Holo Fuel Reserve Account Design and Modelling

Perry Kundert

September 29, 2018

Contents

Abstract

The Holo Reserves are a primary method of purchasing Holo Fuel for Hosting services, and is available for Hosts only to redeem Holo Fuel for cash in various currencies. Others Holo Fuel account holders may buy via the Reserves, and and buy/sell via other exchanges, but the reserve's LIFO tranches are available to Holo Fuel accounts associated with known Holo Hosts.

Holo Fuel credits redeemable for Hosting are purchased at a certain cost, and later redeemed for that cost by Hosts after these services are delivered. Therefore, the purchase price must be palatable for redemption by at least some Hosts. Of course, Holo dApp Owners are free to purchase Holo Fuel on exchanges at lower prices, and Hosts can cash out via exchanges at market prices.

The Holo Fuel / currency sale price is also controlled to adjust net currency in/outflows, both to adjust for changes in relative currency valuation, and to balance the proportion of Reserves in each currency to match the desired Host cash-out currencies.

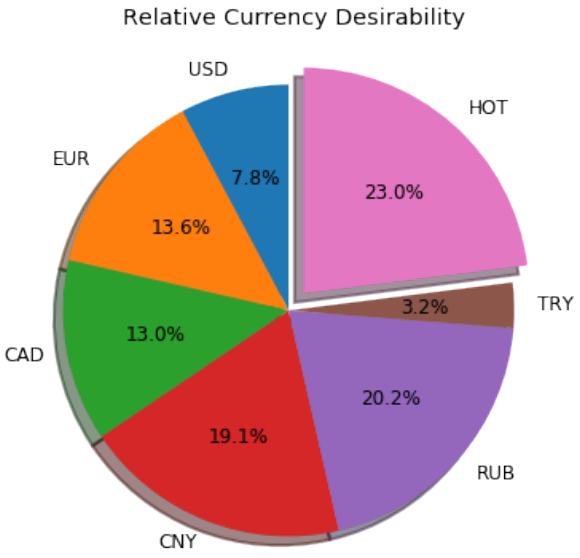
1 Host Currency Preference and Minimum Cash-Out Value

Each Host sets their preference for redemption currencies, and an exchange rate minimum for one of them; the others will be deduced, because each of their exchange rates to Holo Fuel are known.

Lets select a few currencies, with varying levels of desirability for Holo Hosts to cash out with:

```
%matplotlib inline
%config InlineBackend.figure_format = 'retina'

from __future__ import absolute_import, print_function, division
try:
    from future_builtins import zip, map # Use Python 3 "lazy" zip, map
except ImportError:
       pass

import sys
import matplotlib
```

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"]  = (6,3)
plt.rcParams["font.size"]       = 6
import numpy as np
from sklearn import linear_model
import collections
import math
import random
import json
import bisect
import logging
import time
logging.basicConfig( level=logging.DEBUG )

from holofuel.model import trading
from holofuel.model.reserve_lifo import reserve, reserve_issuing

def rnd_std_dst( sigma, mean=0, minimum=None, maximum=None ):
    """Random values with mean, in a standard distribution w/ sigma, clipped to given minimum/maximum."""
    val             = sigma * np.random.randn() + mean
    return val if minimum is None and maximum is None else np.clip( val, a_min=minimum, a_max=maximum )
#
# Compute target currencies, with random distribution of desirabilites (totalling 1.0)
#
currencies = [ 'USD', 'EUR', 'CAD', 'CNY', 'RUB', 'TRY', 'HOT' ]
desi_mean,desi_sigma = 1,    .66 # desirability weighting of various currencies
desirability = [ rnd_std_dst( mean=desi_mean, sigma=desi_sigma, minimum=3*len(currencies)/100 ) # ~3% minimum
                                for _ in range( len( currencies )) ]
desirability        /= np.sum( desirability ) # normalize sum of probabilites to 1.0

explode                 = [ .1 if c == 'HOT' else 0 for c in currencies ]
#with plt.xkcd():
fig1,ax1                = plt.subplots()
ax1.pie( desirability, explode=explode, labels=currencies, autopct='%.1f%%', shadow=True, startangle=90 )
ax1.axis( 'equal' ) # Equal aspect ratio ensures that pie is drawn as a circle.
plt.title( "Relative Currency Desirability" )
plt.show()

[ [ 'Currency', 'Desirability' ], None ] \
+ [ [ curr, "%.4f" % ( desi ) ]
    for curr,desi in zip( currencies, desirability ) ]
```



Obtain some actual exchange rates for a test period:

```
#
# Load some exchange rates.  Convert observations:
#
#  [ { "d": "2017-01-03", "FXAUDCAD": { "v": 0.9702 }, ...},
#    { "d": "2017-01-04" ... }, ... ]
# to rates:
#  { "2017-01-03": { "USD/CAD": 1.29, "CAD/USD": 0.775, "USD/EUR": ... }, "2017-01-04": { ... } }
#
class Rates( object ):
    def __init__( self, fx_rates ):
        rates_data = json.loads( open( fx_rates ).read() )
self.rates = {}
        for rec in rates_data['observations']:
            d = self.rates[rec["d"]] \
= {}
            for c1 in currencies:
                if c1 != 'CAD' and 'FX'+c1+'CAD' not in rec:
                    continue
                for c2 in set( currencies ) - set([c1]):
                    if c2 != 'CAD' and 'FX'+c2+'CAD' not in rec:
                        continue
                    tocad       = 1 if c1 == 'CAD' else rec['FX'+c1+'CAD']['v'] # eg. FXUSDCAD: 1.3
                    frcad       = 1 if c2 == 'CAD' else rec['FX'+c2+'CAD']['v'] # eg. FXEURCAD: 1.47
                    d[c1+'/'+c2]= tocad / frcad                                 # ==>  USD/EUR: 0.884
self.days = sorted( self.rates.keys() )
        #print( json.dumps( rates[self.days[0]], indent=4 ))

