Writing an Indicator Cookbook

Thomas Weigert, weigert@mst.edu

March 2008

1 Introduction

This note attempts to summarize lessons learned writing GT indicators, in the hope that this will be useful to other GT developers. I shall use the Bollinger Band indicator as a starting point and annotate its implementation. Subsequent sections will illustrate more advanced aspects of writing indicators.

In terms of terminology, an indicator defines a time series. In general, a time series is a sequence of data values, ordered linearly by time. The prices of a market are also a time series (and also an indicator, see I:Prices). An indicator is constructed by some or all of the following:

- one or more time series
- an application of a transformation to a time series
- the sequential application of a computation on the individual values of one or more time series.

In the following, code fragments are typeset in red courier font. Throughout the examples, $self refers to an indicator object. Numbered code lines are from the code for the Bollinger Band indicator; unnumbered code lines are other examples.

2 Header

The first section sets up the module and loads the necessary dependencies.

2.1 Package definition and object initialization

Define a package for this indicator. The following use clauses are standard for all indicators. Then define this package to be an instance of the indicator object.

```
1 package GT::Indicators::BOL;
2
3 use strict;
4 use vars qw(@ISA @NAMES @DEFAULT_ARGS);
5 use GT::Indicators;
6 @ISA = qw(GT::Indicators);
```

2.2 Included packages

Load all packages this indicator depends on. For example, Bollinger Bands are moving averages that envelope a securities price. It consists of three series: A simple moving average of the security, and two series plotted n standard deviation levels above and below the moving average. Thus, in this particular case, we need three other indicators: the I:SMA, I:StandardDeviation, and I:Prices.

```
7 use GT::Indicators::SMA;
8 use GT::Indicators::StandardDeviation;
9 use GT::Prices;
```

2.3 Input parameters

The default argument statement defines the default values for the input parameters of the indicator. These are either constant values or they depend on the current value of another series. In this example, the first two parameters are by default given the constant values 20 and 2, respectively. The third parameter is the result of evaluating the indicator {I:Prices CLOSE} at the current period, yielding the current close of the prices array.

```
10 @DEFAULT_ARGS = (20, 2, "{I:Prices CLOSE}");
```

The arguments of the indicator are accessed via the methods, where $n is the number of the argument (starting at 1), $calc is a calculator, and $i is the current period.

```
$self->{args}->get_arg_constant($n)
$self->{args}->get_arg_names($n)
$self->{args}->get_arg_values($calc, $i, $n)
```

The first form requires that the argument is a constant, which can be tested by

```
$self->{args}->get_arg_constant($n).
```

The second form will obtain names of the corresponding argument (for constant arguments, the name is the same as its value).[1] The final form obtains both constant and non-constant values for a given period (of course, constants are the same for all periods).

Most indicators as currently defined perform no type checking on their parameters, resulting in fatal errors when parameters of the wrong type are passed. Care must be taken to pass constant parameters when such are expected, and time series as parameters, where those are required.[2]

3 Output values

Indicators will produce one or more output series. In other words, for each period, and indicator will output one or more values. Output values are defined in the names clause:

```
11 @NAMES = ("BOL[#1,#3]","BOLSup[#1,#2,#3]","BOLInf[#1,#2,#3]");
```

In this instance, we define three output series for Bollinger Bands: the moving average, and the upper and lower bands. These values can be referred to by the names given in the above clause, where the arguments (hashed numbers in brackets) are replaced by the

[1] Several indicators use get_arg_names in a context where the argument is not guaranteed to be constant, and thus will fail when a non-constant parameter is given (e.g., the name of a series).

[2] Preoper run time checking of parameter types is advisable when writing a new indicator.

corresponding input parameters of the indicator. The symbol `#*` is replaced by all input parameters of the indicator.

The name of an indicator is the name of the first output series. Therefore, `{I:BOL 12 2}` and `{I:BOL 12 3}` have the same name `BOL[12, {I:Prices CLOSE]]`. These two series cannot be distinguished when they are both used at the same time. Care should be taken to name the output series wisely so that there are no conflicts between indicators. The name of an output series can be either used literally as a string, or it can be obtained by the `get_name` method:

```
$self->get_name
$self->get_name($n)
```

where `$n` refers to the position of the output series (starting at `0`).

