Holo Fuel Model

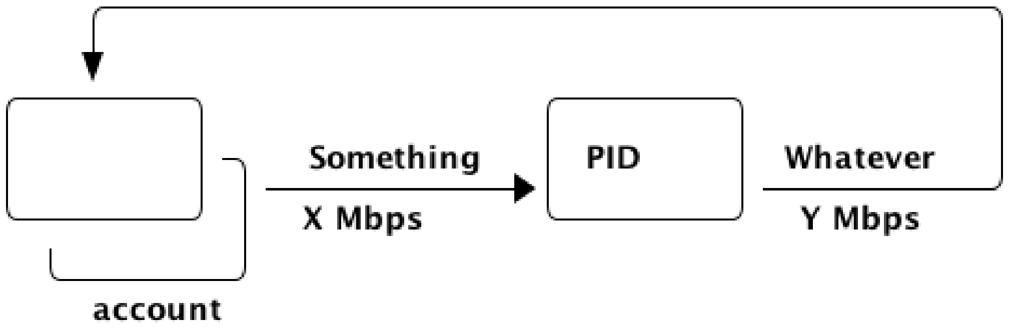
Perry Kundert

July 31, 2018

Contents

1 Holo Fuel Valuation

A value-stable wealth-backed cryptocurrency platform, where each unit is defined in terms of a basked of computational resources, operating in a powerful decentralized verification environment.



1.1 The Computational Resources Basket

One Holo Fuel (HOT) is defined as being able to purchase 1 month of Holo Hosting services for the front-end (ie. web API, databases, etc.) portion of a typical dApp.

This might be roughly equivalent to the 2018 price of (and actual utilization of) a small cloud hosting setup (eg. several $5/month Droplet on Digital Ocean at partial utilization hosting front-ends, DBs, backups, etc.), plus ancillary hosting services (represented by a premium for inclusion in the Holo system).

If the minimal Hosting costs for a small Web app is estimated at USD$100/mo., and comprises 5 cloud hosting nodes and anciliary services for the various aspects of the system. An equivalent Holo Host based system would have similar CPU and storage requirements overall, but a greater redundancy (say, 5 x, so all DHTs spread across 25 Holo Hosts).

Thus, the basket of commodities defining the value of USD$100 ~ 100 Holo Fuel could be defined as:

Commodity	Amount	Units	Weight	Description
holo	25.00	Host	60.000%	Inclusion in the Holo system
net	0.50	TB	20.000%	Internet bandwidth
ram	1.00	GB	5.000%	Processor memory
data	0.25	TB	10.000%	Persistent storage (DHT/DB/file)
cpu	1.00	Core	5.000%	A processing core

1.1.1 Holo Hosting Premium

A Holochain Distributed Application (dApp) hosted on Holo provides a valuable set of features, over and above simply hosting a typical web application on a set of cloud servers. These services must usually be either purchased, or architected by hand and distributed across multiple cloud hosting nodes for redundancy.

- ☐ Reliability. Few single points of failure.
- ☐ Backup. All DHT data is spread across many nodes.
- ☐ Scalability. Automatically scales to absorb increased load.

The value of Holo is substantial in terms of real costs to traditional app developers, and is a component of the basket of commodities defining the price of Holo Fuel. However, it's real monetary value will emerge over time, as the developer community comprehends it. Our pricing algoritm must be able to dig this Holo premium component out of the historical hosting prices, as a separate component.

1.1.2 Resource Price Stability

There are many detailed requirements for each of these commodities, which may be required for certain Holochain applications; CPU flags (eg. AVX-512, cache size, . . .), RAM (GB/s bandwidth), HDD (time to first byte, random/sequential I/O bandwidth), Internet (bandwidth/latency to various Internet backbone routers).

The relative distribution of these features will change over time; RAM becomes faster, CPU cores more powerful. The definition of a typical unit of these commodities therefore changes; as Moore's law decreases the price, the specifications of the typical computer also improve, counterbalancing this inflationary trend.

For each metric, the price of service on the median Holo Host node will be used; 1/2 will be below (weaker, priced at a discount), 1/2 above (more powerful, priced at a premium). This will nullify the natural inflationary nature of Holo Fuel, if we simply defined it in terms of fixed 2018 computational resources.

1.2 Commodity Price Discovery

Value stabilization requires knowledge of the current prices of each commodity in the currency's valuation basket, ideally denominated in the currency itself. If these commodities are traded within the cryptocurrency implementation, then we can directly discover them on a distributed basis. If outside commodity prices are used, then each independent actor computing the control loop must either reach consensus on the price history (as collected from external sources, such as Distributed Oracles), or trust a separate module to do so. In Holo Fuel, we host the sale of Holo Host services to dApp owners, so we know the historical prices.

When a history of Holo Hosting service prices is available, Linear Regression can be used to discover the average fixed (Holo Hosting premium) and variable (CPU, . . .) component costs included in the prices, and therefore the current commodity basket price.

1.2.1 Recovering Commodity Basket Costs

To illustrate price recovery, lets begin with simulated prices of a basket of commodities. A prototypical minimal dApp owner could select 100 Holo Fuel worth of these resources, eg. 25x Holo Hosts, .05 TB data, 1.5 cpu, etc. as appropriate for their specific application's needs.

