ; i<n ; i++) {  // Cumulate return for shuffled system
            if (posptr[shuffle[i]]) // If we are in the market at this time
               ++npos ;           // Count time in market
            trial_return += posptr[shuffle[i]] * rawptr[i] ; // Sum total return
            }

         if (normalize)
            trial_return *= factor / sqrt ( npos ) ;

         if ((ibest == 0)  ||  (trial_return > shuffle_return))
            shuffle_return = trial_return ; // Keep track of the best of the shuffled candidates

         posptr += n ;            // Advance to the next candidate system
         rawptr += n ;
         }

      if (shuffle_return >= cand_return) // If this random system beat candidate
         ++count ;                // Count it
      if (nonpar != NULL) {    // Do the same using only signs if requested
         posptr = pos ;           // Points to first position of each candidate system
         rawptr = raw ;           // Ditto for raw returns
         for (ibest=0 ; ibest<nbest ; ibest++) { // Find the best of the shuffled candidates
            trial_return = 0.0 ;
            npos = 0 ;
            for (i=0 ; i<n ; i++) { // Compute return for this randomly shuffled system
               if (posptr[shuffle[i]]) // If we are in the market at this time
                  ++npos ;          // Count time in market
               prod = posptr[shuffle[i]] * rawptr[i] ;

               if (prod > 0.0)
                  trial_return += 1.0 ;
               else if (prod < 0.0)
                  trial_return -= 1.0 ;
               }

            if (normalize)
               trial_return *= factor / sqrt ( npos ) ;
            if ((ibest == 0)  ||  (trial_return > shuffle_return))
               shuffle_return = trial_return ; // Keep track of the best of the shuffled candidates
            posptr += n ;          // Advance to the next candidate system
            rawptr += n ;
            }
         if (shuffle_return >= nonpar_cand_return) // If this random system beat candidate
            ++nonpar_count ;   // Count it
         } // If doing nonpar
      } // For all reps

   if (nonpar != NULL)
      *nonpar = (double) nonpar_count / (double) nreps ;

   return (double) count / (double) nreps ;
}
```

Most of the operations in the subroutine were explained earlier for the single-candidate version, so they will be ignored here. One difference involves the data structure for the positions and raw returns. Each of these vectors contains nbest*n elements. The first n elements are the positions/raw returns for the first of the nbest candidates. The next n elements are for the second candidate model, et cetera. The pointers posptr and rawptr walk through the candidates. For each value of ibest, these two variables will point to the first element of the corresponding candidate model.

The easiest way to simultaneously shuffle all position vectors is to create an index vector shuffle that contains the integers zero through n–1 and randomly shuffle this vector. By using the indices in this vector to pair positions with raw returns, all positions are shuffled identically.

Astute readers might be wondering why this subroutine employs a separate raw return vector for each candidate model. It would seem that competing models would be differentiated only by their position vectors. They would all have the same raw return for each trade opportunity. The answer is that this is a general implementation that actually allows competing models to trade different markets if desired. For example, an aspiring trader might not be sure if he wants to trade pork bellies or crude oil. He's willing to be flexible and pick whichever provides the best historical returns. So he may try 20 candidate models to trade pork bellies and 15 candidates for crude oil, giving a total of 35 candidates. As long as the raw returns for both markets are commensurate (such as percent returns having about the same variance) and the trading opportunities are parallel (such as daylong exposure) it would be legitimate to mix markets this way. Readers who want to simplify the code to use a single return vector to trade a single market can easily modify the algorithm.

The algorithm includes optional normalization for the amount of time each candidate spends in the market. Motivation for why this is useful is presented on Page 35. Here we discuss how it is done and what some its implications might be.

Normalization is appropriate only if the candidate rules spend significantly different times in the market. In other words, if the number of zeros in the position vector is about the same for all candidates, there is no need to normalize. The effect of normalization is to increase the relative impact of candidates that spend less time in the market. When finding the best performer, instead of comparing their total returns, we compare the scaled returns.

Some might argue that this results in comparing apples to oranges, for instead of judging systems by their total or mean return, we are comparing differently scaled returns. But this is not really a problem. It is more like comparing Red Delicious apples to Yellow Delicious apples. This scaling simply amounts to trading a larger number of contracts for thinly traded systems, an eminently justifiable action.

