CHAPTER 6

Data Representation Techniques

In the Introduction, systematic trading was described as being an art, a science, and a business. This book focuses primarily on the scientific and business aspects. The scientific endeavor of systematic trading consists of discovering persistent and predictable patterns in market activity and the business aspect consists of efficiently exploiting them.

Any science starts with observational data—the raw materials of research on which concepts and theories are consequently built. However, not all data is relevant at all times, and more often than not, data overload can stall scientific progress, as one cannot see the forest for the trees when formulating useful concepts. Hence part of the art of the researcher comes in the form of an intuitive filter that helps decide what data to focus on. This chapter introduces the data filter techniques relevant to systematic trading and forms the basis for data analysis in the book.

6.1 DATA RELEVANCE AND FILTERING OF INFORMATION

The raw materials for systematic trading are (1) time series of transactions (price and volume), (2) time series of orders (depth of book), and (3) time series of news (economic releases, idiosyncratic company news, world events). Most of the models discussed in this book are based on the first two streams of data, however the third one is also briefly discussed in this chapter.

The question of what data is relevant is ultimately a function of the goal of the inquiry. In the world of systematic trading, this translates into understanding the timescale, price scale, and type of pattern one is trying to exploit. A data filter is usually formed of three components: (1) a sampling technique, (2) a compression technique, and (3) a representation technique.

In this book, all the trading strategies are discussed and implemented within the paradigm of distributed trading agents. In this paradigm, any

data consumed by the agent is represented by events that the agent observes and reacts to. This paradigm is the closest model to the real world and has the significant advantage that the concepts and code developed here apply equally well to real-time trading and to simulation environments. A comprehensive event-driven simulation environment is presented later in Part One.

Events fall into two classes, the **MARKETUPDATE** representing all external market data coming from exchanges and ECNs and the internal **COMM** events that carry inter-agent communications.

This chapter focuses on the market update events and the construction of more elaborate events from the sampling and compression of elementary ones. These elementary transaction and orders events fall into two categories:

1. A trade is performed and the price and volume information is communicated to the world.
2. The order book changes and the new state of the order book is communicated to the world.

Events are filtered by the trading agent via its `observe` method that not only can choose what securities prices to observe but in what format. Hence this method contains the sampling and compression techniques alluded to above.

6.2 PRICE AND ORDER BOOK UPDATES

Although introduced in the beginning, it is worth revisiting the object-oriented design of the main **EVENT** class.

Throughout Parts One to Three, it is assumed that the raw data that comes from the ECN (via either the FIX protocol or a more efficient ECN-specific protocol) is converted by a data adaptor to a class instance that is broadcast to the trading agents.

The top-level **EVENT** class contains a timestamp and a value (that can contain anything):

```
(defclass EVENT ()
  ((timestamp
    :initarg :timestamp
    :accessor timestamp)
   (value
    :initarg :value
    :accessor value)))
```

It is specialized for a market update event and contains an additional security identifier field:

```
(defclass MARKETUPDATE (EVENT)
  ((security
    :initarg :security
    :accessor security)))
```

6.2.1 Elementary Price Events

The most elementary market update events that can be observed from an ECN are single traded price updates and are represented by instances of the class **PRC**, a child class of **MARKETUPDATE**:

```
(defclass PRC (MARKETUPDATE)
  ((lastprice
    :accessor lastprice)
   (lastvolume
    :accessor lastvolume)))

(defmethod initialize-instance :after ((e PRC))
  (setf (lastprice e) (first (value e)))
  (when (second (value e))
    (setf (lastvolume e) (second (value e)))))
```

PRC's **value** field is a list with two elements that are the traded price and the volume traded at that price (if that information is available and NIL otherwise). These events basically represent the entries of the quote recap table, which is a feature available for all exchange-traded securities. The **initialize-instance :after** method populates the relevant fields that can be accessed by the agent. Here the use of the **initialize-instance :after** method may seem a bit superfluous but it is actually quite handy when the raw data is passed to the LISP class as a simple array from the ECN interface via a foreign-function implementation.

Figure 6.1 shows what a standard quote recap data time series looks like.

6.2.2 Order Book Data

The order book represents at a point in time the set of posted resting orders for a particular security at an exchange. It is basically two ordered lists of pairs: the bid side of the book is the collection of bid prices and respective sizes of aggregate buy orders at those prices; the offer side of the book is

Time	Size	Price
12:32:59	2	143.62
12:32:55	15	143.62
12:32:54	4	143.62
12:32:51	249	143.56
12:32:41	1	143.62
12:32:21	10	143.63
12:32:13	3	143.63
12:32:04	3	143.63

FIGURE 6.1 Quote Recap Data Example

the collection of ask prices with the respective size of aggregate offers to sell at those prices. The order book data was once the private information of stock specialists, but with the advent of the electronic markets, it became completely public and is displayed in real time on all trading applications (such as Trading Technologies or J-Trader).