This Hosting selection wouldn't actually be a manual procedure; testing would indicate the kind of loads to expect for a given amount and type of user activity, and a calculator would estimate the various resource utilization and costs. At run time, the credit extended to the dApp owner (calculated from prior history of Hosting receipt payments) would set the maximum outstanding Hosting receipts allowed; the dApp deployment would auto-scale out to qualified Hosts in various tranches as required; candidate Hosts (hoping to generate Hosting receipts) would auto-install the application as it reached its limits of various resource utilization metrics across its current stable of Hosts.

```
def rnd_std_dst( sigma, mean=0, minimum=None ):
    """ """
    val                 = sigma * np.random.randn() + mean
    return val if minimum is None else max( minimum, val )

# To simulate initial pricing, lets start with an estimate of proportion of basket value represented
# by each amount of the basket's commodities.  Prices of each of these commodities is free to float
# in a real market, but we'll start with some pre-determined "weights"; indicating that the amount
# of the specified commodity holds a greater or lesser proportion of the basket's value.
# Regardless, 100 Holo Fuel is guaranteed to buy the entire basket.
prices                  = {}
for k in basket:
    price_mean          = basket_target * weight[k] / basket[k] # target price: 1 Holo Fuel == 1 basket / basket_target
    price_sigma         = price_mean / 10 #  difference allowed; about +/- 10% of target
    prices[k]           = rnd_std_dst( price_sigma, price_mean )

[ [ "Commodity", "Price", "Per", "Per" ],
  None ] \
+ [ [ k, "%5.2f" % ( prices[k] ), commodities[k].units, 'mo.' ]
    for k in basket ]
```

Commodity	Price	Per	Per
holo	2.49	Host	mo.
net	34.93	TB	mo.
ram	5.18	GB	mo.
data	38.56	TB	mo.
cpu	5.90	Core	mo.

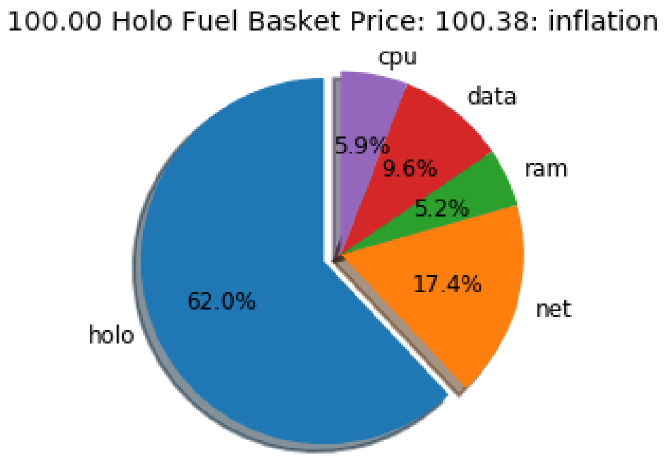
From this set of current assumed commodity prices, we can compute the current price of the Holo Fuel currency's basket:

```
basket_price            = sum( basket[k] * prices[k] for k in basket )
[ [ "Holo Fuel Basket Price" ],
  None,
  [ "$%5.2f / %.2f" % ( basket_price, basket_target ) ] ]
```

Holo Fuel Basket Price
$100.38 / 100.00

If the current price of this basket is >100, then we are experiencing commodity price inflation; if <100, price deflation. Feedback control loops will act to bring the price back to 100 Holo Fuel per basket.

```
labels                  = [ k for k in basket ]
sizes                   = [ basket[k] * prices[k] for k in basket ]
explode                 = [ .1 if k == 'holo' else 0 for k in basket ]
# with plt.xkcd():
fig1,ax1        = plt.subplots()
ax1.pie( sizes, explode=explode, labels=labels, autopct='%1.1f%%', shadow=True, startangle=90 )
ax1.axis( 'equal' ) # Equal aspect ratio ensures that pie is drawn as a circle.
plt.title( "%6.2f Holo Fuel Basket Price: %6.2f: %sflation" % (
    basket_target, basket_price, "in" if basket_price > basket_target else "de" ))
plt.show()
```



1.2.2 Holo Hosting Receipts

Once we have the currency's underlying commodity basket, lets simulate a sequence of trades of various amounts of these commodities. In the Holo system, this is represented by Hosts issuing receipts for services to dApp owners.

Each Hosting receipt will be for a single Holo Host, not for the entire dApp; the sum of all Holo Hosting receipts issued to the dApp owner for our archetypical small dApp would sum to approximately 100 Holo Fuel per month.

We will not know the exact costs of each commodity used to compute the price, or how much is the baseline Holo system premium. However, it will be dependant on the capability of the Host (stronger hosts can charge more, for hosting more specialized dApps), and the amount of various services used.

So, lets issue a bunch of small Holo Hosting receipts, each for approximately 1/25th of the total Holo Hosting load (since our small dApp is spread across 25 Holo Hosts).

```
amounts_mean            = 1.00
amounts_sigma           = 0.5
error_sigma             = 0.10 # +/- 10% variance in bids (error) vs. price
trades                  = []
number                  = 10000
for _ in range( number ):
    # Each dApp consumes a random standard distribution of the target amount of each commodity
    amounts             = { k: 1 if k == 'holo'
       else basket[k] * rnd_std_dst( amounts_sigma, amounts_mean, minimum=0 ) / basket['holo']
    for k in basket }
    price               = sum( amounts[k] * prices[k] for k in amounts )
    error               = price * rnd_std_dst( error_sigma )
    bid                 = price + error
    trades.append( dict( bid = bid, price = price, error = error, amounts = amounts ))

[ [ "Fuel","calc/err", "dApp Requirements" ], None ] \
+ [ [
      "%5.2f" % t['bid'],
      "%5.2f%+5.2f" % ( t['price'], t['error'] ),
       ", ".join( "%5.4f %s %s" % ( v, k, commodities[k].units ) for k,v in t['amounts'].items() ),
    ]
    for t in trades[:5] ] \
+ [ [ '...' ] ]
```

Fuel	calc/err	dApp Requirements
4.74	4.09+0.65	1.0000 holo Host, 0.0215 net TB, 0.0339 ram GB, 0.0119 data TB, 0.0375 cpu Core
4.16	3.93+0.23	1.0000 holo Host, 0.0181 net TB, 0.0582 ram GB, 0.0131 data TB, 0.0000 cpu Core
3.67	3.94-0.27	1.0000 holo Host, 0.0119 net TB, 0.0485 ram GB, 0.0118 data TB, 0.0559 cpu Core
3.07	3.30-0.24	1.0000 holo Host, 0.0132 net TB, 0.0186 ram GB, 0.0010 data TB, 0.0373 cpu Core
4.51	4.23+0.28	1.0000 holo Host, 0.0199 net TB, 0.0276 ram GB, 0.0180 data TB, 0.0354 cpu Core
...		