    def exchange( self, day, fr, to ): # "2017-01-17", 'EUR', 'USD'
        """Finds the exchange rate for a day near to the given YYYY-MM-DD"""
        if fr == to:
            return 1.0
        i = bisect.bisect_left( self.days, day )
        if i >= len( self.days ):
            i        -= 1
        #print( json.dumps( self.rates[self.days[i]], indent=4 ))
        try:
            return self.rates[self.days[i]][fr+'/'+to]
        except:
            return math.nan

rates = Rates( "static/data/FX_RATES_DAILY-sd-2017-01-03.json" )

[ [ 'Day', 'From', 'To', 'Exchange' ],  None ] \
+ [ [ d, fr, to, "%5.3f" % ( rates.exchange( d, fr, to ) ) ]
    for d in [ '2017-01-04', '2018-09-20' ]
    for fr,to in [['EUR','USD'],['USD','EUR'],['CAD','EUR'],['CAD','USD']] ]
```

Day	From	To	Exchange
2017-01-04	EUR	USD	1.046
2017-01-04	USD	EUR	0.956
2017-01-04	CAD	EUR	0.718
2017-01-04	CAD	USD	0.751
2018-09-20	EUR	USD	1.176
2018-09-20	USD	EUR	0.850
2018-09-20	CAD	EUR	0.659
2018-09-20	CAD	USD	0.775

Each Host can specify 0 or more preferred redemption currencies and rates. Only 1 target Fiat currency rate is allowed, because the exchange rates between currencies are deduced by the inflow/outflow equilibrium through the Reserve accounts. Until HOT floats, no exchange rate is supported; it is fixed at 1 HOT == 1 Holo Fuel.

```
class Host( trading.agent ):
    def __init__( self, redemption,
    quanta = 1 * trading.hour,
    **kwds ):
        """Support 0 or 1 specified exchange rate, deducing all others.  Filter out currencies not desired
(target rate is Falsey).

redemption: {
    "CAD": .50,
    "USD": True,
    "CNY": False, # Filtered out
    "EUR": True,
    "HOT": True
}

        """
super( Host, self ).__init__( quanta=quanta, **kwds )
        self.redemption = { c: redemption[c]
                            for c in redemption
    if redemption[c] }

assert 0 <= sum( type( r ) in (int,float) for c,r in self.redemption.items() ) <= 1, \
    "A maximum of one target redemption is allowed; %s supplied" % (
        ', '.join( '%s: %f' % ( c, r )
   for c,r in self.redemption.items()
                           if type( r ) in (int,float) ))

    def redemption_rate( self, day, curr ):
        """Computes the target redemption rate in the specified currency, or Falsey (0/None/False) if not
desired.  If a currency is desired, but no minimum cash-out rate is specified (indicating
that "market" rates are desired), returns True."""
if curr not in self.redemption:
    return False
if curr == 'HOT':
    return 1.0
# find a specified currency w/ a minimum rate specified
for curr_exch,rate_min in self.redemption.items():
    if type( rate_min ) is not bool: # could be int,float, a numpy type
        # An exchange rate minimum was specified!  Compute the target currency's rate vs. that
        # rate using that day's (in "YYYY-MM-DD") exchange rate.  For example, if the exch ==
        # 'USD' and the target is (say) rate == 0.50, and we're asking for 'CAD' and the day's
        # exchange rate is 1.29, we'll return 0.50 * 1.20 == 0.645
        rate_exch = rates.exchange( day, fr=curr_exch, to=curr )
        rate_redeem = rate_min * rate_exch
        if math.isnan( rate_exch ) or math.isnan( rate_redeem ):
            print( "For %s on %s, minimum: %s, %s/%s exchange rate: %s" % (
                curr, day, rate_min, curr_exch, curr, rate_exch ))
        return rate_redeem
# No target currency w/ minimum rate: "market" rates are desired
return True
#
# Compute a number of Host w/ varying numbers of desired currencies and target exchange rates
#
host_count = 100
rate_mean,rate_sigma = 0.50, 0.25 # variance in minimum rates of exchange (CAD)
curr_mean,curr_sigma           = 3,    2       # number of currencies selected
hosts = []

for h in range( host_count ):
    # select between 0 and all currencies as candidates for redemption, with the random choice of each
    # currency weighted by its relative desirability
    curr_cnt                    = max( 0, min( len( currencies ), int( rnd_std_dst( mean=curr_mean, sigma=curr_sigma ))))
    redemption         = { curr: True
                                for curr in np.random.choice( a=currencies, size=curr_cnt, replace=False, p=desirability ) }
    # Choose an exchange rate for one Fiat currency (in CAD$ terms)
    fiat                        = set( redemption ) - set( [ 'HOT' ] )
    rate_num = 1
```