Figure 6.2 is an illustration of a change in order book from one instant to another as well as the associated trade. Note that the trade size (in this case a buy at the best ask price) does correspond to the change in the best ask

Event 1		Event 2		Event 3	
Original book		Trade: Lift 50 lots at 143.58		Removed best offer for 20 lots	
143.62	120	143.62	120	143.62	120
143.61	89	143.61	89	143.61	89
143.6	388	143.6	388	143.6	388
143.59	430	143.59	430	143.59	430
143.58	76	143.58	26	143.58	6
143.57	489	143.57	489	143.57	489
143.56	236	143.56	236	143.56	236
143.55	184	143.55	184	143.55	184
143.54	303	143.54	303	143.54	303
143.53	95	143.53	95	143.53	95

FIGURE 6.2 Order Book Change Example

quantity. At the second instant the best ask quantity is reduced not because of a trade but because someone pulled an existing sell order (potentially in response to the trade that just happened). Here it is assumed that all the events come in the correctly time-stamped order from the ECN, but it has to be pointed out that in practice this is unfortunately not always the case, because of technological failings and latency issues that may occur at the exchange itself.

The state of the order book is represented by the **BOOK** class. The **value** field is a list of two lists l_b and l_a that contain lists of bids and bid sizes and asks and ask sizes. The **BOOK** class also has a specific method of initialization that sets up the best bids, offers, average bid, and offer prices and all associated sizes:

```
(defclass BOOK (MARKETUPDATE)
  ((mid
    :accessor mid)
   (bidbest
    :accessor bidbest)
   (bidbestsize
    :accessor bidbestsize)
   (bidtotsize
    :accessor bidtotsize)
   (bidavgprc
    :accessor bidavgprc)
   (askbest
    :accessor askbest)
   (askbestsize
    :accessor askbestsize)
   (asktotsize
    :accessor asktotsize)
   (askavgprc
    :accessor askavgprc)))

(defmethod initialize-instance :after ((e BOOK) &key)
  (let* ((v (value e))
         (lb (first v))
         (la (second v)))
(setf lb (sort lb #'(lambda (x y) (> (first x) (first y)))))
(setf la (sort la #'(lambda (x y) (< (first x) (first y)))))
(setf (bidbest e) (first (first lb)))
(setf (bidbestsize e) (second (first lb)))
(setf (bidtotsize e) (sum-list (mapcar #'second
                                       lb)))
```

```
(setf (bidavgprc e) (/ (sum-list (mapcar #'(lambda (x)
                                        (* (first x) (second x)))

                                        lb))
                       (bidtotsize e)))
(setf (askbest e) (first (first la)))
(setf (askbestsize e) (second (first la)))
(setf (asktotsize e) (sum-list (mapcar #'second
                                        la)))
(setf (askavgprc e) (/ (sum-list (mapcar #'(lambda (x)
                                        (* (first x) (second x)))
                                        la))
                       (bidtotsize e)))
(setf (mid e) (* 0.5 (+ (bidbest e)
                        (askbest e))))
(setf (value e) (list lb la))))
```

The list of bid-side and ask-side quotes contained in the **value** field of the class instance is processed automatically by the **initialize-instance :after** method and populates all the relevant fields that can be used by the agent's preprocessor and control system.

In practice, the ECN broadcasts the changes to the order book, not the book itself (unless specifically requested by the application). The **DELTA** class is designed to represent the change in the order book from one moment to the next. Those details are covered in Part Four and here one assumes that the trading system's interface to the ECN automatically converts these book updates into the instance of the **BOOK** class before broadcasting it to the agents.

6.2.3 Tick Data: The Finest Grain

The elementary market update events embodied in the **PRC** and **BOOK** events are colloquially called *ticks*. Tick data is the purest event data. The systematic trading activities for which tick data is *essential* are automated market-making and algorithmic trading. The strategies behind those activities use both the trade updates and the full book information to detect changes in supply and demand. Although a much more difficult task, one can also attempt to uncover hidden competing algorithms (like iceberg orders and others).

Collecting, processing, and storing tick data is a daunting task due to the sheer volume of information. An important tool in that matter is to use a networked cached memory that automatically saves chunks of data onto hardware without creating latency bottlenecks due to database read-writes.

Co-locating of trading servers on an ECN for ultrafast market and data access becomes essential for the efficient implementation of such activities because latency creates an unwanted sampling limitation. Indeed if the market access latency is L then the turnaround from receiving market data, processing it, and sending an order is limited from below by $2L$, and despite receiving all the event data, one is still limited in one's actions by latency-induced time-sampling. One should also be careful about the ECNs and brokers throttling data, where they do actually filter data at the outset and only send it by packets in order to not overload the network.

These practical matters are discussed in detail in Part Four of the book.

6.3 SAMPLING: CLOCK TIME VS. EVENT TIME

The two most important sampling techniques for financial data are time and event sampling. The difference between the commonly used clock time and the timescale derived from the occurence of elementary market events is explained here.