The scaling factor is the square root of the number of trading opportunities in which the rule is in the market. A rule that is always in the market will not be rescaled. Its score will be its total return. A rule that is in the market only one-quarter of the time will have its total return doubled.

This factor is chosen because it has the effect of equalizing the variance of the rules, the virtue of which is discussed on Page 35. In practical terms, this means that the rules would all have about the same volatility. It seems natural to compare the total returns of rules that have roughly equal volatility. Readers familiar with linear algebra will also see a nice geometric interpretation of the scaling. The Euclidean lengths of the position vectors are made equal, so when the dot products of the positions with the raw returns are formed, equally scaled projections result.

Serial Correlation, Again

If this algorithm is applied to a market having significant serial correlation in both the raw returns and the position vectors, there will be a strong anti-conservative bias in the results, far more than was seen in the single-candidate case. Rules that are random except for their serial correlation will reject the null hypothesis much more often than expected. This issue was already discussed on Page 7, but it is so important, especially in a best-of scenario, that it bears repeating. With serial correlation in the market and positions, wins and losses will both be exaggerated compared to the fully random case because sometimes position clumps will be aligned advantageously, and sometimes disadvantageously, with periods of unusually large or small raw returns. In other words, candidate systems will tend to be either strong winners or strong losers, with middling performers de-emphasized. Looked at another way, if you want a winning system in a serially correlated market, you would be well advised to employ systems with serially correlated positions. That's where the best winners reside (along with the worst losers!).

This is why when both the raw returns and the positions of candidate systems are serially correlated, the best from among a collection of candidates will tend to appear with a frequency exceeding the computed significance level in a Monte-Carlo permutation test. This test is computing the probability in a universe in which *all* pairings are equally likely, not a universe in which correlated pairings are more likely than uncorrelated pairings. If numerous candidates are compared, it is quite likely that at least one of them will get lucky and obtain one or more favorable alignments of correlated positions with correlated raw returns.

So we are left with a vital philosophic question: Is this good, or is it bad? It is bad in the sense that the supposedly good candidates actually chose their position based on the alignment of celestial bodies, not an effective prediction model. With this in mind, we should not be happy that candidates are being rated as good more often than we should expect. On the other hand, serially correlated candidates apparently were deliberately designed to clump their positions. *This is precisely what we want in an environment of serially correlated raw returns.* So in a very real sense, the candidates *are* showing some intelligence, at least as far as the historical sample goes, and the Monte-Carlo test is discovering this intelligence. Naturally, if the best model truly is random other than being serially dependent, this randomness cannot be expected to continue to align well with market positions. Just as in this particular historical sample it performed unusually well, at some time in the future it will perform unusually badly. But the point is that *if the best model chose serially correlated positions on its own, not as a result of forces imposed by the designer*, chances are good that they resulted from the serial correlation in the market, and hence are intelligent. Of course, if the serial correlation in positions was mostly imposed by design, and the market has strong serial correlation, this argument goes out the window, and best-of selection will have a very strong tendency to pick a lucky worthless model. These are the facts. Form your own opinion relative to your application.

Last, recall that on Page 6 it was pointed out that by centering the raw market returns, we could avoid confounding true predictive quality with prejudice induced by the interaction of market bias with long/short bias. The point was made that such centering is not needed for the Monte-Carlo permutation test because it happens implicitly, but that it had such general utility for other tests that it should perhaps be done habitually. When using Monte-Carlo permutation to test the best of many models, centering does have an impact, and it almost certainly should be used.

The impact occurs in the generation of the null hypothesis return distribution. When the best randomized model is chosen for each iteration, models whose long/short imbalance matches market bias will be favored. The result is that the null distribution is shifted to the right, making the test more conservative. It may be that such favoritism is desired. But it is my opinion that centering the raw market returns is a good practice for Monte-Carlo permutation tests of multiple models, especially when there is strong market bias and a variety of long/short ratios among the competing models.