```
    rate_cad = rnd_std_dst( mean=rate_mean, sigma=rate_sigma, minimum=0 ) # may be 0 ==> no desired rate ("market")

    if fiat and rate_cad:
        for curr in np.random.choice( a=list( fiat ), size=min( rate_num, len( fiat )), replace=False ):
            redemption[curr] = rate_cad * rates.exchange( rates.days[0], 'CAD', curr )
    hosts.append( Host( identity = "Host{}".format( h ), redemption = redemption ))
    #print( "CAD exch: %6.4f, target Fiat %r == %r %s" % (
    #    rate_cad, fiat, hosts[-1].redemption, "" if rate_cad else "==> market rates" ))


#
# See if we can recover a median, mean and std.dev. for each cash-out currency.
#
def currency_statistics( hosts, day, curr ):
    """For a currency 'curr' on a day, compute the Hosts desiring that currency, and the statistical
    distribution of their cash-out minimum.

    """
    curr_stats = {}
    # Ignore bad, Falsey (False/0 == not desired), or -'ve (invalid) exchange rates
    sel = []
    for h in hosts:
        r = h.redemption_rate( day, curr )
if not math.isnan( r ) and r and r > 0:
    sel.append( r )
    if not sel:
        return curr_stats # leave empty (Falsey) if no cash-out currencies selected
    curr_stats['selected'] = sel # contains desired exch. rate, or True (for "market")
    curr_stats['minimums'] = sorted( x for x in sel if type( x ) is not bool )
    mins_cnt = len( curr_stats['minimums'] )
    curr_stats['median'] = curr_stats['minimums'][mins_cnt // 2] if mins_cnt else None
    curr_stats['mean'] = np.mean( curr_stats['minimums'] ) if mins_cnt else None
    curr_stats['sd'] = np.std( curr_stats['minimums'] ) if mins_cnt else None
    return curr_stats

stats = {}
for curr in currencies:
    stats[curr] = currency_statistics( hosts, rates.days[0], curr )
    #print( curr + ': ' + ', '.join( "%7.4f" % r for r in stats[curr]['minimums'] ))