In time-interval sampling, one chooses a particular clock time interval of length T and observes a price at the end of each consecutive interval. For example, many traders build models using closing or last traded prices at the end of each trading day and hence react only to daily changes in prices. Longer-term models use last traded weekly or even monthly prices. Any intermediate data is simply omitted or ignored by those models. One can also sample intraday using sampling every T minutes or seconds.

The markets go periodically through bursts and troughs of activity, which are characterized by a varying volume of transactions in time. Time sampling is appropriate for longer term strategies on timescales where such varying activity is averaged out and is not deemed to influence decision making. However, a more event-driven approach is warranted for shorter term and intraday trading.

In event sampling, instead of sampling every T minutes, one samples every N^{th} event. This means that the sampling naturally accelerates and decelerates in clock time in step with the market activity. This difference between clock and event time is particularly relevant for short-term trading that needs to react quickly to changing market conditions as it usually exploits patterns stemming from such waves of activity.

The decision between clock and event sampling is the central decision for the relevant filtering technique when designing a model. The subsequent compression and representation techniques are then based on the same principles. To illustrate the point, Figure 6.3 shows two price graphs of the same intraday price action in clock and event time.

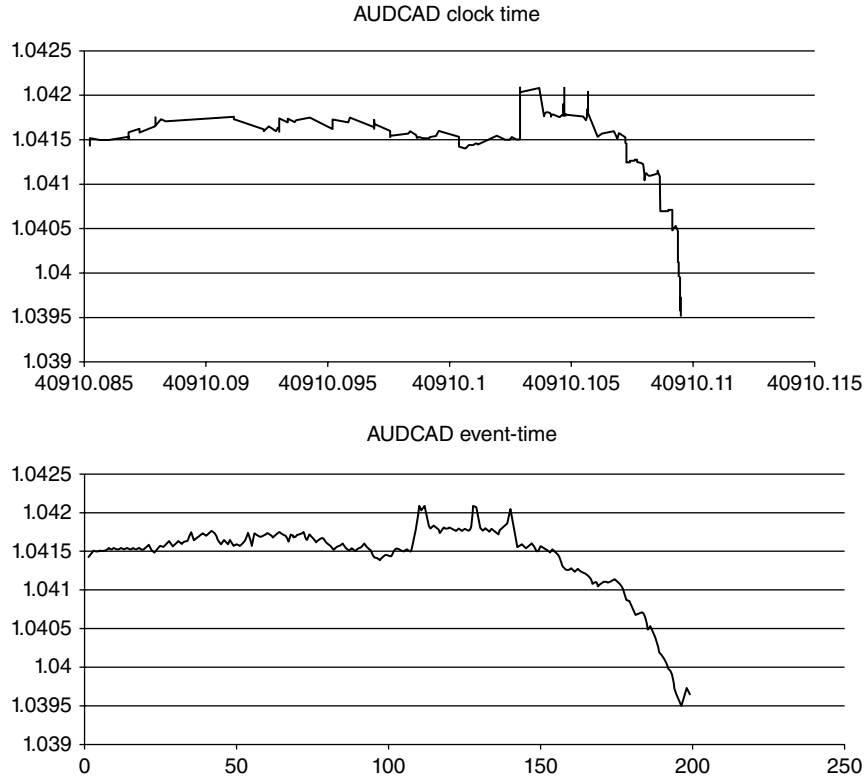

FIGURE 6.3 Price Action in Clock vs. Event Time

In the trading agent paradigm introduced in this book, there is no difference of implementation between models using event or clock time because clock-driven sampling is a subset of event-driven sampling. From a back-testing or forward-testing perspective, the processing of a stream of time-sampled or event-sampled prices is the same. The advantage of the design presented in the book is that it applies equally well for simulation and real-time trading environments.

6.4 COMPRESSION

6.4.1 Slicing Time into Bars and Candles

Some models use intermediate information between time or event sampled data. The most common compression technique is called a *bar* and carries

information on the beginning (opening), the high, the low, and the last (closing) price in each time interval. The following figure shows the 10-minute time-bars and the 1,000-event tick-bars and highlights the difference between the sampling techniques. The *range* of the bar is the distance between the high and the low price.

Bursts of volatility in the market are usually synonymous and concurrent with bursts in event activity, namely trading and order book changes. These events seem to accelerate in clock time. Symmetrically, when the markets are in transitions between time zones or during holidays, the event frequency comes down and so does volatility. Sampling with constant clock time slices through such periods of higher or lower activity yields respectively higher or lower ranges.

A *candle* is a bar of which the "body," that is, the range between the opening and closing price, is either filled, when the closing price is below the opening, or empty, if the closing price is above the opening. This gives a tool for the trader to represent graphically serial correlation of moves (short term trends). Figures 6.4 and 6.5 show representative bar and candle charts.