Permutation Tests During Training

There are many common applications in which the Monte-Carlo permutation test may be used in a best-of-multiple-models situation. Among these (though certainly not limited to these) are the following:

1. The developer may hypothesize a number of models, perhaps intelligently or perhaps by throwing spaghetti against the wall. For example, Model 1 may buy when the market rises ten percent on increasing volume. Model 2 may be the same except requiring a five percent rise. Model 3 may ignore volume, and so forth. Each of these competing models is tested on a dataset and the best is chosen for actual trading. We wish to compute the probability that a model this good could have arisen from pure luck.

2. We may hypothesize a model but suspect that it performs differently in different markets. Thus, we want to test it in a variety of markets, with the intention of trading only the best market.

3. We may design several complex models that require parameter optimization. For example, we may have a moving-average-crossover model that trades only on increasing volume, and another that trades only on decreasing volume, and a third that ignores volume. We need to optimize the MA lengths, as well as a possible volume threshold. After training these competing models on a dataset, they are all tested on virgin data, and the best out-of-sample performer is selected for subsequent trading.

There is, however, an application for which the Monte-Carlo permutation test described earlier is usually not appropriate, although there are several modifications that can enable its use. Most training algorithms test the performance obtained from a large collection of trial parameter values and ultimately choose the parameters that provide the maximum performance. It is often not legitimate to treat the position vectors corresponding to the trial parameters as competitors and apply the preceding algorithm to compute a probability for the final best performers.

There is one situation in which this is permissible. If we are just doing a random or grid search of trial values, there is no problem. For example, suppose we are optimizing two parameters. We may define ten trial values for each of them and then test the 100 possibilities, choosing the best. There is no problem here. Or we may randomly generate trial parameter values and choose the best performer. Again, no problem.

The problem arises when some of the supposed competitors are chosen in a way that is dependent on the performance of other competitors, a practice that leaks knowledge of the market into the new competing model's design. This would be true for hill-climbing algorithms, genetic optimization, and so forth. We now explore the source of this problem.

Suppose we have three designers of trading systems: John, Mark, and Mary. They are asked by their employer to each think up a good trading algorithm. He will then give each of them the same dataset and ask them to test their system. The employer will choose the system that performed the best. A straightforward application of the Monte-Carlo permutation test will tell him the probability that the best performer's results could have arisen from luck.

However, suppose John decides to cheat. He spies on his two competitors and waits until they have backtested their systems. After surreptitiously examining their results, he tweaks his own idea so that it is similar to that of whichever competitor had the better results, hoping that the melding of his idea with an idea that he now knows to be good will result in a superior system.

Consider how a straightforward application of the permutation test will generate the supposed null distribution. The position vectors of the three competitors will be frozen, which is necessary for capturing any correlation structure in their methods. The market returns will be permuted several thousand times. For each permutation, the returns of the three competitors will be computed and the maximum found. This set of maximum values defines the null distribution.

Unfortunately, a great many of these values will be illegal because they could not have happened in real life. When the true test is done, John will look at the returns for Mark and Mary, and design his system in accordance with the winner. Suppose Mary wins. Then John's system will resemble Mary's. But when the permutations are done, many of them will result in Mark beating Mary. In these cases, if we are to generate the null distribution in accord with real life, John's positions should resemble Mark's, not Mary's. But John and Mary are locked together for the test.

Recall that the fundamental assumption of the permutation test is that all possible permutations must be equally likely in real life. In this case, many of the permutations could never happen in real life! This is a deadly violation of the assumption.

One moral of the story is that generation of the null distribution must be in accord with real life. If John cheats by looking at Mary and Mark's results, then the null distribution process must do the same. For each permutation, it should first compute the returns for Mark and Mary, then generate John's system according to whatever rules he applied in real life. Obviously, this would be an impossible task in any but the most trivial situations. We need a different approach.

Before proceeding, it is useful to examine the problem from a different angle. Recall that the null hypothesis for the test is that *all* of the models are worthless. But in the cheating example just given, this can never be the case. Suppose Mark and Mary both have truly worthless systems. In other words, their expectation over the theoretical population of all possible datasets is no better than a random system. Since John designs his system to resemble the better of them, his expectation will be better than a random system. In other words, the null hypothesis will always be violated in this situation!