[ [ '', '', '%r/ea +/-%r' % ( curr_mean, curr_sigma ), 'Rate' ],
  [ 'Currency', '% Weight', '% Selected', 'Mean', 'Median', 'Std.Dev' ],
  None ] \
+ [ [ curr,
      "%.1f" % ( desi * 100 ),
      len( stats[curr]['minimums'] ) * 100.0 / host_count,
      "%.4f" % ( stats[curr]['mean'] or 0 ),
      "%.4f" % ( stats[curr]['median'] or 0 ),
      "%.4f" % ( stats[curr]['sd'] or 0 ) ]
    for curr,desi in zip( currencies, desirability ) ]
```

Currency	% Weight	3/ea +/-2 % Selected	Rate Mean	Median	Std.Dev
USD	7.8	35.0	0.3622	0.3760	0.1313
EUR	13.6	34.0	0.3547	0.3737	0.1287
CAD	13.0	32.0	0.4590	0.3941	0.2033
CNY	19.1	46.0	2.4540	2.3383	1.0203
RUB	20.2	49.0	22.7920	22.5910	8.7220
TRY	3.2	16.0	1.2371	1.3616	0.5116
HOT	23.0	52.0	1.0000	1.0000	0.0000

1.1 Host Auto-pilot Pricing

A Host can specify rates to charge for its various computational resources, in Holo Fuel, or it can set "auto-pilot" pricing. The lower the pricing, the higher the expected utilization of the resource vs. the median Host.

Each Host competes for traffic against other Hosts serving the same Holochain hApp. From time to time, the Holo service polls the Hosts capable of serving an hApp, and groups them into tranches of comparable quality based on price. A proportion of the hApp's traffic will be assigned to each tranche; more to lower-priced tranches, less to the more costly.

Thus, over time the Hosts' pricing decisions will be reflected in the average utilization for the resource. This could be computed over days, not hours, to account for cyclical (day/night) shifts in utilization. Or, it could be computed on a shorter cycle such as every 10 minutes, to allow the auto-pilot to be used to adjust utilization more promptly.

To support real-time utilization modulation, for example increasing the price of Network bandwidth to reduce utilization when the owner is using a streaming video services like Netflix. This would also require the Holo system supporting the hApp to poll its Host resources for pricing more rapidly; at the Nyquist rate; 2x the frequency of change of the signal.

1.1.1 Increasing hApp Hosting Prices Attracts New Hosts

As a Host wishing to maximize revenue per unit of Compute, I want to host hApps that pay well. Each Holo hApp knows what its median and average hosting prices has been across all resources, and this information is published.

Hosts will survey the hApps available from time to time, disabling and eventually ejecting low-paying (probably over-provisioned) hApps in favour of higher-paying (possibly under-provisioned) ones. This eventually frees up the storage and other resources used by the old hApp; once the Host is no longer represented in the hApps tranches, it can power down and delete the hApps' resources.

Each hApp uses various resources (eg. Network bandwidth, CPU power, RAM, Storage) at differing rates. One or more hApps will be ejected only if the replacement hApp(s) fill all of the available Host resources more profitably than the old set.

Equilibrium is reached when hApps are provisioned across the Hosting network with all Hosts' resource utilization more or less level (eg. a High CPU Low Storage hApp, next to a Low CPU High Storage hApp), and the median resource cost more or less equal for each hApp, proportional to its average utilization. For example, given two roughly equivalent hApps, one with 100x more client utilization than the other; the Holo Host pricing system should ensure that roughly 100x more Hosts are hosting the hApp, and that the aggregate Hosting costs to the larger hApp owner are about 100x the costs of the lesser hApp.

1.1.2 hApp/Host Auto-pilot Feedback

If an hApp owner is aware of cyclicality or spikes in its utilization (eg. just before launching an advertising campaign), the owner can even pre-allocate increased resources by temporarily increasing its own hApp Holo service auto-pilot pricing to a higher tier. This increases the amount it is willing to pay for hosting, putting it into contention for installation by Hosts with "hi" (premium) auto-pilot pricing. When the spike actually hits, the hApp owner can restore its own pricing auto-pilot to the normal tier, letting regular Holo price-based levelling distribute the hApp appropriately for the new load.

1.2 Modelling Holo hApp/Host Auto-pilot Pricing

The goal of Holo hApp and Host Auto-pilot pricing is to allow both hApp owners and Hosts achieve equilibrium pricing within a budget they can afford.

Holo hApp owners have clients to serve, and require Host resources within a certain budget. Hosts have resources to sell, and want to make the most money by hosting the hApps paying the most for those resources.

1.2.1 Holo hApp Host Tranching

A core tenet of Holochain applications is that their state is stored privately in a local chain, and publicly in an eventually consistent DHT. So, in theory, any "read only" client request accessing public data can be served by any Host. The application using Holochain must be resilient to the eventually consistent nature of the underlying datastore. Much of Holo's activity will, however, be the establishment of Holochain proxy instances, which are capable of storing/updating a local chain on behalf of a (web-based) user (the identity's signing keys are held by the client; communication encryption keys are held by the proxies).

The Hosts to provide these services are chosen pseudorandomly from pools of Hosts of like performance and cost, called tranches. The probability of getting any Host is proportional to its desirability (cheapest highest performing hosts first). In aggregate, the average price paid per request is intended to be near the "median" price/performance; a mix of high/low priced and high/low performing Hosts is used. The tranches are dynamically updated based on analysis of the actual request performance and current Host pricing. The mix of Hosts used to service requests is adjusted dynamically based on the hApp owner's current Hosting cost targets; a hApp currently targeting below-market **discount** Hosting costs will get a mixture of Hosts averaging that lower target cost (ie. less **premium** priced Hosts, more **discount** and **market** priced Hosts.)

Each set of tranches is an N-dimensional grid of buckets, with axes denominated in the various ratings for the feature. The 'holo' commodity is simple; a single axis based on transaction response time, as computed by Holo's interfaces on the Host. These buckets are at standard deviation boundaries in the measured data, which is assumed to be more or less normally distributed.