1.2.3 Recovery of Commodity Valuations

Lets see if we can recover the approximate Holo baseline and per-commodity costs from a sequence of trades. Create some trades of 1 x Holo + random amounts of commodities around the requirements of a typical Holo dApp, adjusted by a random amount (ie. 'holo' always equals 1 unit, so that all non-varying remainder is ascribed to the "baseline" Holo Hosting premium).

Compute a linear regression over the trades, to try to recover an estimate of the prices.

```
items                  = [ [ t['amounts'][k] for k in basket ] for t in trades ]
bids                   = [ t['bid'] for t in trades ]

regression             = linear_model.LinearRegression( fit_intercept=False, normalize=False )
regression.fit( items, bids )
select                 = { k: [ int( k == k2 ) for k2 in basket ] for k in basket }
predict                = { k: regression.predict( select[k] ) for k in basket }

[ [ "Score(R^2): ", "%.9r" % ( regression.score( items, bids )), '', '' ],
  None ] \
+ [ [ "Commodity",  "Predicted", "Actual", "Error",
      # "selected"
  ],
  None ] \
+ [ [ k,
      "%5.2f" % ( predict[k] ),
      "%5.2f" % ( prices[k] ),
      "%+5.3f%%" % (( predict[k] - prices[k] ) * 100 / prices[k] ),
      #select[k]
    ]
    for k in basket ]
```

Score(R^2):	0.5288336		
Commodity	Predicted	Actual	Error
holo	2.48	2.49	-0.281%
net	35.60	34.93	+1.940%
ram	5.42	5.18	+4.658%
data	38.53	38.56	-0.090%
cpu	5.58	5.90	-5.496%

1.2.4 Commodity Basket Valuation

Finally, we can estimate the current Holo Fuel basket price from the recovered commodity prices.

```
basket_predict         = sum( basket[k] * predict[k]  for k in basket )
[ [ "Holo Fuel Price Recovered", "vs. Actual", "Error" ], None,
  [ "$%5.2f / %.2f" % ( basket_predict, basket_target ),
    "%5.2f" % ( basket_price ),
    "%+5.3f%%" % (( basket_predict - basket_price ) * 100 / basket_price ),
    ]]
```

Holo Fuel Price Recovered	vs. Actual	Error
$100.46 / 100.00	100.38	+0.072%

We have shown that we should be able to recover the underlying commodity prices, and hence the basket price with a high degree of certainty, even in the face of relatively large differences in the mix of prices paid for hosting.

1.3 Simple Value Stability Control via PID

The simplest implementation of value-stability is to directly control the credit supply. Lets establish a simple wealth-backed monetary system with a certain amount of wealth attached to it, from which we extend credit at a factor `K` of 0.5 to begin with; half of the value of the wealth is available in credit. Adjusting `K` increases/reduces the liquid credit supply.

The economy has a certain stock of Host resources available (eg. cpu, net, . . .), and a certain pool of dApp owners wanting to buy various combinations of them. The owners willing to pay more will get preferred access to the resources. In a traditional bid/ask market, greater bids are satisfied first, lesser later or not at all. In Holo, tranches of similar Hosts round-robin requests from clients of the dApps they host.

1.3.1 Host/dApp Pricing

In the Holo Host environments, Hosts are pooled in tranches of like resource capacity (eg. cpu: type, count, . . .), quality (eg. service: availability, longevity, . . .), and price (eg. autopilot/manual pricing: lolo, lo, median, hi, hihi). A multi-dimensional table of Host tranches is maintained; each Host inserts itself into the correct table.

- TODO: How do the DHT peers confirm that a Host isn't lying about its internal computational resources? A dApp could check, and issue a warrant if the Host is lying, but a DHT peer couldn't independently verify these claims. There will be great incentive to inflate claims, to draw and serve higher-priced requests. . .)

A dApp owner also selects the resource requirements (eg. cpu: avx-128+, 4+ cores, . . .) service level and pricing (eg. median).

Requests from hihi priced dApps are distributed first to the lolo, then lo, . . . , hihi tranches of Hosts, as each tranche's resources is saturated; thus, lolo priced Hosts are saturated first. Then, hi dApps are served any by lolo, lo, . . . Hosts not yet saturated, and so on. Thus, in times of low utilization (less dApps than Hosts), the highest priced Hosts may remain idle; in high utilization (more dApps than Hosts), the lowest priced dApp's requests may remain unserved (or, perhaps throttled and served round-robin, to avoid complete starvation of the lower priced dApp groups). Of course, these tranches of Hosts are also limited (via a set Union) to those Hosts in each tranche that **also** host a given target dApp, and requests for a dApp are only sent to those hosts who can service it.