It should be clear then, that this problem is not unique to the permutation test. It also applies to the bootstrap test described in the next section, as well as any other best-of test that I can imagine.

So, what do we do if we want to use a Monte-Carlo permutation test on the results of a training run? There are several possibilities. We'll explore a few of them.

The simplest, though often impractical method involves slightly changing the null hypothesis. First, difference the series. Do each of the several thousand permutations on the differences, and them sum the differences to produce a new series whose basic statistical properties should be similar to those of the original series. For each of these series, apply the training algorithm and record the performance of the trained model. This set of performances defines the null distribution against which the actual performance is compared. In this case, we are testing the null hypothesis that the sequence of daily changes provides no predictive information that the model can usefully detect. In many cases this is a perfectly reasonable null hypothesis.

This method often has a serious practical drawback. The training algorithm must be run several thousand times. If a training run requires considerable time, repeating it this many times may be impractical.

A closely related and often superior method may be used, especially if the application involves laborious precomputation of a set of predictor variables. This will often be the case if we are computing a vast number of candidate predictors and using a selection procedure to choose a small subset of them for use by the model. In this case, we compute all predictors and market returns and arrange the data in a matrix. Each row of the matrix corresponds to a single observation period. The columns are the predictors, except for the last column, which is the market return. Then you would keep the predictors fixed and permute the column of market returns several thousand times. For each permutation, run the training algorithm. This tests the null hypothesis that the model is unable to find any useful predictive information in the predictors.

The principal advantage of this method over permuting differences is that the predictors remain fixed. In many cases this is a more realistic approach than generating a new set of predictors for each permutation. Unfortunately, this method also suffers from the problem of requiring several thousand runs of the training algorithm.

My own favorite permutation test when training a complex model on a single dataset is to use either walkforward testing or cross validation to generate a set of out-of-sample positions, and then apply the ordinary permutation test to these results. In other words, hold out a small block of data and train the model on the remainder of the dataset. Apply the trained model to the hold-out set and record the position vector. Then replace the hold-out set and remove a different block. Repeat this test until all cases have been held out exactly once. (This is cross validation, my preference. You may wish to walk forward instead.) After the complete set of positions is found, apply the single-sample permutation test in order to compute a probability value for the return. This method requires that the training procedure be run only once for each hold-out block. Of course, if you are simultaneously training several competing models or applying a model to several markets, you will need to use the best-of version of the test.

Astute readers may notice that this method is not actually testing a single model, a model produced by a training algorithm or model generator. Rather, it is simultaneously testing a group of models, almost certainly a different model for each of the dataset splits. So what are we really testing? We are testing the model generator itself, whether it be something as simple as optimizing parameters in a fixed design, or something as complex as an evolutionary algorithm that builds a model. If we find that the statistical significance of the complete out-of-sample set is good, we can be confident that the procedure we used to generate the individual models is effective. Thus, when we conclude the operation by using our model factory on the entire dataset, we can be confident that the model it produces will be effective. There is usually no way to test the efficacy of this final model unless we can procure a virgin dataset. But at least we can trust the model if we believe that the procedure that produced it is trustworthy.

In summary, the null hypothesis of the multiple-model permutation test is that *all* competing models are worthless in the sense that their returns are no better than what could be obtained by random guessing. But when some of the competitors are designed based on leaked information about the market returns, which happens when the design of the models depends on the performance of other competitors, the null hypothesis is violated, rendering the test useless. The test must be redesigned in such a way that no market information goes into the design of any of the competitors.

Bootstrapping the Best of Many Models

Just as was the case for Monte-Carlo randomization, there is a simple extension of the bootstrap to handle the case of many competing models. We begin by reviewing the single-model case. Let θ be the unknown true mean return of the single candidate system. Our null hypothesis is that $\theta=0$; the system is worthless. The observed return of the candidate system is $\hat{\theta}$, and we wish to know whether it is so large as to be unlikely when the null hypothesis is true. The observed return is subject to random sampling error, $\hat{\theta}-\theta$. If we knew the true distribution of this quantity, we could compute the area of its tail to the right of the observed $\hat{\theta}$ in order to find the basic-method probability of having obtained a return this good when the true mean return is zero. Naturally we do not know this distribution. But we can estimate it by drawing a large number of bootstrap samples from the original set of trade results and computing the mean $\hat{\theta}^\star$ of each bootstrap sample. The distribution of $\hat{\theta}^\star-\hat{\theta}$ should resemble the distribution of the error, $\hat{\theta}-\theta$. (We hope!)

