Investment Idiocy

Friday, 24 October 2014

The worlds simplest execution algo

As you will know I run a fully automated systematic trading system. As its fully automated, due to my extreme laziness, all the trades are put into the market completely automatically.

When I first started running my system I kept the execution process extremely simple:

- Check that the best bid (if selling) or best offer (if buying) was large enough to absorb my order
- Submit a market order

Yes, what a loser. Since I am trading futures, and my broker only has fancy orders for equities, this seemed the easiest option.

I then compounded my misery by creating a nice daily report to tell me how much each trade had cost me. Sure enough most of the time I was paying half the inside spread (the difference between the mid price, and the bid or offer).

After a couple of months of this, and getting fed up with seeing this report add up my losses from trading every day, I decided to bite the bullet and do it properly.

Creating cool execution algorithms (algos) isn't my area of deep expertise, so I had to work from first principles. I also don't have much experience of writing very complicated fast event driven code, and I write in a slowish high level language (python). Finally my orders aren't very large, so there is no need to break them up into smaller slices and track each slice. All this points towards a simple algo being sufficient.

Only one more thing to consider; I get charged for modifying orders. It isn't a big cost, and its worth much less than the saving from smarter execution, but it still means that creating an algo that modifies orders an excessive number of times where this is not necessary probably isn't worth the extra work or cost.

Finally I can't modify a limit order and turn it into a market order. I would have to cancel the order and submit a new one.

What does it do?

A good human trader, wanting to execute a smallish buy order and not worrying about game playing or spoofing etc, will probably do something like this:

- Submit a limit order, on the same side of the spread they want to trade, joining the current level. So if we are buying we'd submit a buy at the current best bid level. In the jargon this is **passive** behaviour, waiting for the market to come to us.
- In an ideal world this initial order would get executed.We'll have gained half the spread in negative execution cost (comparing the mid versus the best bid).
- If:
  - the order isn't being executed after several minutes,
  - or there are signs the market is about to move against them, and rally
  - or the market has already moved up against them
- ... then the smart trader would cut their losses and modify their order to pay up and cross the spread. This is **aggressive** behaviour.
- The new modified aggressive order would be a buy at the current best offer. In theory this would then be executed, costing half the spread (which if the market has already moved against us, would be more than if we'd just submitted a market order initially).
- If we're too slow and the market continues to move against us, keep modifying the order to stay on the new best offer, until we're all done

Although that's it in a nutshell there are still a few bells and whistles in getting an algo like this to work, and in such a way that it can deal robustly with anything that gets thrown at it. Below is the detail of the algo. Although this is shown as python code, its not executable since I haven't included many of the relevant subroutines. However it should give you enough of an idea to code something similar up yourself.

Pre trade

It's somewhat dangerous dropping an algo trade into the mix if the market isn't liquid enough; this routine checks that.

Pages

- Home
- About me
- Legal Disclaimer (please don't sue me)
- Useful links
- Books to read
- Personal investment start here
- Systematic trading blog

Tags

Systematic Trading (18) Technology (13) Interactive Brokers (10) Python (10) Investment idiocy (6) Behavioural finance (5) Finance industry economics (3) Hedge funds (3) Novice investors (3) Portfolio optimization (3) sqllite (3) Execution (2) Git (2) High frequency trading (2) Politics (2) risk management (2) ETF (1) Financial industry pay (1) LOBO (1) Systems building (1) banks (1) housing associations (1)

Blog Archive

- ► 2015 (11)
- ▼ 2014 (21)
  - ► December (2)
  - ► November (1)
  - ▼ October (2)
    - Using sqllite3 to store static and time series dat...
    - The worlds simplest execution algo
  - ► September (3)
  - ► August (1)
  - ► June (3)
  - ► May (4)
  - ► April (3)
  - ► March (2)
- ► 2013 (1)