- TODO: Each TCP/IP HTTP socket, representing 1 or more HTTP requests or a WebSocket initiation, is assigned a Host; does Holo terminate the connection and relay I/O to/from the Host? It should pre-establish a pool of sockets to candidate Hosts, ready to be distributed to incoming requests, thus eliminating the delay of the 3-way handshake, and pre-eliminating dead/unreachable Hosts.) This requires a persistent proxy a.la. Cloudflare. Much more simply, perhaps, we could build DNS servers that advertise multiple A records from an appropriate tranche of candidate servers, in round-robin fashion, and let the end-user sort out servers that disappear (until the DNS server figures out they're dead and stops serving their IP address). However, intervening caching DNS servers (eg. at large ISPs) could conduit large numbers of request (ie. from the entire ISP!) to those few Host A-records for the time-to-live of the cached DNS query.

1.3.2 Host/dApp Pricing Automation Approaches

How does the system compute the actual price that "median" Hosts get paid? How does it evolve over time? 1/2 of requests should go to median, lo, lolo Hosts, and 1/2 should go to median, hi, hihi Hosts. A PID loop could move the "Median" Host price to make this true, perhaps. Hosts should set a minimum average price they'll earn, dApps a maximum average price they're willing to pay, and their requests are throttled to only the Host tranches which satisfy these limits.

By automatically switching a Host to higher/lower pricing tiers, and the dApp to lower/higher pricing selections, as their limit prices are reached, the numbers of Hosts/dApps above/below "median" changes – and the PID loop adjusts the median price to achieve above/below equilibrium. Thus, as more dApps exceed their high limit, switch to lower tiers (eg. from hi –> median –> lo), the mix of requests above/below median price changes, and the PID loop responds by adjusting the median Hosting price, which affects average dApp request pricing, which causes the dApp to hit its limits, which causes it to (again) switch to a lower tier…

Of course, the dApp owner is informed of this, in real time, and can make price limit adjustments, to re-establish dApp performance. Likewise, a Hosting owner can see that their Hosts are saturated/idle, and increase/decrease their minimum price, or maximum utilization targets; the Host should increase its desired pricing tier, to stay under its maximum utilization target.

1.3.3 Simple Host/dApp Pricing Model

For the purposes of this simple test, we'll assume that the Host will simply spend all the credit the dApp has available serving its requests (we won't simulate the dApps). So, lets generate a sequence of request service receipts from the Host to dApp owners, tuned to the credit available to the dApp.

```
class credit_static( object ):
    """Simplest, static K-value, unchanging basket and prices."""
    def __init__( self, K, basket, prices ):
self.K          = K
self.basket     = basket
self.prices     = prices

    def value( self, prices=None, basket=None ):
"""Compute the value of a basket at some prices (default: self.basket/prices)"""
if prices is None: prices = self.prices
if basket is None: basket = self.basket
return sum( prices[k] * basket[k] for k in basket )

# Adjust this so that our process value 'basket_value' achieves setpoint 'basket_target'
# Use the global basket, prices defined above
credit                  = credit_static( K=0.5, basket=basket, prices=prices )

#print( "Global basket: %r, prices: %r" % ( basket, prices ))
#print( "credit.basket: %r, prices: %r" % ( credit.basket, credit.prices ))

duration_hour           = 60 * 60
duration_day            = 24 * duration_hour
duration_month          = 365.25 * duration_day / 12 # 2,629,800s.

used_mean               = 1.0                   # Hourly usage is
used_sigma              = used_mean * 10/100    # +/-10%
reqs_mean               = 2.0                   # Avg. Host is 2x minimal
reqs_sigma              = reqs_mean * 50/100    # +/-50%
reqs_min                = 1/10                  #   but at least this much of minimal dApp
```