Now suppose that instead of having just one candidate system, we have K of them in competition. Let the mean return of system k be $\hat{\theta}_k$. In order for the algorithm to be valid, each of these competing systems must have the same number of individual returns, and the returns must be parallel in time. In other words, the first individual return in all of the systems must refer to the same time period, as must the second, et cetera. This is easily satisfied if, for example, the individual returns are daily. It is perfectly legitimate to insert a few returns of zero if this is needed to achieve parallel structure, although too many zeros can distort the test.

In order to preserve any correlation among the systems, each bootstrap sample must encompass each competing system identically. So, for example, in a particular bootstrap sample we might ignore the first and second returns in each system, include the third return four times for all systems, et cetera.

In the single-system case we computed the bootstrap distribution of $\hat{\theta}^\star-\hat{\theta}$. In the best-of case we compute $\hat{\theta}_k^\star-\hat{\theta}_k$ separately for each system and find the maximum of these quantities across all K systems. The distribution of this maximum is the null hypothesis distribution. The area in its tail to the right of the best observed mean return is the basic-method probability of having obtained a best mean return this good or better from a set of candidate rules all of which are worthless.

Remember that Monte-Carlo randomization and the bootstrap are actually testing different null hypotheses, although in practice the hypotheses will usually be similar. The Monte-Carlo method tests the null hypothesis that the pairing of long/short/neutral positions with raw returns is random as opposed to intelligent. The bootstrap tests the null hypothesis that the true (expected in the future) returns of the candidate systems are zero. In most cases the difference is more theoretical than practical, but it should be kept in mind.

The advantages and disadvantages of the bootstrap approach to best-of evaluation compared to Monte-Carlo randomization are similar to those in the single-system case. The big advantage of the bootstrap method is that it requires knowledge of obtained returns only for periods in which trades actually took place. Monte-Carlo randomization requires the additional knowledge of the raw market returns and positions for all trading opportunities, ideally including even those in which no trade took place. This information concerning positions and hypothetical returns for trades that never happened may not be available in all trading scenarios, leaving the bootstrap as the only alternative.

The main problem with the bootstrap is its critical assumption that the distribution of $\hat{\theta}^{\star}-\hat{\theta}$ is an effective proxy for the distribution of the sampling error, $\hat{\theta}-\theta$. As is well known, this is an often dubious assumption. Worse, the best-of modification of the bootstrap does not support any apparent way that I can see of using the generally superior percentile or BC_a methods of hypothesis testing. The basic method seems to be the only choice. Nonetheless, the bootstrap does have some strong asymptotic convergence properties, and it is almost always easier to apply than Monte-Carlo randomization due to the fact that it has no need of positions or hypothetical returns. For an extremely detailed examination of the bootstrap algorithm just described, including asymptotic convergence theorems and extensions to other performance measures, see Halbert White's assorted papers on the subject, most of which are readily available on the Internet.

As a final note, readers should be warned that this algorithm has been patented by Dr. Halbert White, who also sells a software implementation under the trademarked name Reality Check. Experimenters who wish to use this bootstrap test for evaluating the best of competing models should contact Dr. White about obtaining a license or purchasing his software.

Spoiler Models

When testing one model for quality, the official null hypothesis is that the model is worthless or worse, and the alternative is that the model has predictive ability. In the case of the Monte-Carlo permutation test, worthless means that the pairings of positions with raw market returns is random, and for the bootstrap worthless means that the expected mean return is zero. There is the possibility that a model is actually worse than worthless. The pairings that it produces may, on average, be nonrandom in a way that is inferior to truly random pairings. The expected mean return may be negative. Still, we treat the null hypothesis as if it is random. We are justified in doing this because randomness is the *least favorable hypothesis*. It should be obvious that when we compute the probability of an observed good model having attained its level of performance by pure luck under a hypothesis of randomness, the probability of such performance under a worse hypothesis will be even less than the computed value. This allows us to keep the null hypothesis simple.