Share It

Share this on Facebook
Tweet this
View stats
(NEW) Appointment gadget >>

+1 0

Subscribe To

Posts
Comments

Google+ Badge

```
pretrademultiplier=4.0
def EasyAlgo_pretrade(ctrade, contract, dbtype,  tws):
    """
    Function easy algo runs before getting a new order

    ctrade: The proposed trade, an signed integer
    contract: object indicating what we are trading
    dbtype, tws: handles for which database and tws API server we are
dealing with here.


    Returns integer indicating size I am happy with

    Zero means market can't support order



    """


    ## Get market data (a list containing inside spread and size)

    bookdata=get_market_data(dbtype, tws, contract, snapshot=False,
maxstaleseconds=5, maxwaitseconds=5)

    ## None means the API is not running or the market is closed :-(

    if bookdata is None:
        return (0, bookdata)


    ## Check the market is liquid; the spread and the size have to be
within certain limits. We use a multiplier because we are less
discerning with limit orders - a wide spread could work in our favour!

    market_liquid=check_is_market_liquid(bookdata, contract.code,
multiplier=pretrademultiplier)

    if not market_liquid:
        return (0, bookdata)

    ## If the market is liquid, but maybe the order is large compared
to the size on the inside spread, we can cut it down to fit the order
book.

    cutctrade=cut_down_trade_to_order_book(bookdata,  ctrade,
multiplier=pretrademultiplier)

    return (cutctrade, bookdata)
```

New order

Not just one of the best bands in the eighties, also the routine you call when a new order request is issued by the upstream code.

```
MAX_DELAY=0.03

def EasyAlgo_new_order(order, tws, dbtype, use_orderid, bookdata):

  """
    Function easy algo runs on getting a new order

    Args:
    order - object of my order type containing the required trade
    tws - connection object to tws API for interactive brokers
    dbtype - database we are accessing
    use_orderid- orderid
    bookdata- list containing best bid and offer, and relevant sizes


    """

    ## The s, state, variable is used to ensure that log messages and
diagnostics get saved right. Don't worry too much about this



    log=logger()
    diag=diagnostic(dbtype, system="algo",  system3=str(order.orderid))
    s=state_from_sdict(order.orderid, diag, log)

    ## From the order book, and the trade, get the price we would pay
if aggressive (sideprice) and the price we pay if we get passive
```

```
(offsideprice)


    (sideprice, offsideprice)=get_price_sides(bookdata,
order.submit_trade)

    if np.isnan(offsideprice) or offsideprice==0:
        log.warning("No offside / limit price in market data so can't
issue the order")
        return None

    if np.isnan(sideprice) or sideprice==0:
        log.warning("No sideprice in market data so dangerous to issue
the order")
        return None

    ## The order object contains the price recorded at the time the
order was generated; check to see if a large move since then (should be
less than a second, so unlikely unless market data corrupt)









    if not np.isnan(order.submit_price):
        delay=abs((offsideprice/order.submit_price) - 1.0)
        if delay>MAX_DELAY:
            log.warning("Large move since submission - not trading a
limit order on that")
            return None

    ## We're happy with the order book, so set the limit price to the
'offside' - best offer if selling, best bid if buying



    limitprice=offsideprice

    ## We change the order so its now a limit order with the right
price

    order.modify(lmtPrice = limitprice)
    order.modify(orderType="LMT")

    ## Need to translate from my object space to the API's native
objects









    iborder=from_myorder_to_IBorder(order)
    contract=Contract(code=order.code, contractid=order.contractid)
    ibcontract=make_IB_contract(contract)

    ## diagnostic stuff
    ## its important to save this so we can track what happened if
orders go squiffy (a technical term)


    s.update(dict(limit_price=limitprice, offside_price=offsideprice,
side_price=sideprice,
                  message="StartingPassive", Mode="Passive"))
    timenow=datetime.datetime.now()


    ##  The algo memory table is used to store state information for
the algo. Key thing here is the Mode which is PASSIVE initially!


    am=algo_memory_table(dbtype)
    am.update_value(order.orderid, "Limit", limitprice)
    am.update_value(order.orderid, "ValidSidePrice", sideprice)
    am.update_value(order.orderid, "ValidOffSidePrice", offsideprice)
    am.update_value(order.orderid, "Trade", order.submit_trade)
    am.update_value(order.orderid, "Started", date_as_float(timenow))
```

```
    am.update_value(order.orderid, "Mode", "Passive")
    am.update_value(order.orderid, "LastNotice",
date_as_float(timenow))

    am.close()


     ## Place the order
    tws.placeOrder(
            use_orderid,                                  # orderId,
            ibcontract,                                 # contract,
            iborder                                     # order
        )

    ## Return the order upstream, so it can be saved in databases etc.
Note if this routine terminates early it returns a None; so the
upstream routine knows no order was placed.




    return order
```