The values of the output series are set and read via a calculator `$calc`, where `$name` is the name of the output series and `$i` is the period:

```
$calc->indicators->get($name, $i)
$calc->indicators->set($name, $i, $value)
```

4 Initialization

If an indicator requires intermediate series to compute its value or requires data from past periods, these are set up in the initialization method. This method is passed an indicator object as the single parameter:

```
12 sub initialize {
13     my ($self) = @_;
```

4.1 Intermediate series

Many indicators depend in their computation on other series. For example, Bollinger bands need the simple moving average of the price of the security for each period, as well as the standard deviation of the price of the security for each period. Both of these intermediate values form a series. Each intermediate series must be created in the initialization method and be assigned to an attribute of the indicator. A series is created by calling the new method on its class and passing the appropriate arguments, or by evaluating the textual representation of the indicator defining the series. These intermediate series may rely on output values or on temporary data, see Section 7.2. For Bollinger Band we define the I:SMA and I:StandardDeviation as intermediate series, passing both the first (period) and third (data array the indicator is applied to, typically I:Prices) arguments. If the indicator was not given any parameters upon creation, the default values are used.

```
14     $self->{'sma'} = GT::Indicators::SMA->new([
15         $self->{'args'}->get_arg_names(1), $self->{'args'}->get_arg_names(3)]);
16     $self->{'sd'} = GT::Indicators::StandardDeviation->new([
17         $self->{'args'}->get_arg_names(1), $self->{'args'}->get_arg_names(3)]);
```

Note that when such an intermediate series uses other series as its arguments, these cannot be defined by their constructor functions but must be given in their textual representation. For example, the following doubly smoothes the SMA above:

```
$self->{'sma'} = GT::Indicators::SMA->new([
    $self->{'args'}->get_arg_names(1),
    '{I:SMA '.$self->{'args'}->get_arg_names(1).' '
        .$self->{'args'}->get_arg_names(3).'}']);
```

An intermediate series can also conveniently be constructed using the GT::Eval::create_standard_object method:

```
$self->{'sma2'} = GT::Eval::create_standard_object("I:SMA", "12 {I:Prices CLOSE}");
```

During the computation of the indicator, the intermediate series is either computed via the dependency mechanism (see Section 4.2) or by explicitly computing the series via:

```
$self->{'sma'}->calculate($calc, $i)
$self->{'sma'}->calculate_interval($calc, $i, $j)
```

where $calc is a calculator, and $i and $j are time periods. The values of these series are obtained via the standard get method, e.g., the ith value of the SMA is obtained via

```
$calc->indicators->set($self->{'sma'}->get_name, $i)
```

4.2 Dependencies

Many indicators depend on past data to calculate their current value, either on past price information, or on the previous values of the indicator or on the previous value of intermediate series. A key feature of GT is that the computation of those past values can be largely driven automatically through a dependency mechanism. We can declare the current value of an indicator to be dependent on the previous values of its parameters, or of other series, or of the price information it is operating on. Such dependencies are declared for n periods of data; when updating dependencies those n values will be ensured to be available. To satisfy dependencies may in turn require additional data, the computing the dependencies may in turn depend on other values. The dependency mechanism propagates automatically until all dependencies are satisfied.

Determining the correct dependencies is important to be able to compute the indicator both correctly and efficiently. If too little data is available, an indicator may not be able to be computed, at best, or may give incorrect results, at worst. If too much data is required, less history of an indicator can be computed.

When the dependencies are known at the time the indicator is created, the dependencies are defined in the initialization section (volatile dependencies see Section 7.3, allow dependencies to be computed dynamically during the computation of the values of the indicator). The following methods can be used to define dependencies:

```
$self->add_indicator_dependency($indic, $p)
$self->add_arg_dependency($n, $p)
$self->add_prices_dependency($p)
```

where $indic is an indicator, $n refers to the n-th parameter of the indicator (counting from 1), and $p is the number of periods of data this value depends on. The first form states that the current value of the indicator depends on $p periods of data of indicator $indic. The second form states that the indicator depends on $p periods of data referenced by parameter $n. The third form states that the indicator depends on $p periods of data of the input series (this form of dependency is only needed when the indicator depends on more data periods than is established by the dependency mechanism).[3]

[3] While it can be found in a number of indicators, this dependency is rarely (if ever) needed.