But when we are testing the best performer from among a set of competitors, the situation is not so clear. It is certainly reasonable to assume a null hypothesis that all competitors are random. To do otherwise would force us to confront an infinitude of null hypotheses, and with them an infinitude of null distributions. But are we on solid ground in doing so? Actually, the ground here is a bit mushy. This section will explore the situation on a heuristic level. Strict mathematical rigor is beyond the scope of this text.

Before getting into the discussion, one other situation must be mentioned. The null hypothesis says nothing about the variance of the rule returns. Neither Monte-Carlo permutation nor the bootstrap of the best requires that all competitors have equal variance. Nonetheless, it is good to at least roughly equalize their variances. If different markets are used for different rules, the raw market returns should have nearly equal variance. If different rules spend different amounts of time in the market, it is best to normalize their returns according to the square root of their summed absolute positions, as discussed with the code earlier.

The reason is that we usually do not want high-variance rules to dominate the test. It is not illegal for this to happen. But it is almost never good. Suppose one competitor has a variance that is much greater than its peers. When the null hypothesis is generated, either by Monte-Carlo permutation or by bootstrapping, the right tail, which is the one that interests us, will be defined almost entirely by the high-variance model. Any best return that is exceptionally large can only come from this model. So in a sense, we might as well not bother with the other models. Even moderately good models, if they have relatively low variance, will never be able to land far enough in the right tail to be significant. *Models that have high variance will dominate the competition, while models that have unusually low variance will be largely ignored.*

What happens when all of the competitors have roughly equal variance, but one of them is so bad that on average it actually performs worse than a random model? This causes an unavoidable problem. Understand that rejection of the null hypothesis does not always happen due to outstanding performance of the best model. It will often be the case that the best model is not so great that it clearly stands out. Sometimes a model that is not really the best will get lucky and not only beat the best, but also lead to rejection of the null hypothesis. At first glance this may seem to be undesirable. And in a way it is, because it means that the observed best model is not always the truly best model. Nonetheless, this occasional usurpation is vital to correct execution of the test. If it did not happen, the test would be excessively conservative. In order for the presence of unlucky good models to influence rejection of the null hypothesis, we need some other models to sometimes get lucky.

This necessary condition is subverted when one or more competitors are worse than random. Such models will find it difficult or impossible to get lucky enough to lead to rejection of the null hypothesis enough times to compensate for the times that a truly good model is unlucky. The test will be overly conservative.

Think of it this way. Suppose we have ten competing models, and our null hypothesis is that they are all random. Also suppose that unknown to us, one of these ten models is actually so much worse than random that it has no chance of ever being the best. In real life, we will be testing the best of a set of nine competitors, while the generated null distribution will be that of ten competitors. The best of ten models will be, on average, better than the best of just nine models. Thus, if one model is very bad, a truly good model will need to be better than if the other competitors were truly random, in order to make up for that one bad model that will never be lucky enough to reject the null hypothesis. *If the competitors include one or more models that are worse than random, the test will be conservative.*

Note that this does not mean that we should eliminate any models that show poor performance. An observed bad performer is different from a truly bad performer. If we were to simply go through the competitors and remove any that had a negative return, we would distort the test. Any model that was tested must be included. An observed poor performance may be due to an inherent flaw in the model, or it may be due to bad luck. We have no way of knowing. What we can do, though, is try to avoid even testing models that we have reason to believe may be bad. Sometimes this is possible, and sometimes it is not. This is yet another argument against wildly throwing spaghetti against the wall, to see what sticks.