Action on tick

A **tick** comes from the API when any part of the inside order book is updated (best bid or offer, or relevant size).

Within the tws server code I have a routine that keeps marketdata (a list with best bid and offer, and relevant sizes) up to date as ticks arrive, and then calls the relevant routine.

What does this set of functions do?

- If we are in a passive state (the initial state, remember!)
    - ... and more than five minutes has elapsed, **change to aggressive**
    - if buying and the current best bid has moved up from where it started (an adverse price movement), **change to aggressive**
    - if selling, and the current best offer has moved down from where it started (also adverse)
    - If there is an unfavourable order imbalance (eg five times as many people selling than buying on the inside spread if we're also selling), **change to aggressive**.
- If we are in an aggressive state
    - ... and more than ten minutes has elapsed, **cancel the order**.
    - if buying and the current best offer has moved up from where it was last (a further adverse price movement), then **update our limit** to the new best offer (chase the market up).
    - if selling and the current best bid has moved down from where it was last (a further adverse price movement), then **update our limit** to the new best offer

```
passivetimelimit=5*60 ## max five minutes
totaltimelimit=10*60 ## max another five minute aggressive
maximbalance=5.0 ## amount of imbalance we can copy with

def EasyAlgo_on_tick(dbtype, orderid, marketdata, tws, contract):
    """
    Function easy algo runs on getting a tick

    Args:
    dbtype, tws: handles for database and tws API
    orderid: the orderid that is associated with a tick
    marketdata: summary of the state of current inside spread
    contract: what we are actually trading



    """

    ## diagnostic code
    log=logger()
    diag=diagnostic(dbtype, system="algo",  system3=str(int(orderid)))
    s=state_from_sdict(orderid, diag, log)


    ## Pull out everything we currently know about this order

    am=algo_memory_table(dbtype)
    trade=am.read_value(orderid, "Trade")
    current_limit=am.read_value(orderid, "Limit")
```

```
    Started=am.read_value(orderid, "Started")
    Mode=am.read_value(orderid, "Mode")
    lastsideprice=am.read_value(orderid, "ValidSidePrice")
    lastoffsideprice=am.read_value(orderid, "ValidOffSidePrice")
    LastNotice=am.read_value(orderid, "LastNotice")

    ## Can't find this order in our state database!


    if Mode is None or Started is None or current_limit is None or
trade is None or LastNotice is None:
        log.critical("Can't get algo memory values for orderid %d
CANCELLING" % orderid)
        FinishOrder(dbtype, orderid, marketdata, tws, contract)

    Started=float_as_date(Started)
    LastNotice=float_as_date(LastNotice)
    timenow=datetime.datetime.now()


    ## If a buy, get the best offer (sideprice) and best bid
(offsideprice)
    ## If a sell, get the best bid (sideprice) and best offer
(offsideprice)
    (sideprice, offsideprice)=get_price_sides(marketdata, trade)

    s.update(dict(limit_price=current_limit,
offside_price=offsideprice, side_price=sideprice,
                  Mode=Mode))

    ## Work out how long we've been trading, and the time since we last
'noticed' the time


    time_trading=(timenow - Started).total_seconds()
    time_since_last=(timenow - LastNotice).seconds

    ## A minute has elapsed since we


    if time_since_last>60:
        s.update(dict(message="One minute since last noticed now %s,
total time %d seconds - waiting %d %s %s" % (str(timenow),
time_trading, orderid, contract.code, contract.contractid)))
        am.update_value(orderid, "LastNotice", date_as_float(timenow))

    ## We've run out of time - cancel any remaining order

    if time_trading>totaltimelimit:
        s.update(dict(message="Out of time cancelling for %d %s %s" %
(orderid, contract.code, contract.contractid)))
        FinishOrder(dbtype, orderid, marketdata, tws, contract)
        return -1

    if not np.isnan(sideprice) and sideprice<>lastsideprice:
        am.update_value(orderid, "ValidSidePrice", sideprice)

    if not np.isnan(offsideprice) and offsideprice<>lastoffsideprice:
        am.update_value(orderid, "ValidOffSidePrice", offsideprice)

    am.close()


    if Mode=="Passive":


        ## Out of time (5 minutes) for passive behaviour: panic


        if time_trading>passivetimelimit:
            s.update(dict(message="Out of time moving to aggressive for
%d %s %s" % (orderid, contract.code, contract.contractid)))

            SwitchToAggresive(dbtype, orderid, marketdata, tws,
contract, trade)
            return -1


        if np.isnan(offsideprice):
            s.update(dict(message="NAN offside price in passive mode -
waiting %d %s %s" % (orderid, contract.code, contract.contractid)))
            return -5

        if trade>0:
```

```
            ## Buying
            if offsideprice>current_limit:
                ## Since we have put in our limit the price has moved
up. We are no longer competitive

                s.update(dict(message="Adverse price move moving to
aggressive for %d %s %s" % (orderid, contract.code,
contract.contractid)))