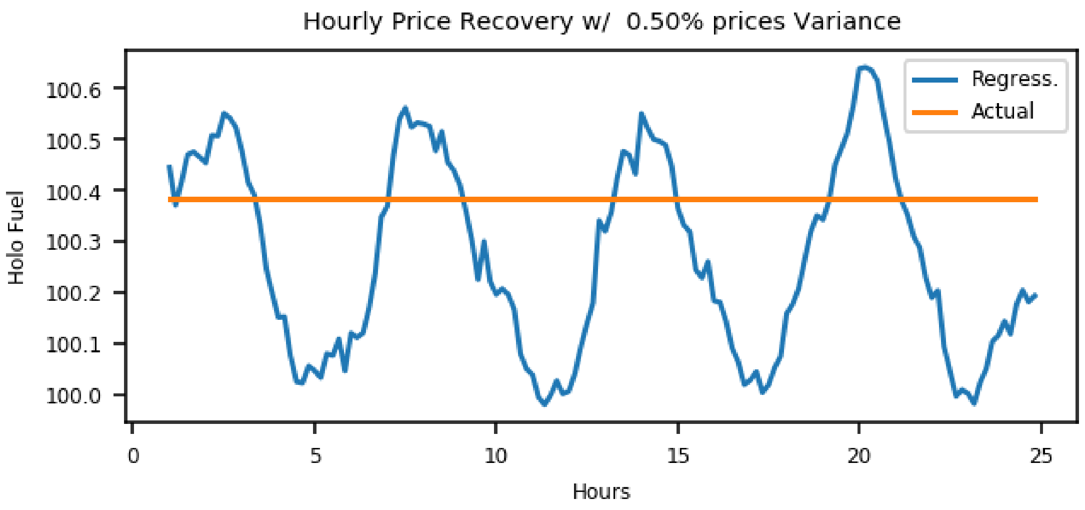
```
class dApp( object ):
    def __init__( self, duration=duration_month ): # 1 mo., in seconds
"""Select a random basket of computational requirements, some multiple of the minimal dApp
represented by the Holo Fuel basket (min. 10% of basket), for the specified duration."""
self.duration   = duration
self.requires   = { k: rnd_std_dst( sigma=reqs_sigma, mean=reqs_mean, minimum=reqs_min ) \
 * credit.basket[k] * duration / duration_month
     for k in credit.basket }
# Finally, compute the wealth required to fund this at current credit factor K
self.wealth      = credit.value( basket=self.requires ) / credit.K
#print( repr( self ))

    def __repr__( self ):
return "<dApp using %8.2f Holo Fuel / %5.2f mo.: %s" % (
   credit.value( basket=self.requires ), self.duration/duration_month,
   ", ".join( "%6.2f %s %s" % ( self.requires[k] * self.duration/duration_month,
       commodities[k].units, k ) for k in credit.basket ))

    def available( self, dt=None ):
"""Credit available for dt seconds (1 hr., default) of Hosting."""
return self.wealth * credit.K * ( dt or duration_hour ) / self.duration

    def used( self, dt=None, mean=1.0, sigma=.1 ):
"""Resources used over period dt (+/- 10% default, but at least 0)"""
return { k: self.requires[k] * rnd_std_dst( sigma=sigma, mean=mean, minimum=0 ) * dt / self.duration
 for k in self.requires }

class Host( object ):
    def __init__( self, dApp ):
self.dApp       = dApp

    def receipt( self, dt=None ):

"""Generate receipt for dt seconds worth of hosting our dApp.  Hosting costs more/less as prices
fluctuate, and dApp owners can spend more/less depending on how much credit they have
available.  This spending reduction could be acheived, for example, by selecting a lower
pricing teir (thus worse performance)."""

avail           = self.dApp.available( dt=dt )                # Credit available for this period
used            = self.dApp.used( dt=dt, mean=used_mean, sigma=used_sigma ) # Hhosting resources used
value           = credit.value( basket=used )                 # total value of dApp Hosting resources used

# We have the value of the hosting the dApp used, at present currency.prices.  The Host
# wants to be paid 'value', but the dApp owner only has 'avail' to pay. When money is
# plentiful/tight, dApp owners could {up,down}grade their service teir and pay more or less.
# So, we'll split the difference.  This illustrates the effects of both cost variations and
# credit availability variations in the ultimate cost of Hosting, and hence in the recovered
# price information used to adjust credit.K.

#print( "avail: {}, value: {}, used: {}".format( avail, value, used ))
return (( avail + value ) / 2, used )

hosts_count             = 60 * 60 # ~1 Hosting receipt per second
hosts                   = [ Host( dApp() ) for _ in range( hosts_count ) ]
hours_count             = 24

class credit_sine( credit_static ):
    """Compute a sine scale as the basis for simulating various credit system variances."""
    def __init__( self, amp, step, **kwds ):
self.sine_amp   = amp
self.sine_theta = 0
self.sine_step  = step
self.K_base     = 0
super( credit_sine, self ).__init__( **kwds )

    def advance( self ):
self.sine_theta+= self.sine_step
```

```
    def reset( self ):
"""Restore credit system initial conditions."""
self.sine_theta = 0

    def scale( self ):
return 1 + self.sine_amp * math.sin( self.sine_theta )

class credit_sine_K( credit_sine ):
    """Adjusts credit.K on a sine wave."""
    @property
    def K( self ):
return self.K_base * self.scale()
    @K.setter
    def K( self, value ):
"""Assumes K_base is created when K is set in base-class constructor"""
self.K_base     = value

class credit_sine_prices( credit_sine ):
    """Adjusts credit.prices on a sine wave."""
    @property
    def prices( self ):
return { k: self.prices_base[k] * self.scale() for k in self.prices_base }
    @prices.setter
    def prices( self, value ):
self.prices_base     = prices

# Create receipts with a credit.K or .prices fluctuating +/- .5%,  1 cycle per 6 hours
#credit.advance          = lambda: None # if using credit_static...
#credit.sine_amp         = 0
credit                   = credit_sine_prices( K=0.5, amp=.5/100,
 step=2 * math.pi / hosts_count / 6,
 prices=prices, basket=basket ) # Start w/ the global basket
receipts                = []
for _ in range( hours_count ):
    for h in hosts:
receipts.append( h.receipt( dt=duration_hour ))
credit.advance()
credit.reset()

items                   = [ [ rcpt[k] for k in credit.basket ] for cost,rcpt in receipts ]
costs                   = [ cost for cost,rcpt in receipts ]

regression              = linear_model.LinearRegression( fit_intercept=False, normalize=False )
regression.fit( items, costs )
select                  = { k: [ int( k == k2 ) for k2 in credit.basket ] for k in credit.basket }
predict                 = { k: regression.predict( select[k] ) for k in credit.basket }

actual_value            = credit.value()
predict_value           = credit.value( prices=predict )
[ [ "%dhr. x %d Hosts Cost" % ( hours_count, hosts_count ) ] + list( rcpt.keys() ),
  None,
  [ "%8.6f" % sum( cost for cost,rcpt in receipts ) ] \
  + [ "%8.6f" % sum( rcpt[k] for cost,rcpt in receipts ) for k in credit.basket ],
  None,
  [ "Score(R^2) %.9r" % ( regression.score( items, costs )) ],
  [ "Predicted" ] + [ "%5.2f" % predict[k] for k in credit.basket ],
  [ "Actual" ]    + [ "%5.2f" % credit.prices[k] for k in credit.basket ],
  [ "Error" ]     + [ "%+5.3f%%" % (( predict[k] - credit.prices[k] ) * 100 / credit.prices[k] )
      for k in credit.basket ],
  None,
  [ "Actual  Basket", "%5.2f" % actual_value ],
  [ "Predict Basket", "%5.2f" % predict_value ],
  [ "Error" , "%+5.3f%%" % (( predict_value - actual_value ) * 100 / actual_value ) ],
]
```