A potentially disastrous situation is obtained when a rule has both poor average performance and unusually high variance. Look at Figure 3. The solid-line bell curve shows the assumed distribution of a set of competitors having random returns and equal variance. The dotted line shows the distribution of the best of these competitors. It is shifted to the right, as expected. The dashed line shows the distribution of a model that has a very negative expected value and a high variance. Observe that in real life this model will dominate the right tail of the distribution of the best performer. In other words, any observed best model whose performance is good enough to reject the null hypothesis will be virtually guaranteed to be a lucky instance of this spoiler model. A truly good model will have to be extremely, perhaps unrealistically good in order to outperform a lucky

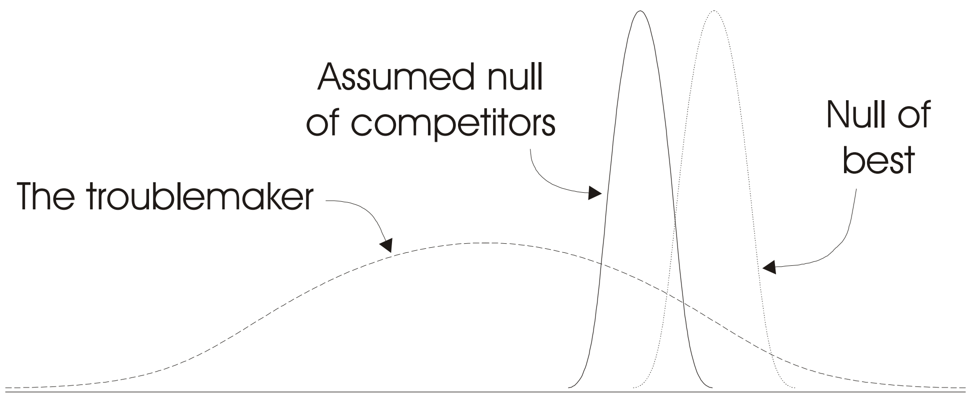

Figure 3 Trouble from a spoiler model.

spoiler. This is bad.

The Monte-Carlo permutation test generally handles this situation well. As long as the raw returns have similar variances and the returns are normalized according to the amount of time spent in the market, as already discussed, such high-variance domination will not occur. Thus, the test will experience the same conservative behavior that we saw in conjunction with a poor model having the same variance as its competitors. This is annoying, but almost never serious.

The bootstrap of the best model is another story. Recall that the observed mean return of each model is subtracted from the bootstrap sample mean in order to find the distribution of the departures from the mean. This means that the assumption of zero mean for all models is unavoidably implicit in the test. A bad high-variance model will dominate the right tail. In the extreme case of a very negative expected mean and a very large variance, the power of the bootstrap test will be reduced to essentially zero. The presence of even one such serious spoiler among the competitors will mean that it will be nearly impossible to reject the null hypothesis, even when one or more quite good models are included.

I am not aware of any general method for modifying the bootstrap test to deal with this problem, although position scaling similar to that recommended for the Monte-Carlo test will solve the problem when it is feasible. The moral of the story is that if you plan to bootstrap the best of several models, you must do your best to make sure that either all competitors have similar variance, or that no truly terrible models are included in the competition. *The combination of the two can render the bootstrap worthless.*

The good news is that the spoiler problem is rare nearly to the point of being non-existent in real-life market applications. I've worked with a wide variety of models in numerous commodity markets, and I've never seen a situation in which a serious spoiler model appears. If the competing models are all trading the same market, or if the different markets have their variances equalized, and if the competing systems are in the market for even roughly the same amounts of time, the variance of the competing returns will not come close to being disparate enough to cause spoiler problems. And trading systems that are significantly worse than random are difficult to find. In fact, if one were to be lucky enough to locate such a system, one could simply go long when it says to go short, and conversely, and rapidly become wealthy! The bottom line is that the spoiler problem is far more theoretical than practical, and hence multiple-model bootstrapping is almost always safe from its deleterious effects.

Permuting Thresholded Results

A common practice is to develop a trading model that makes numeric predictions (as opposed to simple class decisions like long/short/neutral). We then base each trade decision on the size of the prediction relative to a specified threshold. For example, we may train a model to predict the number of points that the market will move in the next day. After running the model on a batch of historical data, we may observe that the prediction exceeds, say, 3.7 points approximately ten percent of the time, and this upper decile provides many good trades. (Intelligently choosing the threshold is another story for another day. For now, assume that we have done so in a decent manner.) We may then decide that in the future, we will take a position if and only if the prediction is 3.7 or greater. With this in mind, we procure an independent historical dataset and test this rule. If the obtained total return is nicely positive, we rejoice in our brilliance.