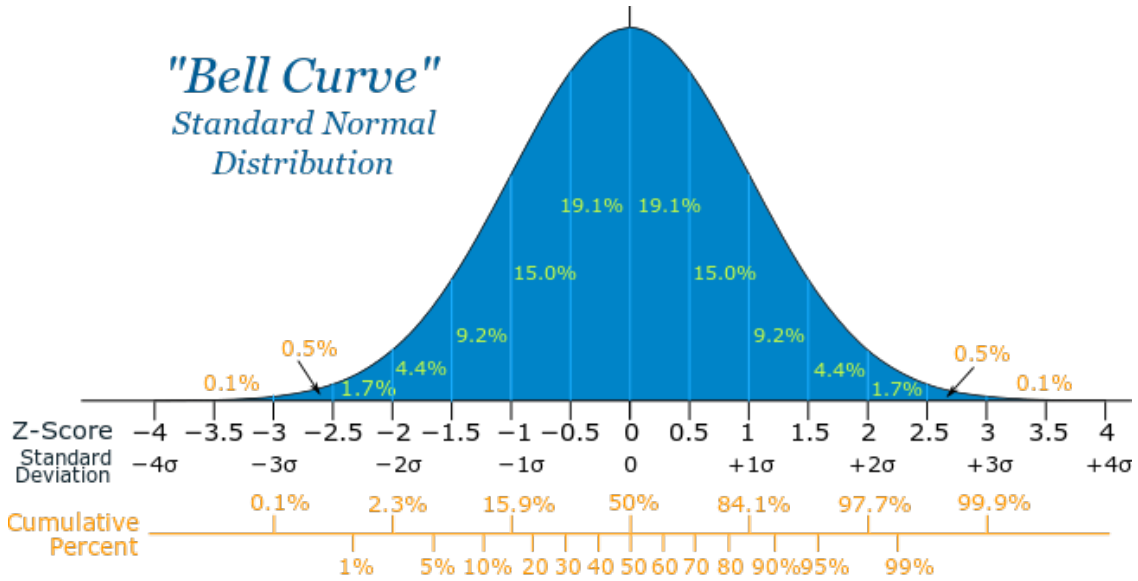


Figure 1: Standard Deviation

The 5 buckets on the "performance" axis contain Hosts which fall in the standard deviation groups $< -1\sigma$ (lolo 15%), $< 0\sigma$ (lo 35%), $> 0\sigma$ (hi 35%) and $> +1\sigma$ (hihi 15%) of the median response time. The 3 buckets on the Holo Host "pricing" axis, **discount** (-2σ - $-.5\sigma$, lo 29%),

market ($-.5\sigma$ - $+.5\sigma$, mid 38%) and **premium** ($+.5\sigma$ - $+2\sigma$, hi 29%) are selected by each Hosts' dynamically adjusted pricing. The lowest performing $< -2\sigma$ (**bulk** 2%) are not used for serving real-time requests, until their response times to bulk requests moves them out of the lolo group. The **peak** nodes can be reserved for the most performance sensitive requests (eg. CDN like activities).

Performance »	2% bulk	28% slow	40% median	28% fast	2% peak
v Price v	$< -2\sigma$	-2σ - $-.5\sigma$	$-.5\sigma$ - $+.5\sigma$	$+.5\sigma$ - $+2\sigma$	$> +2\sigma$
discount	"background"	some requests	most requests	some requests	CDN, web
market	API requests		fewer requests		proxies,
premium	(none; ignored?)	very few	yet fewer request	very few	relay, etc.

This arrangement leads to a cooperative feedback loop, allowing both Holo hApps and Hosts to dynamically adjust their pricing:

- A **premium** Host modulates its prices to keep its utilization in a low band, a **discount** Host does so to keep its utilization high. This causes the Holo hApp administration DNA to collect this information from time to time recompute price statistics and standard deviations, and move it directly between **discount**, ..., **premium** tranches in its performance band.
  - Eventually (as its performance reflects its changed utilization), the Holo hApp manager will also migrate it between **bulk**, **slow**, **median**, **fast** and **peak** performance tranches.
- A **discount** hApp adjusts its cost targets to keep its performance in the lower acceptable range, the **premium** hApp adjusts to keep performance in the higher end of the band. It selects random Hosts from various tranches with varying probabilites to achieve its target average Hosting cost.
  - If costs escalate due to overall increasing Hosting costs, its pool of credit supports less runtime. The owner should be informed that they may want to drop to a lower cost target (eg. from **market** to **discount**) to stretch out its hosting account, or put more money in.

Overall, the process of deploying an hApp:

- Holo Host installs the hApp, identifies itself to the hApp manager
- hApp Manager adds it to the lowest performing (probably cheapest) tranche in each resource category
- Holo begins sending requests, collecting signed service logs
- The Host performance tranches are recomputed based on service log resource utilization and response timing
  - Each service response carries the total used Hosts wall-clock duration (units of Holo) and CPU seconds, plus the total (hourly exponential moving average) Storage, RAM and Network utilization and total wall-clock duration of requests served. This allows us to assign a fraction of the total hApp resource utilization to this request, and deduce a price

  - The cost tranche boundaries are recomputed from the latest set of new Host pricing data collected from ongoing DHT scans of all Hosts, and the Hosts are distributed into their new tranches on the cost axis.

- Prices paid per avg. request increase as Hosts move to higher cost tranches, and/or the standard deviation "boundaries" change and the average price in the target tranches increases.

  - The hApp manager warns the owner of significant changes in hosting costs, so they can adjust their preferred cost settings.

1.2.2 Simulation of Client, hApp, Host and Reserve Interaction

We will simulate a set of tranches of Hosts over a single commodity, 'holo' hosting. This is the commodity representing wall-clock duration of requests serviced by Holo Hosts. Slower hosts are priced cheaply, faster hosts are more expensive. This is an aggregate of all types of requests made to a hosted hApp, so represents the full spectrum of Host behaviours (requests that are not satisfiable in a deterministic time should be excluded for the purposes of Host characterization). For example, a very fast host on a low-latency network but with slow disk storage will be penalized vs. an identical host with SSDs, because its disk-intensive requests will have a response time distribution with higher mean and standard deviation. However, it may offset this by pricing its 'storage' and 'bandwidth' commodities at a premium. For the purposes of the model, various Hosts will satisfy requests at various rates, but they will all be considered to be in one pool. Normally, a host that takes longer to process requests would migrate to a lower performance pool, where its price would probably move it to the **premium** price tranche in that performance band, reducing its request rate, and hence lowering its income.

The value of Holo fuel (its basket) float; the market value (eg. USD$/fuel bid/ask on the Reserve) will fluctuate. Fixing the value-stability of Holo fuel is arbitrary, and can be maintained at an arbitrary value by limiting the inflow of Holo fuel into the core Holo ecosystem. We will not be doing that in this model. However, when prices rise (Client requests increase, Host load increases prices), eventually they will reach the Reserve ask price set by the Hosts' cash-out settings. This will result in the issuance of more Holo fuel, moderating price deflation. Basically, Holo fuel prices should cap out at Host cash-out price. We can limit Reserve issuance and/or increase prices to allow further increase in Holo fuel price, but we won't do that in this model.

Hosts will sell fuel on the Reserve (which is also an Exchange) to whomever is buying at market rates, to maintain their monthly cost needs. Holo hApps will buy on the Reserve/exchange to maintain service. As prices increase HODLers should begin liquidating some of the HOT$177B in Holo fuel holdings to furnish this need. They would sell on a HOT/fuel exchange to do this, but we'll just simulate everyone holding Holo fuel.

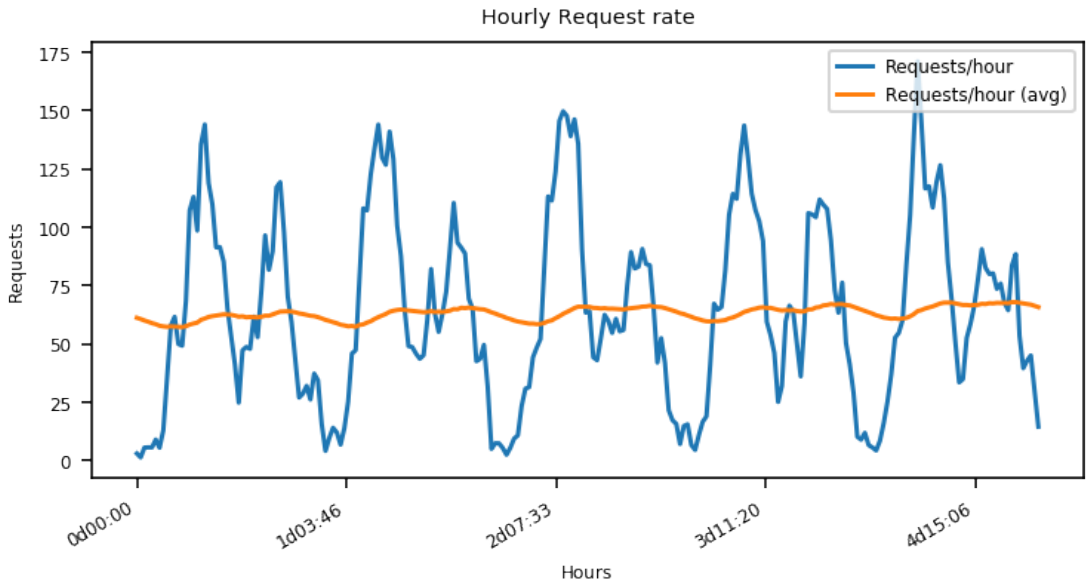
```
from scipy import stats # stats.zscore, stats.norm.cdf, ...