For the Bollinger Bands, each value depends on the current value of the moving average and the standard deviation. However, dependencies require at least one day of data, and thus the below declare the current value to be dependent of 1 day of data of the moving average and the standard deviation. Note that each of these in turn require data to be computed, but that dependency is declared as part of the definition of these indicators, and is automatically accounted for by the dependency mechanism.

```
18      $self->add_indicator_dependency($self->{'sma'}, 1);
19      $self->add_indicator_dependency($self->{'sd'}, 1);
```

The Bollinger Band indicator in addition establishes a dependency on the period passed as the first parameter, assuming that parameter is constant. However, this declaration is technically not necessary, as this dependency is already established by the dependencies of the intermediate series.

```
20      if ($self->{'args'}->is_constant(1)) {
21          $self->add_prices_dependency($self->{'args'}->get_arg_constant(1));
22      }
23
24  }
```

5 Calculating the value of the indicator

The GT framework provides two means of calculating the value of an indicator: we can either compute a single value of the indicator, given its dependencies, or we can compute the value of the indicator throughout a given interval. One or the other of these methods must be defined,[4] albeit often both methods are given. Typically, calculating the value of the indicator over the full interval required will be faster, potentially much faster as calculating the value of the indicator one period at a time may often repeat much of the computation needlessly.

5.1 Calculating a single value of the indicator

The current value of the indicator is computed by the calculate method, which takes as arguments a calculator and the current period. This method typically follows the following steps:

```
25  sub calculate {
26      my ($self, $calc, $i) = @_;
```

Define temporary variables. Several temporaries are defined for convenience: The distance of the upper and lower bands from the moving average, as determined by the second parameter, the names of the intermediate series used, and the names of the output values.

```
27      my $nsd = $self->{'args'}->get_arg_values($calc, $i, 2);
28      my $sma_name = $self->{'sma'}->get_name;
29      my $sd_name = $self->{'sd'}->get_name;
30      my $bol_name = $self->get_name(0);
```

[4] Note that if the calculate method is omitted, the indicator may fail if this method is indirectly invoked (e.g., when running anashell.pl), as this method is not defined in the superclass. It is safer to omit the calculate_interval method.

```
31      my $bolsup_name = $self->get_name(1);
32      my $bolinf_name = $self->get_name(2);
```

Return if the required values of the indicator are already available. These may have been computed earlier.

```
34      return if ($calc->indicators->is_available($bol_name, $i) &&
35                 $calc->indicators->is_available($bolsup_name, $i) &&
36                 $calc->indicators->is_available($bolinf_name, $i));
```

Return if the dependencies required to compute the value of this indicator are not satisfied. This check will attempt to compute the dependencies but fail when the dependencies cannot be computed. This triggers the dependency mechanism.

```
37      return if (! $self->check_dependencies($calc, $i));
```

Compute the current value of the indicator. For the Bollinger Band indicator, we first obtain the values of the moving average and the standard deviation. The upper band is obtained by adding the appropriate factor of the standard deviation to the moving average; the lower band is calculated similarly.

```
39      my $sma_value = $calc->indicators->get($sma_name, $i);
40      my $sd_value = $calc->indicators->get($sd_name, $i);
41
42      my $bolsup_value = $sma_value + ($nsd * $sd_value);
43      my $bolinf_value = $sma_value - ($nsd * $sd_value);
```

Note that computing the current value of the indicator may in fact require iterating over past periods.

Update the output values for the current period. For the Bollinger Band store the moving average value into the first output series, the upper band value into the second output series, and the lower band value into the last output series.

```
45      $calc->indicators->set($bol_name, $i, $sma_value);
46      $calc->indicators->set($bolsup_name, $i, $bolsup_value);
47      $calc->indicators->set($bolinf_name, $i, $bolinf_value);
48  }
```

5.2 Calculating a the indicator throughout an interval

The calculate_interval method computes the value of the indicator over a given interval. It is passed a calculator as well as the beginning and end of the interval of interest. This method can be obtained systematically from the calculate method by the following steps:

