Holo Fuel Reserve Account Design and Modelling

Perry Kundert

September 25, 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
```

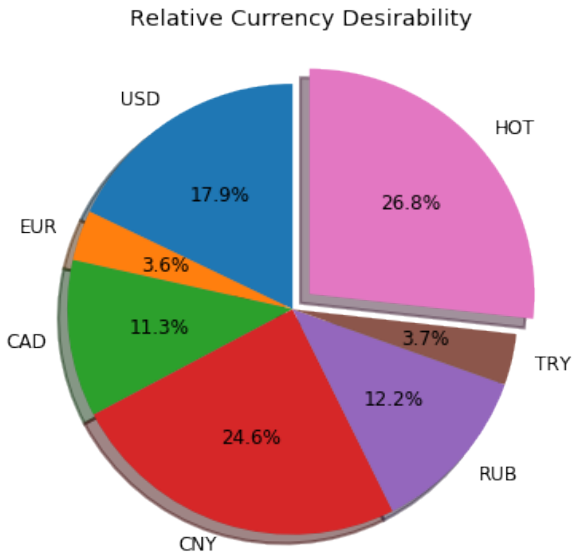
```
from sklearn import linear_model
import collections
import math
import random
import json
import bisect

from holofuel.model import trading


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 ):
        """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
}

        """
        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( 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	17.9	46.0	0.4271	0.4731	0.1911
EUR	3.6	21.0	0.3777	0.3732	0.1467
CAD	11.3	28.0	0.4853	0.4437	0.2542
CNY	24.6	45.0	2.9580	3.1561	1.2583
RUB	12.2	37.0	25.5505	26.2552	10.4647
TRY	3.7	13.0	1.3631	1.1278	0.7017
HOT	26.8	46.0	1.0000	1.0000	0.0000

1.1 Host Autopilot 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 Autopilot 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 autopilot 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