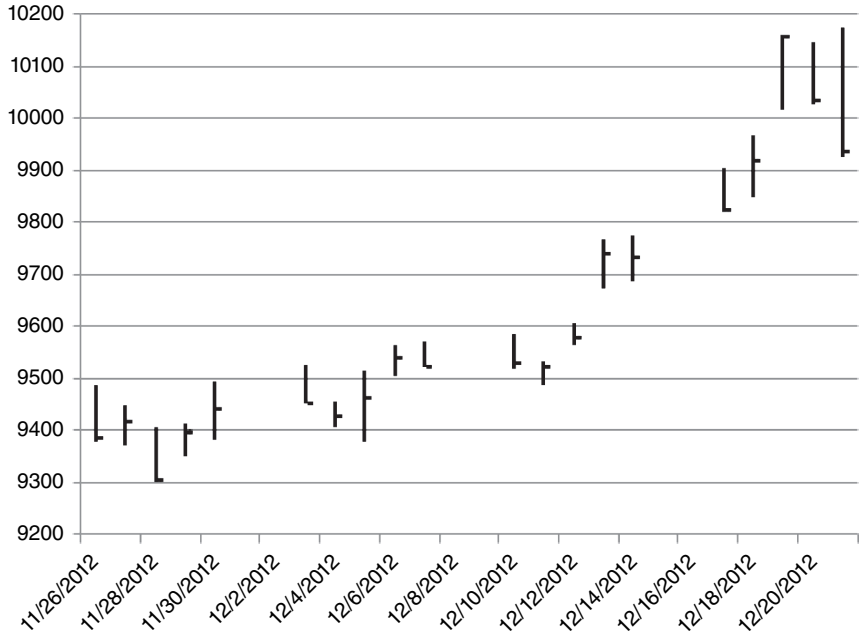

FIGURE 6.4 Bar Chart Example

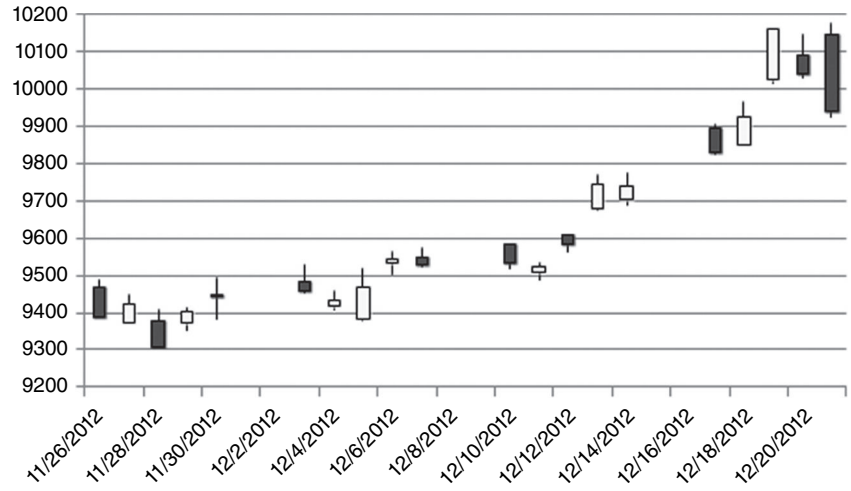

FIGURE 6.5 Candle Chart Example

The bars and candles are represented as instances of the class **BAR**:

```
(defclass BAR (MARKETUPDATE)
  ((pivot
    :accessor pivot)
   (O
    :accessor O)
   (H
    :accessor H)
   (L
    :accessor L)
   (C
    :accessor C)
   (bodyfill
    :accessor bodyfill)))

(defmethod initialize-instance :after ((e BAR) &key)
  (let ((v (value e)))
    (setf (pivot e) (avg-list v))
    (setf (O e) (first v))
    (setf (H e) (second v))
    (setf (L e) (third v))
    (setf (C e) (fourth v))
    (setf (bodyfill e) (if (>= (C e) (O e))
                           NIL
                         T))))
```

The **value** field of the class is just the list of the opening, high, low, and closing prices. The **initialize-instance :after** method populates the readable fields and also determines the candle's body fill field. Hence this class contains the candle information as well if needed.

The class is further specialized into event-driven and time-driven sampling:

```
(defclass TICKBAR (BAR)
  ((numticks
    :accessor numticks
    :initarg :numticks)))

(defclass TIMEBAR (BAR)
  ((numtimeunits
    :accessor numtimeunits
    :initarg :numtimeunits)
   (timeunit
    :accessor timeunit
    :initarg :timeunit
    :initform :MINUTE)))
```

Bars and candles can be easily generated on the fly given a stream of real-time or database-read **PRC** events. The illustration of it is given here for **TICKBAR**s and the methodology uses the very same concepts that underpin the agent-based paradigm introduced in the earlier chapters. Namely the **TICKBARGENERATOR** will be an agent that consumes **PRC** events and broadcasts **TICKBAR**s to the top of the ***events-queue*** stack. The class is defined by:

```
(defclass TICKBARGENERATOR (FSMAGENT)
  ((MKT
    :accessor MKT
    :initarg :MKT)
   (N
    :accessor N
    :initarg :N)
   (COUNTER
    :accessor COUNTER
    :initform 0)
   (BUFFER
    :accessor BUFFER
    :initform NIL)
   (OP
    :accessor OP
    :initform NIL)
```

```
   (HI
    :accessor HI
    :initform NIL)
   (LO
    :accessor LO
    :initform NIL)
   (CL
    :accessor CL
    :initform NIL)))
```