24hr. x 3600 Hosts Cost	holo	net	ram	data	cpu
23771.336824	5919.350677	118.116487	237.688079	59.503334	235.787431
Score(R^2) 0.9877677					
Predicted	2.45	34.99	5.53	39.30	6.27
Actual	2.49	34.93	5.18	38.56	5.90
Error	-1.686%	+0.175%	+6.786%	+1.918%	+6.342%
Actual Basket	100.38				
Predict Basket	100.27				
Error	-0.108%				

Lets see how well an hourly linear regression tracks the actual Basket price, in 10 minute intervals (so, 6 x 1-hour regression samples per hour). Lets see if we can pick up the 1% sine-wave variation in Credit Factor K every 6 hours:

```
# x is the fractional hour of the end of each hour-long segment
x_divs            = 6
x                 = [ 1 + s / x_divs for s in range( hours_count * x_divs ) ]
reg               = []
act               = []
for h in x: # Compute beg:end indices from fractional hour at end of each 1-hour range
    beg,end       = int( (h-1) * hosts_count ),int( h * hosts_count )
    items         = [ [ rcpt[k] for k in credit.basket ] for cost,rcpt in receipts[beg:end] ]
    costs         = [ cost                             for cost,rcpt in receipts[beg:end] ]
    regression.fit( items, costs )
    select        = { k: [ int( k == k2 ) for k2 in credit.basket ] for k in credit.basket }
    predict       = { k: regression.predict( select[k] ) for k in credit.basket }
    reg.append( credit.value( predict ))
    act.append( credit.value() )
plt.plot( x, reg, label="Regress." )
plt.plot( x, act, label="Actual" )
plt.xlabel( "Hours" )
plt.ylabel( "Holo Fuel" )
plt.legend( loc="upper right" )
plt.title( "Hourly Price Recovery w/ %5.2f%% %s Variance" % (
    credit.sine_amp * 100, credit.__class__.__name__.split( '_' )[-1] ))
plt.show()
```



1.3.4 Simple Credit Feedback Control

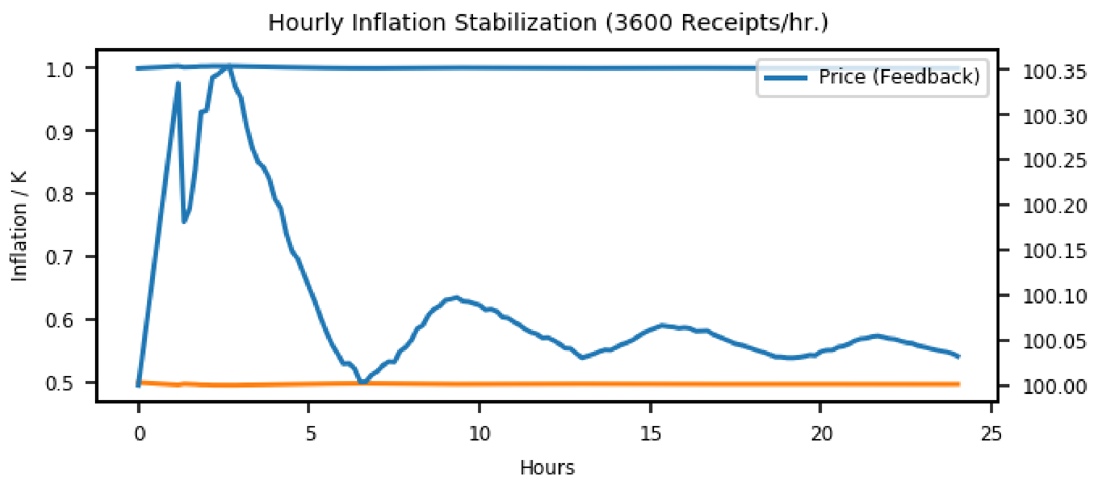
Finally, we have almost everything required to actually control the currency, using a simple PID controller.

```
import time
import sys
import math
if not hasattr( math, 'nan' ):
    math.nan            = float( 'nan' )

timer                   = time.clock if sys.platform == 'win32' else time.time

Kpid_t                  = collections.namedtuple( 'Kpid_t', ['Kp', 'Ki', 'Kd'] )
Lout_t                  = collections.namedtuple( 'Lout_t', ['lo', 'hi'] )

def clamp( val, lim ):
    """Return value if between range limits, otherwise the limit; math.nan indicates no limit"""
    if val < lim[0]:
return lim[0]
    elif val > lim[1]:
return lim[1]
    return val

class controller( object ):
    """Simple PID loop with Integral anti-windup, bumpless transfer, and setpoint change damping."""
    def __init__( self, Kpid, setpoint=None, process=None, output=None,
  Lout=( math.nan, math.nan ), now=None ):
self.Kpid       = Kpid( 1, 1, 1 ) if Kpid is None else Kpid_t( *Kpid )
self.Lout       = Lout_t( math.nan, math.nan ) if Lout is None else Lout_t( *Lout )

self.setpoint   = setpoint or 0
self.process    = process or 0
self.output     = output or 0

self.now        = now if now is not None else timer()
self.P          = self.setpoint - self.process
# Bumpless transfer; compute I required to maintain steady-state output
self.I          = ( self.output - self.P * self.Kpid.Kp ) / self.Kpid.Ki if self.Kpid.Ki else 0
self.D          = 0

    def loop( self, setpoint=None, process=None, now=None ):
"""Any change in setpoint? If our error (P - self.P) is increasing in a direction, and the
setpoint moves in that direction, cancel that amount of the rate of change."""
dS              = 0
if setpoint is not None:
    dS          = setpoint - self.setpoint
    self.setpoint = setpoint
if process is not None:
    self.process = process
if now is None:
    now         = timer()
if now > self.now: # No contribution if no +'ve dt!
    dt          = now - self.now
    self.now= now
    P           = self.setpoint - self.process # Proportional: setpoint and process value error
    I           = self.I + P * dt              # Integral:     total error under curve over time
    D           = ( P - self.P - dS ) / dt     # Derivative:   rate of change of error (net dS)
    self.output= ( P * self.Kpid.Kp + I * self.Kpid.Ki + D * self.Kpid.Kd )
    self.P      = P
    if not ( self.output < self.Lout.lo and I < self.I ) and \
       not ( self.output > self.Lout.hi and I > self.I ):
self.I  = I                            # Integral anti-windup; ignore I if saturated
    self.D      = D
return self.value

    @property
    def value( self ):
return clamp( self.output, self.Lout )

    def __repr__( self ):
       return "<%r: %+8.6f %s %+8.6f --> %+8.6f (%+8.6f) P: %+8.6f * %+8.6f, I: %+8.6f * %+8.6f, D: %+8.6f * %+8.6f>" % (
   self.now, self.process,
   '>' if self.process > self.setpoint else '<' if self.process > self.setpoint else '=',
   self.setpoint, self.value, self.output,
   self.P, self.Kpid.Kp, self.I, self.Kpid.Ki, self.D, self.Kpid.Kd )

def near( a, b, significance = 1.0e-4 ):
    """ Returns True iff the difference between the values is within the factor 'significance' of
    one of the original values.  Default is to within 4 decimal places. """
    return abs( a - b ) <= significance * max( abs( a ), abs( b ))

def nearprint( a, b, significance = 1.0e-4 ):
    if not near( a, b, significance ):
print( "%r != %r w/in +/- x %r" % ( a, b, significance ))
return False
    return True

control             = controller( Kpid = ( 2.0, 1.0, 2.0 ), setpoint=1.0, process=1.0, now = 0. )
assert near( control.loop( 1.0, 1.0, now = 1. ),   0.0000 )
assert near( control.loop( 1.0, 1.0, now = 2. ),   0.0000 )
assert near( control.loop( 1.0, 1.1, now = 3. ),  -0.5000 )
assert near( control.loop( 1.0, 1.1, now = 4. ),  -0.4000 )
assert near( control.loop( 1.0, 1.1, now = 5. ),  -0.5000 )
assert near( control.loop( 1.0, 1.05,now = 6. ),  -0.3500 )
assert near( control.loop( 1.0, 1.05,now = 7. ),  -0.5000 )
assert near( control.loop( 1.0, 1.01,now = 8. ),  -0.3500 )
assert near( control.loop( 1.0, 1.0, now = 9. ),  -0.3900 )
assert near( control.loop( 1.0, 1.0, now =10. ),  -0.4100 )
assert near( control.loop( 1.0, 1.0, now =11. ),  -0.4100 )
```