But could the obtained return have arisen from nothing more than dumb luck? A Monte-Carlo permutation test can provide useful information to help answer this question. On the surface, this test seems to be quite different from the test described earlier. In fact, though, it is a simple specialization of the same test.

The test is performed by arranging the *n* actual returns in an array, and counting the number of them, *k*, whose predictions equal or exceed the pre-specified threshold. These *k* returns represent the trades that were taken. Use these *k* returns to compute a test statistic. This may be a mean return, total return, Sharpe ratio, profit factor, or any other performance measure that does not depend on order. (Actually, in most cases, even order-based measures such as the return-to-drawdown ratio can be used, although they tend to be too unstable for my taste.)

After computing the return, shuffle the array, choose the first *k* returns in the shuffled array, and compute the test statistic based on these *k* values. Repeat this a large number of times, perhaps 1000 to 10000 times. Count the fraction of these times that the test statistic from the shuffled data equaled or exceeded that from the original, unshuffled data. This is a good approximation of the probability that a truly worthless model could have performed as well as it did purely by luck. Unless this probability is small, be worried.

Note that this test is very easy to program and fast to execute. Yet it is enormously informative. For these reasons, any program that tests thresholded models should routinely include this algorithm and report its results to the user.

Summary

Monte-Carlo permutation tests have been around for several decades at least, and primitive forms existed even before high-speed computers became widely available. The reason is simple: They are generally easy to implement, they have fewer critical assumptions than most statistical tests, and in most cases they are fast to compute.

The fundamental basis of a Monte-Carlo permutation test is that if, under the null hypothesis, all possible permutations are equally likely, then we can generate the null-hypothesis distribution of our test statistic by randomly sampling a large number of permutations. Once we have this null distribution in hand, it is trivial to compute the fraction of its area that is at or beyond our observed value of the test statistic. And behold, we thus have the probability that our presumably great results could have been obtained by nothing more than good luck.

Of course, the assumption that all permutations are equally likely is critical to the validity of the test. As we saw in earlier discussions, certain designs of the trading system can violate this assumption, making the permutation test suspect, usually anti-conservative.

But don't despair. Apparent violations are often benign. For example, suppose we track performance daily. Also suppose that the rules that govern the trading system implicitly or explicitly demand that long positions stay open for exactly five consecutive days, and short systems stay open for ten consecutive days. At first glance it would appear that permuting the position vector would horribly violate this rule, meaning that most of the random perturbations could never happen in real life. But remember that we are only concerned with the *pairing* of positions with raw market returns. Exactly the same test would be obtained by permuting the returns, leaving the positions unchanged, in full compliance with the trading rule. If we are willing to assume that the raw market returns have little or no serial correlation, permuting them would be reasonable, and hence the test would be valid. Even if they do have noticeable serial correlation, the demonstration of Figure 2 on Page 9 shows that the impact of this violation is tolerable.

The obvious lesson here is that if we are dealing with a highly correlated market, we should, to the greatest degree possible, avoid designing trading systems whose position decisions are inherently correlated. Of course, the appearance of serially correlated positions does not automatically indict the system or statistical analysis of its results. If the system *on its own* comes up with strings of long or short positions, this is fine. Problems arise only when the system designer incorporates rules that produce serially correlated positions *regardless of the behavior of the market*. Following this guidance does not just make statistical tests of results more reliable. It has other, more practical effects that are beyond the scope of this document.

Finally, note that the potential problems with violations of the fundamental assumption are not weaknesses specific to Monte-Carlo permutation tests. If you look back at Figure 2, you will see that the two bootstrap tests were impacted even more severely than the permutation test. And parametric tests like the venerable *t-test* would have fared every bit as badly. This is because parametric and bootstrap tests absolutely require that the observations be independent. If the market and the position vectors *both* suffer serial correlation, individual returns will also be correlated. This is devastating to the tests. (There are *dependent bootstrap* tests that circumvent this problem, though at a considerable cost in power. My experience is that they are rarely worthwhile.) The truth is that Monte-Carlo permutation tests tend to be more robust than most other tests. This is good.