The original inputs are the security identifier **MKT** and the number of events **N** to build the bar from. For example one can define two bar generators for two different securities and event numbers:

```
(defparameter *b1* (make-instance
                    'TICKBARGENERATOR
                    :MKT "AAPL"
                    :N 5))

(defparameter *b2* (make-instance
                    'TICKBARGENERATOR
                    :MKT "MSFT"
                    :N 7))

(push *b2* *agents*)
(push *b1* *agents*)
(setf (recipientslist *a1*) *agents*)
(setf (recipientslist *a2*) *agents*)
```

As with any other FSM agent, one needs to define the filter, initialization, and preprocessor methods:

```
(defmethod observe ((a TICKBARGENERATOR) (e MARKETUPDATE))
   (and
    (equal (MKT a) (security e))
    (not (equal (type-of e) 'BAR))))

(defmethod initialize ((a TICKBARGENERATOR))
   (with-slots (MKT N states name) a
    (when (null states)
     (push :EMIT states)
     (setf name (concatenate 'string
                             "TICKBARGENERATOR_"
                             (format NIL "~A_~A" MKT N))))))
```

```
(defmethod preprocess ((a TICKBARGENERATOR) (e MARKETUPDATE))
  (with-slots (COUNTER BUFFER positions) a
              (push 0 positions)
              (setf COUNTER (length BUFFER))))
```

In this case the agent is not a trading agent and hence the **positions** list only contains zeros. Also as the agent emits the bars to the ***events-queue*** it should not observe any bars by definition.

The FSM representation is very simple and contains only 2 states, **:CALC** that creates the bar from the stream and **:EMIT** when it emits it to the queue:

```
(defmethod setfsm ((a TICKBARGENERATOR))
  (with-slots (MKT N COUNTER BUFFER OP HI LO CL states currentstate
                   revalprices transitions positions name) a
    (setf currentstate (first states))
    (setf transitions (list
                       (make-instance
                       'TRANSITION
                       :initialstate :CALC
                       :finalstate :CALC
                       :sensor #'price
                       :predicate #'(lambda (p)
                                      (< COUNTER N))
                       :actuator #'(lambda (p)
                                     (setf CL p)
                                     (setf HI (max HI p))
                                     (setf LO (min LO p))
                                     (push p BUFFER)
                                     (format T
                                             "~S CALC->CALC  ~%"
                                             name)))
                       (make-instance
                        'TRANSITION
                        :initialstate :CALC
                        :finalstate :EMIT
                        :sensor #'price
                        :predicate #'(lambda (p)
                                       (equal COUNTER N))
                        :actuator #'(lambda (p)
                                      (emit a (make-instance
                                               'TICKBAR
                                               :timestamp (first
                                                (timestamps a))
                                               :security MKT
```

```
                                :value (list OP HI LO
                                 CL)
                                :numticks N))
                        (setf BUFFER NIL)
                        (format T
                                "~S CALC->EMIT  ~%"
                                name)))
(make-instance
 'TRANSITION
  :initialstate :EMIT
  :finalstate :CALC
  :sensor #'price
  :predicate #'(lambda (p)
                 T)
  :actuator #'(lambda (p)
                (push p BUFFER)
                (setf OP p)
                (setf HI p)
                (setf LO p)
                (setf CL p)
                (format T
                        "~S EMIT->CALC  ~%"
                        name)))
(make-instance
 'TRANSITION
 :initialstate :EMIT
 :finalstate :EMIT
 :sensor #'price
 :predicate #'(lambda (p)
                NIL)
 :actuator #'(lambda (p)
                NIL))))))
```

Finally, there is no need for a postprocessor. The other agents in the ***agents*** list would be able to observe the emitted bar as soon as the last relevant **PRC** event is processed by the bar generator and emitted on the events queue.

6.4.2 Slicing Price into Boxes

A useful and complementary compression technique focuses solely on the price dimension. One starts with a price or return scale B and the current price $P(0)$. The intuitive idea is as follows: Whenever the market is in an uptrend and does not reverse by more than B from its local high, a series

of ascending boxes of height B are drawn on top of each other. When the market finally reverses by more than B from its top, a new set of descending boxes is drawn one notch to the right. Figure 6.6 gives the FSM representation for the box chart agent and Figure 6.7 the resulting chart example:

Box charts remove the time dimension (be it clock or event time) because each box represents a state of the market where no opposite move occurs relative to the previous trend state (as measured by the box size, i.e., price scale B). The market can stay a long time or a little time in that state. The box charts are useful compression techniques to automate the recognition of chart patterns as we will see in a subsequent chapter.