                SwitchToAggresive(dbtype, orderid, marketdata, tws,
contract, trade)

                return -1
        elif trade<0:
            ## Selling
            if offsideprice<current_limit:
                ## Since we have put in our limit the price has moved
down. We are no longer competitive

                s.update(dict(message="Adverse price move moving to
aggressive for %d %s %s" % (orderid, contract.code,
contract.contractid)))

                SwitchToAggresive(dbtype, orderid, marketdata, tws,
contract, trade)
                return -1

        ## Detect Imbalance (bid size/ask size if we are buying; ask
size/bid size if we are selling)



        balancestat=order_imbalance(marketdata, trade)

        if balancestat>maximbalance:
                s.update(dict(message="Order book imbalance of %f
developed compared to %f, switching to aggressive for %d %s %s" %
(balancestat , maximbalance, orderid, contract.code,
contract.contractid)))

                SwitchToAggresive(dbtype, orderid, marketdata, tws,
contract, trade)
                return -1



    elif Mode=="Aggressive":

        if np.isnan(sideprice):
            s.update(dict(message="NAN side price in aggressive mode -
waiting %d %s %s" % (orderid, contract.code, contract.contractid)))
            return -5


        if trade>0:
            ## Buying
            if sideprice>current_limit:
                ## Since we have put in our limit the price has moved
up further. Keep up!

                s.update(dict(message="Adverse price move in aggressive
mode for %d %s %s" % (orderid, contract.code, contract.contractid)))
                SwitchToAggresive(dbtype, orderid, marketdata, tws,
contract, trade)


                return -1
        elif trade<0:
            ## Selling
            if sideprice<current_limit:
                ## Since we have put in our limit the price has moved
down. Keep up!

                s.update(dict(message="Adverse price move in aggressive
mode for %d %s %s" % (orderid, contract.code, contract.contractid)))

                SwitchToAggresive(dbtype, orderid, marketdata, tws,
contract, trade)
                return -1

    elif Mode=="Finished":
        ## do nothing, still have tick for some reason
        pass
```

```
    else:
        msg="Mode %s not known for order %d" % (Mode, orderid)
        s.update(dict(message=msg))

        log=logger()
        log.critical(msg)
        raise Exception(msg)


    s.update(dict(message="tick no action %d %s %s" % (orderid,
contract.code, contract.contractid)))


    diag.close()



    return 0

def SwitchToAggresive(dbtype, orderid, marketdata, tws, contract,
trade):
    """
    What to do... if we want to eithier change our current order to an
aggressive limit order, or move an order is already aggressive limit
price

    """
    ## diagnostics...
    log=logger()
    diag=diagnostic(dbtype, system="algo",  system3=str(int(orderid)))
    s=state_from_sdict(orderid, diag, log)


    if tws is None:
        log.info("Switch to aggressive didn't get a tws... can't do
anything in orderid %d" % orderid)
        return -1

    ## Get the last valid side price (relevant price if crossing the
spread) as this will be our new limit order


    am=algo_memory_table(dbtype)

    sideprice=am.read_value(orderid, "ValidSidePrice")

    ordertable=order_table(dbtype)
    order=ordertable.read_order_for_orderid(orderid)
    ordertable.close()

    if np.isnan(sideprice):
        s.update(dict(message="To Aggressive: Can't change limit for %d
as got nan - will try again" % orderid))
        return -1

    ## updating the order

    newlimit=sideprice


    order.modify(lmtPrice = newlimit)
    order.modify(orderType="LMT")


    iborder=from_myorder_to_IBorder(order)
    ibcontract=make_IB_contract(contract)


    am.update_value(order.orderid, "Limit", newlimit)
    am.update_value(order.orderid, "Mode", "Aggressive")
    am.close()


     # Update the order
    tws.placeOrder(
            orderid,                                    # orderId,
            ibcontract,                                   # contract,
            iborder                                       # order
        )

    s.update(dict(limit_price=newlimit, side_price=sideprice,
                  message="NowAggressive", Mode="Aggresive"))
```

```
    return 0



def FinishOrder(dbtype, orderid, marketdata, tws, contract):
    """
    Algo hasn't worked, lets cancel this order
    """    diag=diagnostic(dbtype, system="algo",
system3=str(int(orderid)))

    s=state_from_sdict(orderid, diag, log)        log=logger()

    if tws is None:
        log.info("Finish order didn't get a tws... can't do anything in
orderid %d" % orderid)
        return -1


    log=logger()
    ordertable=order_table(dbtype)

    order=ordertable.read_order_for_orderid(orderid)

    log.info("Trying to cancel %d because easy algo failure" % orderid)
    tws.cancelOrder(int(order.brokerorderid))

    order.modify(cancelled=True)
    ordertable.update_order(order)

    do_order_completed(dbtype, order)

    EasyAlgo_on_complete(dbtype, order, tws)


    s.update(dict(message="NowCancelling", Mode="Finished"))

    am=algo_memory_table(dbtype)
    am.update_value(order.orderid, "Mode", "Finished")
    am.close()

    return -1
```