def std_dst_prob( SD ):
    """Given a number of SD away from the mean, compute the probability of that number being part of the
    normal distribution.  For example, if we're +2 standard deviations away, we're in the 97.7th
    percentile; only 2.23% of the population should exceed this value in a normal distribution.  At
    0 SD away, we're right on the 50th percentile; 1/2 should be less, 1/2 more.  We want a function
    that, given a SD, provides us a probability of 1.0 at exactly 0 SD away, and falls off in the
    shape of the Bell Curve as we retreat from the mean; At 0.0, we want 2 x 0.5 == 1.0; At +2.0 or
    -2.0 SD away, we want the result == 2 * 0.0227 == 0.0454 .

        >>> stats.norm.cdf( 0 )
        0.5
```

```
        >>> stats.norm.cdf( +2 )
        0.9772498680518208
        >>> stats.norm.cdf( -2 )
        0.022750131948179195
        >>> 1 - stats.norm.cdf( +2 )
        0.02275013194817921

    """
    if SD < 0:
        return 2.0 * stats.norm.cdf( SD )
    else:
        return 2.0 * ( 1 - stats.norm.cdf( SD ))



def exponential_moving_average( current, sample, weight ):
    return sample if current is None else current + weight * ( sample - current )


class Client_hApp( trading.actor ):
    """A client tries to perform a certain number of requests per hour (via an hApp), during a 12-hour
    window of time peaking around some time during the day.  Our quanta is 1 hour, so we'll
    recompute our next hour's requests every hour -- as well, each Client will have a random start
    time during the first hour.  So, if we sample all the client's self.requests at the Nyquist
    rate, we'll have a good instantaneous view of the current request rate, in request/hr. """
    def __init__( self,
            midday_mean = 12 * trading.hour, midday_sigma = 2,# noon, +/- 2 hours has 2/3 of requests
    requests_mean = 10, requests_sigma = 2,# 10 req/h, +/- 2 peak
    quanta = 1 * trading.hour,# compute next hour's requests hourly
    requests_avg_dur = 2 * trading.day,# keep ?-day rolling average of request rate
    holofuel_runway = 7 * trading.day,# target this many day's fuel on hand
    minimum = -math.inf,# allow client to go into debt
    **kwds ):
        super( Client_hApp, self ).__init__(
quanta = quanta,
minimum = minimum,
needs = [ trading.need_t( 0, None, 'Holofuel', quanta, 0 )], # dummy; adjust targets manually
**kwds )
self.midday_mean = midday_mean # eg. -7 (Mountain), +5 (China Standard)
self.midday_sigma = midday_sigma
self.requests_mean = requests_mean
self.requests_sigma = requests_sigma
self.requests = 0
self.unsatisfied = 0
# avg. should be somewhere around here
self.requests_avg = requests_mean * ( 3 * midday_sigma/2 / 24 ) # 99.8% w/in 3 sigma of mean

self.requests_avg_dur = requests_avg_dur
self.compute_requests_rate( now=self.start )