The size B of the box dictates the price scale of the patterns that appear from such a compression and also, indirectly, their time scale. The relationship between the price and time scales is a function of volatility that varies in time. To produce more consistent price-time scale relationships one can adapt the box size to volatility V. A linear scaling $B_i = \alpha V_{i-1}$ is the simplest example where the next box size is chosen as a function of the average volatility that occured while the market was in the previous box, series of boxes, or a fixed time period.

6.4.3 Market Distributions

Another interesting compression technique is the price distribution over a time period. Namely, one divides the price scale into intervals of length B and fills a horizontal bar between each division as a function of the frequency of occurence of the price in that interval during the time period. Usually the time over which a distribution is accumulated is a trading day.

Figure 6.8 presents the time series of daily price distributions and shows the occurence of unimodal and bimodal daily price distributions. Bimodal distributions occur when news moves the market from one level to another around which the price then oscillates. Using the distributions can be useful for certain intraday mean-reversion models and for some algorithmic execution applications.

6.5 REPRESENTATION

Once data has been sliced and diced by appropriate sampling and compression, different representation techniques can be applied. Representation can be seen as a coding technique and is different for data fed to humans or to machines.

Box Chart Agent							
L<H	S is the Size of the Box						
H-L = S	Box is either a CROSS (local uptrend) or CIRCLE (local downtrend)						
	L and H are the Low Level and the High Level of the Current Box						
	A new CROSS is drawn to the top of current one or to the top right of the current CIRCLE						
	A new CIRCLE is drawn to the bottom of current one or to the bottom right of the current CROSS						
	START	**SAME-CROSS**	**NEW-CROSS-HIGHER**	**NEW-CROSS-RIGHT**	**SAME-CIRCLE**	**NEW-CIRCLE-LOWER**	**NEW-CIRCLE-RIGHT**
START	NIL	NIL	NIL	NIL	NIL	NIL	NIL
SAME-CROSS	T	L <= p <= H	T	T	NIL	NIL	NIL
NEW-CROSS-HIGHER	NIL	p > H	NIL	NIL	NIL	NIL	NIL
NEW-CROSS-RIGHT	NIL	NIL	NIL	NIL	p > H	NIL	NIL
SAME-CIRCLE	NIL	NIL	NIL	NIL	L <= p <= H	T	T
NEW-CIRCLE-LOWER	NIL	NIL	NIL	NIL	p < L	NIL	NIL
NEW-CIRCLE-RIGHT	NIL	p < L	NIL	NIL	NIL	NIL	NIL

FIGURE 6.6 Box Chart Pseudocode

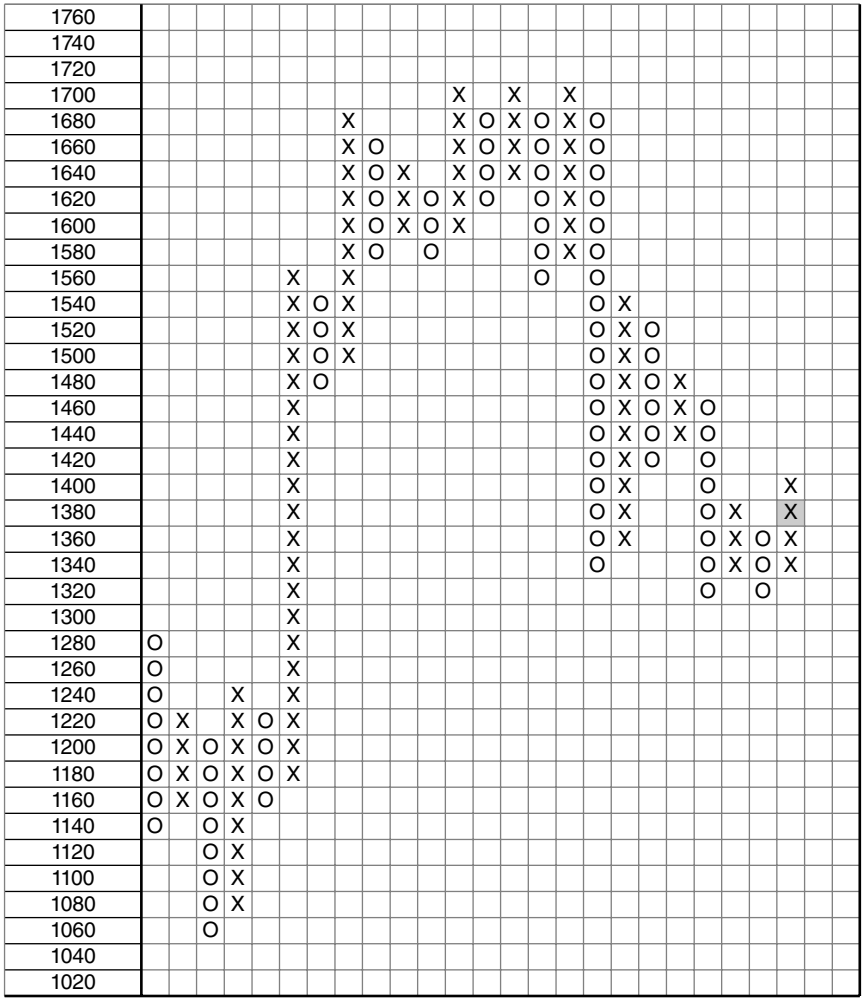

FIGURE 6.7 Box Chart Example

6.5.1 Charts and Technical Analysis

Most commonly, representation is of graphical nature and is designed to help the human trader recognize patterns visually. All the charts shown above are compressed sampled data represented in a particular way as a time series.

