9/14/2016
Proposal: Implied Pricing Engine

PROJECT IDENTIFICATION:

IMPLIED PRICING ENGINE: ENERGY COMPLEX

SUBMITTED TO:

(Owner)

PROPOSAL:

We will construct an implied pricing engine for the following futures markets:

CME Crude + Products	: {CL, BZ, RB, HO, QM}
CME Natural Gas	: {NG}
ICE Crude + Products	: {B, I, T, GO}
ICE Natural Gas	: {H}

By an implied engine we mean a real-time system for publishing inside implied bid, inside ask prices and sizes for a specified number of prompt outright contracts in the above complexes, based on a simpler stream of user-entered ("non-implied") quotes. The implied engine consumes a feed (stream, multicast feed, etc.) of user-entered ("non-implied") quote prices and sizes in a preset universe of legs, calendar spreads, strategies, and produces as output an improved set of quotes implied from the user-entered quotes.

In general, marketable orders are price-improved to the best implied quote, which will often be more favorable than the best user-entered quote. In many cases the exchange does not publish the best implied quote, and this price-improvement is "invisible" to the trader. Knowing true implied inside quotes can provide invaluable information for price and fair value calculations in the cases in which non-implied markets are wide or even non-published. In particular, these uses are common:

- Price implication with lower latency than exchange-published implied quotes;
- Generation of fair value price, size for markets in which no implied prices are published;
- Generation of improved fair value in all markets from broader inputs to implied engine;
- Improved and faster signal generation.

DETAILS:

Denoting by *IE* the implied engine we have

$$IE(L, S, T) = (l_1, l_2, \ldots, l_n)$$

where

L = inside non-implied quote price, size for leg products
S = inside non-implied quote price, size for calendar spread products
T = inside non-implied quote price, size for strategy products (strips, condors)

and

l_i = inside quote price, size for ith prompt outright.

The algorithm we use is fast, efficient, and complete. Each outright calculation on the right side above is done separately and can be calculated on its own execution thread, asynchronously. There is some similarity across these outright calculations that can be used to further enhance efficiency.

For more precise timing, we can conceptually divide the algorithm into 3 steps: update, calculate, publish. The calculate step is the calculation of the vector-valued *IE* function above. The update and publish steps are highly platform-dependent and involve consumption and publication of quotes.

The calculate step calculates n leg bid prices and sizes, based on refreshed data from the update step. For a typical problem, the number of outright contracts is $n = 12$, and the number of calendar spread contracts is 12*11/2 = 66, we call this the *n-leg* problem. The n-leg problem runs in $O(n^2)$ time.

CPU time for each calculate step for an n-leg problem is given below. For the 12-leg problem, the calculate step comes in at under 10 micros. We run 1e6 trials, removing the best and worst times. [Process run on Intel x86_64 i5 single-core machine[1], all times are in microseconds]:

---

[1] Process run on virtualized commodity linux (Ubuntu) 2.3GHz server:

```
      *-cpu
        product: Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz

      Architecture:          x86_64
      CPU op-mode(s):        32-bit, 64-bit
      Byte Order:            Little Endian
      CPU(s):                1
      On-line CPU(s) list:   0
      Thread(s) per core:    1
      Core(s) per socket:    1
      Socket(s):             1
      CPU MHz:               2194.924
      Hypervisor vendor:     KVM
      Virtualization type:   full
      L1d cache:             32K
      L1i cache:             32K
      L2 cache:              256K
      L3 cache:              3072K
```

#legs	average	min	max	stdev	#trials
0	0.000000	0	0	0.000000	999998
1	0.000000	0	0	0.000000	999998
2	1.450224	0	16335	2.157503	999998
3	1.744384	1	12412	3.437309	999998
4	2.052856	1	8510	3.748844	999998
5	2.687152	2	11847	3.882017	999998
6	3.173562	2	12475	3.407387	999998
7	3.325170	2	3444	6.540734	999998
8	4.148126	3	4086	7.450844	999998
9	4.994720	4	3519	8.256040	999998
10	5.668034	4	4799	8.603363	999998
11	6.691828	5	11881	17.564463	999998
12	7.913124	6	10121	10.412915	999998
13	9.453345	7	13721	21.430757	999998
14	9.586401	8	7875	24.379706	999998
15	10.084670	8	8246	16.196867	999998
16	11.625233	10	4032	12.451166	999998
17	12.661665	11	12258	19.556137	999998
18	13.993410	12	9616	18.107115	999998
19	16.782749	13	9092	29.949022	999998
20	19.950690	14	59956	62.886111	999998
21	24.439750	16	66186	43.988407	999998
22	21.510550	17	11464	42.936446	999998
23	22.624264	18	15784	41.984122	999998
24	24.506462	19	12628	47.052661	999998

The algorithm is customizable, allowing for tradeoffs between completeness of derivations and latency, and different handling of implied prices by exchange. The Natural Gas ("NG") markets are handled differently, due to different exchange implied rules and products offered, specific seasonality, etc.

The publish step supports a number of protocols for distribution of quotes:

- In-process;
- Shared memory/mapped memory;
- FIFO, sockets;
- Etc.

The most efficient solution depends on latency requirements and existing architecture. The process is quite lightweight and in-process instantiation is probably possible in most cases, allowing for customization per application.

TIMELINE:

The work will be completed in approximately 2 months with milestones below.

Milestones:

Python prototype (**optional**) : 3-4 weeks
C++ single threaded engine : 1 months

C++ multi-threaded engine : 2 months

Required libraries/technology:

Above language support [C++11/14; gcc complier >= 4.9]
Optional: Boost C++ libraries (BGL (for tie-outs), aggregates, asio, etc.)
Optional: Eigen C++ linear algebra library

BASE BID

____ $ ______________