Lets implement a simple credit system that adjust K via the PID loop to move the price of the credit basket towards our target value. We'll produce a stream of Hosting receipts, based on the current basket price and available credit. Then, we'll compute the

```
class credit_sine_prices_pid_K( credit_sine_prices ):
    """Adjusts credit.K via PID, in response to prices varying according to a sine wave."""
    def __init__( self, price_target=None, price_initial=None, Kpid=None, now=None, **kwds ):
"""A current price_target (default: 100.0 ) and price_feedback (default: price_target) is
is used to initialize a PID loop."""
super( credit_sine_prices_pid_K, self ).__init__( **kwds )
self.now        = now or 0 # hours?
self.price_target = price_target or basket_target # 100.0 Holo Fuel / basket, defined above
self.price_curr = price_initial or self.price_target
self.price_curr_trend = [(self.now, self.price_curr)]
self.inflation  = self.price_curr / self.price_target
self.inflation_trend = [(self.now, self.inflation)]
# Bumpless start at setpoint 1.0, present inflation, and output of current K
self.K_control  = controller( # TODO: compute Kpid fr. desired correction factors vs. avg target dt
   Kpid = Kpid or ( 1., .1, .01 ),
    now = self.now,
       setpoint = 1.0,                  # Target is no {in,de}flation!
process = self.inflation,
 output = self.K )
self.K_trend   = [(self.now, self.K)]

    def price_feedback( self, price, now ):
"""Supply a computed basket price at time 'now', and compute K via PID."""
self.now        = now
self.price_curr = price
self.price_curr_trend += [(self.now, self.price_curr)]
self.inflation  = self.price_curr / self.price_target
self.inflation_trend += [(self.now, self.inflation)]
self.K          = self.K_control.loop(
process = self.inflation,
    now = self.now )
self.K_trend   += [(self.now, self.K)]
```

```
    def receipt_feedback( self, receipts, now ):
"""Extract price_feedback from a sequence of receipts"""
items           = [ [ rcpt[k] for k in credit.basket ] for cost,rcpt in receipts ]
costs           = [ cost                             for cost,rcpt in receipts ]
regression.fit( items, costs )
select          = { k: [ int( k == k2 ) for k2 in self.basket ] for k in self.basket }
predict         = { k: regression.predict( select[k] ) for k in self.basket }
self.price_feedback( self.value( prices=predict ), now=now )

credit                   = credit_sine_prices_pid_K(
 K=0.5, amp=.5/100,
 step=2 * math.pi / hosts_count / 6,
 prices=prices, basket=basket ) # Start w/ the global basket
print( "credit.basket: %r, prices: %r" % ( credit.basket, credit.prices ))

receipts                = []
for x in range( hours_count ):
    for h in hosts:
if len( receipts ) >= hosts_count \
   and  int(  len( receipts )       * x_divs / hosts_count ) \
     != int(( len( receipts ) + 1 ) * x_divs / hosts_count ):
    # About to compute next hours / x_divs' receipt! Compute and update prices using last
    # hour's receipts.  The now time (in fractional hours) is the length of receipts.
    credit.receipt_feedback( receipts[:-hosts_count], now=len( receipts ) / hosts_count )
receipts.append( h.receipt( dt=duration_hour ))
credit.advance() # adjust market prices algorithmically
credit.reset()

fig,ax1                 = plt.subplots()
ax1.plot( [ x for x,I in credit.inflation_trend ], [ I for x,I in credit.inflation_trend ], label="Inflation" )
ax1.plot( [ x for x,K in credit.K_trend ],         [ K for x,K in credit.K_trend ], label="K" )
ax1.set_xlabel( "Hours" )
ax1.set_ylabel( "Inflation / K" )

ax2                     = ax1.twinx()
ax2.plot( [ x for x,P in credit.price_curr_trend ], [ P for x,P in credit.price_curr_trend ], label="Price (Feedback)" )

#fig.tight_layout()
plt.legend( loc="upper right" )
plt.title( "Hourly Inflation Stabilization (%d Receipts/hr.)" % ( hosts_count ))
plt.show()
print( "done" )
```