The most important feature of a graphically represented time series is the *memory* that is embedded in it. The human eye is very well trained to

740									
735									
730		A							
725		A	B						
720		A	A	B					
715		B							
710		B	C	D					
705		B	B	C	D	D			
700		C	C	C	D				
695		D							
690									

FIGURE 6.8 Market Distribution Chart Example

recognize patterns in pictures. Time series are particular pictures that represent an unfolding history. Part of the game is, given such a history sample, to predict the next set of events. This is the central focus of technical analysis.

There is a whole plethora of market patterns that have varying predictive power as to the future evolution of the price action (see Bulkowski, *The Encyclopedia of Chart Patterns*, 2005). The patterns that have proven to have superior predictive power and are amenable to coding efficiently are (1) linear trend channels, (2) breakout from volatility compression (triangles and pennants), (3) breakout or deceleration around support and resistance

FIGURE 6.9 Examples of Volatility Compression Patterns

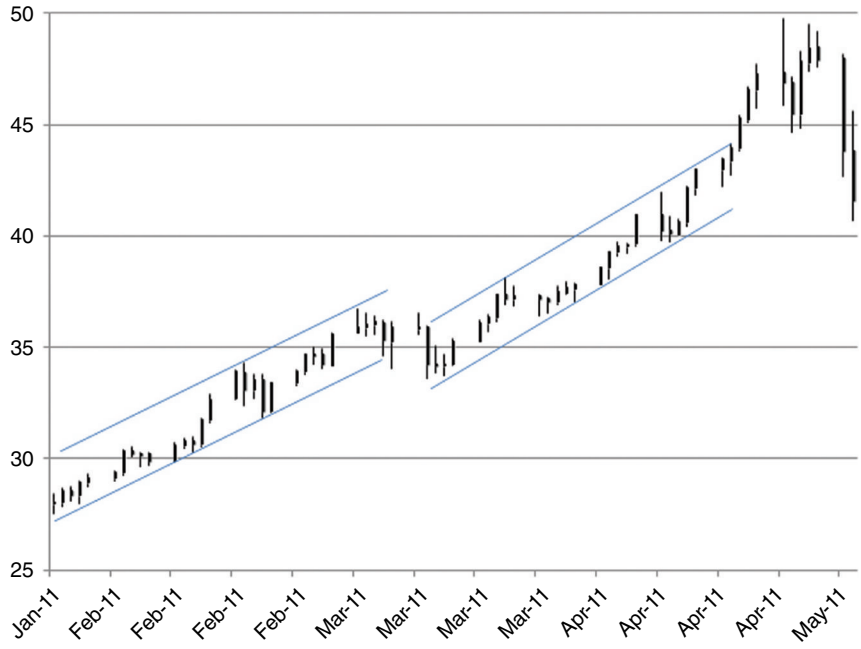

FIGURE 6.10 Examples of Linear Patterns

(including double tops and bottoms, head and shoulders), and (4) trend change via breaking linear trend channels.

The examples given in Figure 6.9 and in Figure 6.10 show several bar and candle charts in time and event sampling that exhibit some common behavioral set up patterns and subsequent market activity. Note that when looking at longer-term charts with large price moves it is more convenient to use a logarithmic scale, as shown in Figure 6.11. The markets tend to move in return space rather than price space at those larger scales and trends that appear nonlinear (exponentially accelerating) on a linear scale become linear on a logarithmic scale.

6.5.2 Translating Patterns into Symbols

Some chart patterns can be recognized algorithmically (examples of code are provided in the next chapter). This opens the door to study whether the occurence of patterns is not random but presents certain statistical rules. One can almost hear the phrase used by some experienced traders: "Hear the market speak." To test such serial correlation one can encode each recognized pattern into a symbol (a letter for example) and study the