1. Change all occurrences of get_arg_values to the corresponding get_arg_constant
2. Change all occurrences of check_dependencies to the corresponding check_dependencies_interval
3. Change all occurrences of is_available to the corresponding is_available_interval
4. Compute the current value of the indicator within a loop from the beginning of the interval to the end of the interval.

```
sub calculate_interval {
    my ($self, $calc, $first, $last) = @_;
    my $nsd = $self->{'args'}->get_arg_constant(2);
    my $sma_name = $self->{'sma'}->get_name;
    my $sd_name = $self->{'sd'}->get_name;
    my $bol_name = $self->get_name(0);
    my $bolsup_name = $self->get_name(1);
    my $bolinf_name = $self->get_name(2);

    return if ($calc->indicators->is_available_interval($bol_name, $first, $last) &&
               $calc->indicators->is_available_interval($bolsup_name, $first, $last) &&
               $calc->indicators->is_available_interval($bolinf_name, $first, $last));
    return if (! $self->check_dependencies_interval($calc, $first, $last));

    for (my $i=$first;$i<=$last;$i++) {
      my $sma_value = $calc->indicators->get($sma_name, $i);
      my $sd_value = $calc->indicators->get($sd_name, $i);

      my $bolsup_value = $sma_value + ($nsd * $sd_value);
      my $bolinf_value = $sma_value - ($nsd * $sd_value);

      $calc->indicators->set($bol_name, $i, $sma_value);
      $calc->indicators->set($bolsup_name, $i, $bolsup_value);
      $calc->indicators->set($bolinf_name, $i, $bolinf_value);
    }
}
```

If this method is not provided, it is inherited from the indicator object and falls back on `calculate`. Typically, a provided `calculate_interval` method would not invoke calculate.

6 End of file

As common practice in Perl modules, conclude the file with a successful value.

```
1;
```

7 Additional capabilities

There are a number of additional tools provided by GT which are not leveraged in the Bollinger Bands indicator illustrated above. These are discussed below.

7.1 Temporary series

In addition to storing results in output values, as discussed in Section 3, and indicator may also store data into temporary series that are not visible outside of the indicator. To create a temporary series, assign a I:Generic:Container indicator to an attribute of the indicator object:

```
$self->{'temp'} = GT::Indicators::Generic::Container->new(['temp']);
```

Above creates a new temporary series with the name `temp`. This series is an indicator and thus values can be read and written to this series as to any indicator:

```
$calc->indicators->get($self->{'temp'}->get_name, $i)
$calc->indicators->set($self->{'temp'}->get_name, $i, $value)
```

7.2 Constructing intermediate series from other series

An intermediate series may rely on another intermediate series, on a temporary series, or on an output series. In this situation, when defining an intermediate series, the dependent series are provided as parameters.

For example, to define a standard moving average of the upper band of the Bollinger Band indicator (within the computation of the Bollinger Band indicator) use:

```
$self->{'upper'} = GT::Indicators::SMA->new([$self->{'args'}->get_arg_names(1),
                                "{I:Generic:ByName " . $self->get_name(1) . "}"]);
```

This constructs an intermediate SMA from the second output series of the current indicator, with the period taken from the first parameter of the current indicator and assigns it to an attribute of the indicator object. The indicator I:Generic:ByName references another series by its name (i.e., the name of the first output series, see Section 3). Care must be taken that the correct name is used.

Similarly one can construct a series that depends on a temporary series or an intermediate series. For example, the simple moving average of the temp indicator from Section 7.1 is defined as follows:

```
$self->{'sma1'} = GT::Indicators::SMA->new([ $self->{'args'}->get_arg_names(1),
                              "{I:Generic:ByName temp}" ]);
```

The further smoothing of the simple moving average of the upper Bollinger Band (see above) can be defined by[5]

```
$self->{'sma2'} = GT::Indicators::SMA->new([ $self->{'args'}->get_arg_names(1),
                            "{I:Generic:ByName '" . $self->{'upper'}->get_name . "}" ]);
```

If the intermediate series has multiple outputs, the proper name must be used (e.g., use get_name($n) to construct a series based on the nth output value of the intermediate series.

7.3 Volatile dependencies

It is also possible for indicator dependencies to dynamically change during the computation of a series, either by the length of the dependency being computed at each iteration, or by it depending on the value of a series. Dynamically changing dependencies are referred to as "volatile". They are defined analogously to static dependencies using the following methods, where $indic is an indicator, $n refers to the n-th parameter of the indicator (counting from 1), and $p is the number of periods of data this value depends on:

```
$self->add_volatile_indicator_dependency($indic, $p)
$self->add_volatile_arg_dependency($n, $p)
$self->add_volatile_prices_dependency($p)
```

Before defining volatile dependencies, all volatile dependencies from the previous period must be removed through calling

```
$self->remove_volatile_dependencies()
```

Volatile dependencies are mostly useful only when indicators are calculated one period at a time (i.e., in the calculate method).[6]

[5] This requires a correction to the I:Generic:ByName indicator available from the mailing list archives at http://www.geniustrader.org/lists/devel/msg02362.html.