self.holofuel_runway = holofuel_runway

    def compute_requests_rate( self, now ):
# quanta satisfied; .now updated, .dt has time period since last run.  Compute our next
# hour's number of satisfied/unsatisfied requests, based on the last hour's Holo system
# thruput.  Thus, when Clients want to perform 10 tx but the last hour saw 125% utilization
# of Host resources, the next hour we'll compute the target rate we'd like (say, 10
# requests), but reduce it by / 1.25, and say 8.0 satisfied, 2.0 unsatisfied.
peak_desired = rnd_std_dst( mean=self.requests_mean, sigma=self.requests_sigma, minimum=0 )
# how many hours -/+ from our midday peak utilization?  Pick a random hour we want to peak,
# somewhere near our desired "midday".  Lets pick a random normal value around our target,
# and then compute our Z-score: what is the probability of something being in the normal
# distribution, that far from the mean.  For example, if our curr_hour is 11:00, and our peak
# hour comes out to be exactly 12:00, we're -1 hour away.  If our sigma (size of 1 standard deviation)
# is 6 hours, we're -1/6th SD away from the mean.
curr_hour = ( now % trading.day ) / trading.hour # (0,24] UTC
peak_hour = rnd_std_dst( mean=self.midday_mean, sigma=self.midday_sigma ) # (-12,+12)
norm_hour = ( peak_hour + 24 ) % 24 # convert eg. timezone -7 to +17 hour of UTC day
```

```
diff_hour = curr_hour - norm_hour
if diff_hour < -12:
    diff_hour         += 24
elif diff_hour > +12:
    diff_hour         -= 24
#print( "midday_mean: {self.midday_mean}, peak: {peak_hour}, curr: {curr_hour}, norm: {norm_hour}, diff: {diff_hour}".format( **

curr_sd = diff_hour / self.midday_sigma
curr_hour_prob = std_dst_prob( curr_sd )
hour_target = peak_desired * curr_hour_prob
#print( "curr/peak hour: {} vs. {}, curr_sd: {}, hour_target: {}, prob: {} ".format(
#    curr_hour, peak_hour, curr_sd, hour_target, curr_hour_prob ))
self.requests = hour_target
self.unsatisfied = 0 # TODO: get this from Host average utilization
self.requests_avg = exponential_moving_average( self.requests_avg, self.requests, self.quanta / self.requests_avg_dur )

    def run( self, **kwds ):
        # Runs the Client/hApp trading.actor's needs/targets and issues trades as required to get
# self.targets satisfied.
        if not super( Client_hApp, self ).run( **kwds ):
            return False

# Transfer our last hour's request rate Hosting payment in Holofuel, to a random Host.
host = random.choice( hosts )
cost = self.requests
try:  host.assets['Holofuel'] +=  cost
        except KeyError: host.assets['Holofuel']  = +cost
try:  self.assets['Holofuel'] -=  cost
        except KeyError: self.assets['Holofuel']  = -cost

# Compute our next hour's Request rate, based on time of day
self.compute_requests_rate( now=self.now )

# We've computed our requests_avg Requests/hour.  We have an hourly 'Holofuel' need: but,
# we've set it to 0; we'll adjust self.targets manually to keep (say) a week's worth of
# runway Holo fuel on hand.  We'll be paying Hosting fees hourly for the last hour's
# requests, out of our Holo fuel assets, reducing them, triggering a buy to bring us back up
# to targets.
self.target['Holofuel'] = self.holofuel_runway * self.requests_avg / self.quanta
return True


class Sample_engine( trading.engine_status ):
    def __init__( self, **kwds ):
        super( Sample_engine, self ).__init__( **kwds )
self.requests = [] # [ (<now>,<client-requests>), ... ]
self.hApp_holdings = [] # [ (<now>,{'Holofuel': 123, 'USD':-123}), ... ]
self.Host_holdings = []

    @property
    def hApps( self ):
        for a in self.agents:
            if isinstance( a, Client_hApp ):
             yield a

    @property
    def Hosts( self ):
        for a in self.agents:
            if isinstance( a, Host ):
             yield a

    def status( self, now ):
        """Collect hourly snapshots of all of our Client/hApps' simulated requests for that hour and rolling avg."""
        super( Sample_engine, self ).status( now=now )
self.requests.append( (now, sum( c.requests for c in self.hApps ), sum( c.requests_avg for c in self.hApps )) )
self.hApp_holdings.append( (now, {
    'USD': sum( c.balances.get( 'USD' ) or 0 for c in self.hApps ),
    'Holofuel': sum( c.assets.get( 'Holofuel' ) or 0 for c in self.hApps )
```

```
}) )
self.Host_holdings.append( (now, {
    'USD': sum( c.balances.get( 'USD' ) or 0 for c in self.Hosts ),
    'Holofuel': sum( c.assets.get( 'Holofuel' ) or 0 for c in self.Hosts )
}) )
print( "{}: Holofuel Target: {}, Holdings: {}, USD: {} Order Book:\n{}".format(
    str( self.world ),
    sum( c.target.get( 'Holofuel' ) or 0 for c in self.hApps ),
    sum( c.assets.get( 'Holofuel' ) or 0 for c in self.hApps ),
    sum( c.balances.get( 'USD' ) or 0 for c in self.hApps ),
    self.exchange.format_book() ))