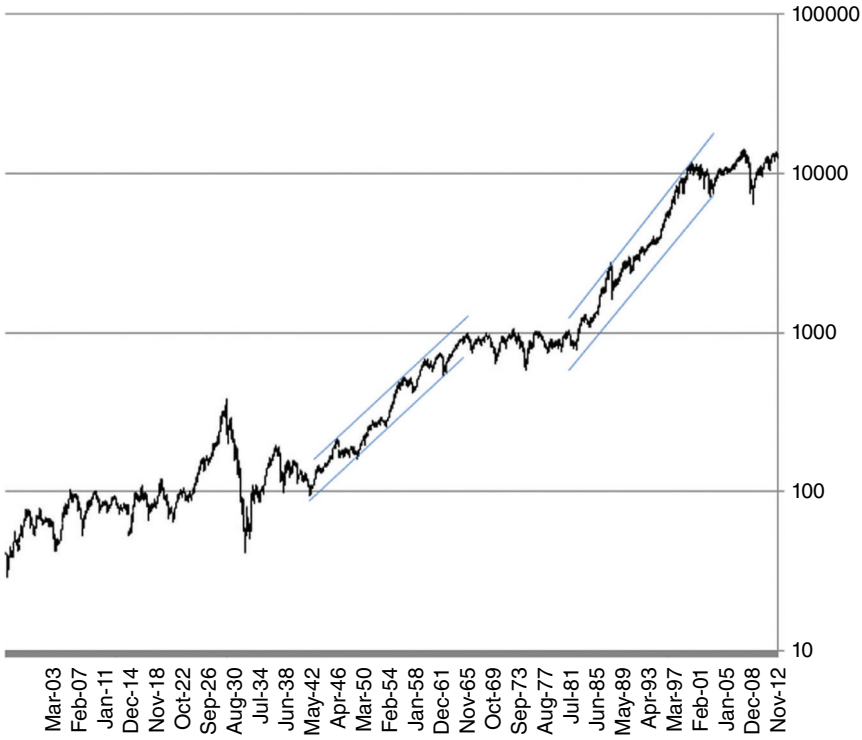

FIGURE 6.11 Example of Trend Channel in Log Scale

statistics of words that are thus generated by the time series of patterns. One can also test for robustness of such analysis by running the test on different markets at different time and event scales.

Figure 6.12 is an example that illustrates the repeated occurence of the following most common pattern:

$$(\textit{Volatility Contraction}) \rightarrow (\textit{Trend}) \rightarrow (\textit{Volatility Expansion}) \rightarrow \cdots$$
$$\cdots \rightarrow (\textit{Range}) \rightarrow (\textit{Volatility Contraction}) \rightarrow \cdots$$

6.5.3 Translating News into Numbers

The modern world is awash with news that hits us from all sides: TV, financial news services such as Bloomberg and Reuters, the Internet, and so on. It is impossible to absorb all this data without some filtering. With the

FIGURE 6.12 Contraction-Trend-Expansion-Range Pattern

advent of the streaming news services it has become possible to automatically process that data using language recognition techniques. News pieces are events of a very different nature from prices—they are semantic in nature and communicated to the world in a symbolic language rather than in numeric format.

Language recognition is an active area of research in artificial intelligence. It is a difficult problem because syntax does not imply semantics and phrases usually cannot be taken out of context. The field touches on the most subtle areas of framing and building context (i.e., basic understanding) through learning. Most of computing is performed on numeric problems, and symbolic and semantic problems have always been harder to solve on the prevailing computing architecture. Nontheless languages like LISP (used here) and Prolog have been specifically developed to deal with symbolic problems and the most powerful pattern-matchers and context generators are written in them.

For trading, the most useful understanding of a piece of news is whether the information embedded in it is going to have a positive or negative impact on the price of a security. Some news affects particular securities (like corporate announcements for a stock) and some news affects the market as a whole (like unexpected monetary policy changes). Hence one needs to be able to correlate a semantic datum with a numeric one.

Some reasonably simple techniques have been developed that compute a numerical sentiment index for a given stock. Those techniques process in real time all the news pertaining to that stock and some general news that may affect the whole market. The output of the process for each piece of

news is a number that measures the positiveness of the news for that stock. The sentiment index is then computed by adding (integrating) these numbers over time or over a rolling time window.

It is not clear yet how effective such techniques are in systematic trading and research is in progress in this field. There is a prevailing feeling that prices tend to move faster and anticipate news generally. When totally unexpected events occur (shocks to sentiment about a particular stock or to the market as a whole), liquidity dries up very quickly and the reaction to prices can be so violent that one has the impression that the price moves ahead of the news.

6.5.4 Psychology of Data and Alerts

The representation of data has two goals—informative and psychological—and so far I have discussed the informative aspect. The psychological goal is to help traders anticipate better and react to market moves faster. As was pointed out in the Introduction, trading cannot be seen outside of a risk management context. Any representation of data that sharpens the mind with regard to risk management is therefore useful for human traders.

In an open outcry context, the traders could sense the danger and opportunity by the noise level of the pit and the facial expressions of their fellows. In an electronic context that information is not available, but there are ways to substitute it, at least partially.

For example, the increase in market volume and velocity of transactions can be represented graphically. Also, the transactions can be sent to the loudspeaker by a voice synthetizer that can simulate nervousness via the speed of arrival of trades combined with the increase in volume of orders on the book.

Of course, such gadgets may be detrimental to the trader who, instead of keeping a clear head, could be drawn into the market hysteria and overreact. Nevertheless, they can present an advantage by helping a human trader to not be continuously glued to the screen.

One major feature of systematic trading is to remove the psychological element from the trading decision. This does not mean that the psychology of the market should not be an input into the decision making—in fact, it is exactly what makes the various tradable patterns reoccur. Hence measuring such psychological changes is useful.