Partial or complete fill

Blimey this has actually worked, we've actually got a fill...

```
def EasyAlgo_on_partial(dbtype, order, tws):
    diag=diagnostic(dbtype, system="algo",
system3=str(int(order.orderid)))

    diag.w(order.filledtrade, system2="filled")
    diag.w(order.filledprice, system2="fillprice")

    return 0




def EasyAlgo_on_complete(dbtype, order_filled, tws):
    """
    Function Easy algo runs on completion of trade
    """


    diag=diagnostic(dbtype, system="algo",
system3=str(int(order_filled.orderid)))

    diag.w("Finished", system2="Mode")
    diag.w(order_filled.filledtrade, system2="filled")
    diag.w(order_filled.filledprice, system2="fillprice")

    am=algo_memory_table(dbtype)
    am.update_value(order_filled.orderid, "Mode", "Finished")
    am.close()

    return 0
```

And we're done

That's it. Its not perfect and it would be very easy to write high frequency code that would game this kind of strategy. However the proof is in the proverbial traditional English dessert, and my execution costs have reduced by approximately 80% from when I was doing market orders, i.e. I am paying an average of 1/10 of the spread. So it's definitely an improvement, and well worth the day or so it took me to code it up and test it.

Posted by Rob Carver at 11:08

Labels: Execution, High frequency trading, Interactive Brokers, Python, Systematic Trading, Technology

No comments:

Post a Comment

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)