#
# Create varying populations with different request activity times
#
pop_t = collections.namedtuple( 'Population', ['locale', 'tz', 'P'] )
populations = [
    pop_t( 'NA-west', -8,  100e6 * .9 ),
    pop_t( 'NA-mid',  -6,   75e6 * .8 ),
    pop_t( 'NA-east', -5,  150e6 * .9 ),
    pop_t( 'Africa',  +2, 1.21e9 * .2 ),
    pop_t( 'EU',      +0,  750e6 * .7 ),
    pop_t( 'RU',      +3,  150e6 * .6 ),
    pop_t( 'Asia',    +8, 1.38e9 * .3 ),
]

def population_middays( count, *matches ): # eg. 'NA', 'Asia'; empty matches == everything
    pops_selected = [ p for p in populations for m in ( matches or [''] ) if m in p.locale ]
    pop_prob = [ p.P for p in pops_selected ] # population * middle-class,technical
    pop_prob        /= np.sum( pop_prob ) # normalize sum of probabilites to 1.0
    pop_tz = [ p.tz for p in pops_selected ]
    pop_middays = np.random.choice( a=pop_tz, size=count, replace=True, p=pop_prob )
    print( ', '.join( "{:>10}: UTC {:+d} P({:.2f}):".format( locale, tz, prob )
                      for locale,prob,tz in zip( (p.locale for p in pops_selected ), pop_prob, pop_tz )))
    return pop_middays


#
# Render a duration of client_count Client's request activity
#
client_count = 50
sim_duration = 5 * trading.day

clients = [ Client_hApp(
identity = "Cli/hApp{}".format( n ),
midday_mean = midday_mean )
                                    for n,midday_mean in enumerate( population_middays( client_count, 'NA', 'Asia' )) ]


res = reserve_issuing( "Holofuel/USD", LIFO=True, supply_period=trading.hour, supply_available=1e6 )
wld = trading.world( duration=sim_duration, quanta=trading.hour / 4 ) # 2x Nyquise
eng = Sample_engine( world=wld, exch=res, agents=clients + hosts, status_period=1 * trading.hour / 2 )
print( str( eng.world ))
eng.run()
print( "Done: %s" % ( eng.world ))


x_now,y_req,y_avg = [],[],[]
if eng.requests:
    x_now,y_req,y_avg = zip( *eng.requests )

fig,ax = plt.subplots()
plt.plot( x_now, y_req, label="Requests/hour" )
plt.plot( x_now, y_avg, label="Requests/hour (avg)" )
formatter = matplotlib.ticker.FuncFormatter( lambda s, x: '%dd%02d:%02d' % (
    s // trading.day, ( s % trading.day ) // trading.hour, ( s % trading.hour ) // trading.minute ))
ax.xaxis.set_major_formatter( formatter )
fig.autofmt_xdate()
```

```
plt.xlabel( "Hours" )
plt.ylabel( "Requests" )
plt.legend( loc="upper right" )
plt.title( "Hourly Request rate" )
plt.show()
```



So, here we observe the Client/hApp hourly request load generated by a number of clients in a couple of time zones, over a few days; Asia is bigger and more concentrated, NA is more spread out. Of course, we'll spread out the midday times to simulate loads coming from various sizes of populations.

The request rate is dimensionless; we'll assume that it is denominated in 'holo'; 1 'holo' hour is 1 hour of wall-clock duration worth of requests served by a Host. This is generally linearly related to the number of cores on the Host, but is also affected the CPU speed, RAM speed, bandwidth and latency and storage speed. We'll assume our Hosts are pretty homogeneous; basically, 1 hour of 'holo' requires 1 Host for 1 hour; to scale up to service 100 more 'holo' per hour requires 100 more Holo hosts.

This specifies the load exerted on the Hosts, and the hosting payment due by the hApp owners.

Instead of simulating the hApp owner buying Holo fuel and paying it to the Host, we'll simply have the Clients directly buy the Holo fuel required to service their requests. Like a hApp owner, they'll need to buy it at "market" for that period's worth of requests.

The "Requests/hour (avg)" gives the Client/hApp the information it needs to fund future requests. We ensure that our Client/hApps have always got enough Holo fuel to fund a week's worth of hosting.

Holo fuel is transferred to a random Host on an hourly basis by each Client/hApp. Here we see the purchases of Holofuel from the Reserve being transferred to the Hosts, and the increasing total USD$ cost of Hosting accruing to the Client/hApps:

```
fig,(ax0,ax1,ax2)= plt.subplots( 3, sharex=True, figsize=(6,3) )

ax0.plot( [ x for x,h in eng.hApp_holdings ], [h['Holofuel'] for x,h in eng.hApp_holdings ],
              label='Client/hApp Holofuel held' )
```

```
ax0.fmt_ydata = lambda x: '%.2f' % x
ax0.grid( True )
ax0.set_ylabel( 'Holofuel' )

ax1.plot( [ x for x,h in eng.hApp_holdings ], [h['USD'] for x,h in eng.hApp_holdings ],
              label='Client/hApp USD' )
ax1.fmt_ydata = lambda x: '%.2f' % x
ax1.grid( True )
ax1.set_ylabel( 'USD' )

ax2.plot( [ x for x,h in eng.Host_holdings ], [h['Holofuel'] for x,h in eng.Host_holdings ],
              label='Host Holofuel' )
ax2.fmt_ydata = lambda x: '%.2f' % x
ax2.grid( True )
ax2.set_ylabel( 'Holofuel' )

ax2.set_xlabel( "Time (hours)" )
formatter = matplotlib.ticker.FuncFormatter( lambda s, x: '%dd%02d:%02d' % (
    s // trading.day, ( s % trading.day ) // trading.hour, ( s % trading.hour ) // trading.minute ))
ax2.xaxis.set_major_formatter( formatter )
fig.autofmt_xdate()

for a in ax0,ax1,ax2:
    a.legend( loc="right" )
```