[6] Note that several indicators add volatile indicators in the calculate_interval method. This will work only if careful attention is paid to that the dependency period is correctly obtained. In many such situations,

8 Styles of calculating indicators

The value of an indicator in the following three ways: (i) by obtaining the value of an input data series, (ii) by applying an indicator to a data series (either an input series or a temporary our output series, or (iii) by performing some computation on the current or past values of one or more available data series. These can be combined in arbitrary ways.

The Bollinger Band indicator above used each of these: It obtains the value of the input data series, applies two indicators (SMA, StandardDeviation) to these values, and then performs a calculation on the current value of these indicators. Other indicators require more complicated scenarios: For example, an indicator may require smoothing of the calculated value (as in the stochastic indicator I:STO, the Fisher indicator I:FISH, or the Volume Oscillator I:VOSC). The stochastic momentum indicator (I:SMI) first obtains values from an input series and applies an indicator to these values, then performs some calculation to produce a temporary series, then applies smoothing to these temporary series, perform some computation on the results, and apply a final smoothing. These more complicated calculations can be constructed in the following manner: Consider the dependencies required by each of the steps in the calculation and begin the calculation at the earliest point in the chain of dependencies.

1. The current or previous value of an indicator can always be obtained as described above.
2. If an indicator application is not the final step, then calculate the value of that indicator starting from the earliest period it satisfies a dependency for subsequent computations up to the current period.
3. If a computation on current or past values of one or more series is not the final step, then calculate all subsequent values in a loop from the earliest period the computation satisfies a dependency for subsequent computations up to the current period.

For example, the following is the calculate method for I:VOSC. The oscillator is calculated by first computing the value of the volume and then smoothing that value with a period given by the first parameter. The smoothing is performed after the computation of the volume measure, and thus the indicator first computes sufficient data values for the smoothing operator in the loop on lines 11-25. After that the smoothing operator is applied (line 27).

```
sub calculate {
    my ($self, $calc, $i) = @_;
    my $vosc_name = $self->get_name(0);
    my $volume_name = $self->get_name(1);
    my $volume = 0;

    return if ($calc->indicators->is_available($vosc_name, $i));
    return if (! $self->check_dependencies($calc, $i));

    my $nb_days = $self->{'args'}->get_arg_values($calc, $i, 1);
    for (my $n = 0; $n < $nb_days; $n++) {
```

---

the dependency period is established correctly only when the corresponding parameter is both constant and positive. Further, unless the dependencies are updated throughout the loop, they reduce to static dependencies (in those situations, if calculate is desired to support volatile dependencies, it is useful to define the volatile dependencies also in calculate_interval to avoid duplicated dependency computation in calculate where static dependencies defined).

```
    next if $calc->indicators->is_available($volume_name, $i - $n);
    if ($calc->prices->at($i - $n)->[$CLOSE] > $calc->prices->at($i - $n)->[$OPEN]) {
       $volume = $calc->prices->at($i - $n)->[$VOLUME];
    }
    if ($calc->prices->at($i - $n)->[$CLOSE] < $calc->prices->at($i - $n)->[$OPEN]) {
       $volume = -$calc->prices->at($i - $n)->[$VOLUME];
    }
    if ($calc->prices->at($i - $n)->[$CLOSE] eq $calc->prices->at($i - $n)->[$OPEN]) {
       $volume = 0;
    }
    $calc->indicators->set($volume_name, $i - $n, $volume);

    }

    $self->{'sma'}->calculate($calc, $i);
    my $vosc_value = $calc->indicators->get($self->{'sma'}->get_name, $i);
    $calc->indicators->set($vosc_name, $i, $vosc_value);
}
```

The transformation to the calculate_interval method is similar to as described in Section 5.2, with the exception that the bounds of any loop used in calculate will have to take the required data history in account. For an example of a more complex indicator as well as for the transformation of the calculate_interval method see the Stochastic Momentum Indicator I:SMI.

9 Documentation

Adequate documentation in pod format should be provided for each indicator.