2 Holo Fuel Value Stabilization

Price discovery gives us the tools we need to detect {in,de}flation as it occurs. Control of liquid credit available in the marketplace gives us the levers we need to eliminate it.

Traditional Fiat currencies control the issuance of liquidity by influencing the commercial banks to create more or less money through lending, and to increase/reduce liquidity through the net issuance/retirement of debt (which creates/destroys the principal money).

2.1 Wealth Monetization

In a wealth-backed currency, credit is created by the attachment of wealth to the monetary system, and credit lines of varying proportions being extended against the value of that wealth.

Depending on savings rates, monetary velocity, public sentiment etc., the amount of credit available to actually be spent varies. Since this available liquid credit is split between possible expenditures in priority order, the amount available to spend on each specific commodity therefore varies, driving the market price up and down.

If reliable indicators of both the liquid credit supply within, and the quality and amount of wealth attached, exist within the system itself then control systems can be executed within the system to automatically control the monetization of wealth to achieve credit unit value equilibrium – value-stability.

Each reserve of wealth provided different flows and indicators, and can support value-stability in different ways.

2.1.1 Reserve Accounts

The Reserve Accounts provide the interface between external currencies (eg. USD$, HOT ERC20 Tokens) and Holo Fuel.

Deposits to the reserve creates Holo Fuel credit limit (debt) at a current rate of exchange (TBD; eg. market rate + premium/discount). The corresponding Holo Fuel credits created are deposited to the recipient's account.

If Holo Fuel inflation occurs within the system, credit must be withdrawn. One way to accomplish this is to discourage creation of Holo Fuel (and encourage the redemption of Holo Fuel), by increasing the exchange rate. The inverse (lowering exchange rate) would result in more Holo Fuel creation (less redemption), reducing the Holo Fuel available, and thus reduce deflation.

The Reserve Accounts can respond very quickly, inducing Holo Hosts with Holo Fuel balances to quickly convert them out to other currencies when exchange rates rise. Inversely, reducing rates would release waiting dApp owners to purchase more Holo Fuel for hosting their dApps, deploying it into the economy to address deflation (increasing computational commodity prices).

A PD (Proportional Differential) control might be ideal for this. This type of control responds quickly both to direct errors (things being the wrong price), but most importantly to changes in the 2nd derivative (changes in rate of rate of change); eg. things getting more/less expensive at an increasing rate.

By eliminating the I (Integral) component of the PID loop, it does **not** slowly build up a systematic output bias; it simply adjusts the instantaneous premium/discount added to the current market exchange rate (eg. the HOT ERC20 market), to arrive at the Reserve Account exchange rate. When inflation/deflation disappears, then the Reserve Account will have the same exchange rate as the market.

Beginning with a set of reserves:

```
reserve_t               = collections.namedtuple(
    'Reserve', [
'rate',     # Exchange rate used for these funds
'amount',   # The total value of the amount executed at .rate
     ] )            #   and the resultant credit in Holo Fuel == amount * rate

reserve                 = {
    'EUR':          [],     # LIFO stack of reserves available
    'USD':          [ reserve_t( .0004, 200 ), reserve_t( .0005, 250 ) ], # 1,000,000 Holo Fuel
    'HOT ERC20':    [ reserve_t( 1, 1000000 ) ], # 1,000,000 Holo Fuel
}

def reserves( reserve ):
    return [ [ "Currency", "Rate avg.", "Reserves", "Holo Fuel Credits", ], None, ] \
   + [ [ c, "%8.6f" % ( sum( r.amount * r.rate for r in reserve[c] )
       / ( sum( r.amount for r in reserve[c] ) or 1 ) ),
 "%8.2f" % sum( r.amount for r in reserve[c] ),
 "%8.2f" % sum( r.amount / r.rate for r in reserve[c] ) ]
       for c in reserve ] \
   + [ None,
       [ '', '', '', sum( sum( r.amount / r.rate for r in reserve[c] ) for c in reserve ) ]]

summary                 = reserves( reserve )
summary # summary[-1][-1] is the total amount of reserves credit available, in Holo Fuel
```

Currency	Rate avg.	Reserves	Holo Fuel Credits
HOT ERC20	1.000000	1000000.00	1000000.00
USD	0.000456	450.00	1000000.00
EUR	0.000000	0.00	0.00
			2000000.0

As a simple proxy for price stability, lets assume that we strive to maintain a certain stock of Holo Fuel credits in the system for it to be at equilibrium. We'll randomly do exchanges of Holo Fuel out through exchanges at a randomly varying rate (also varied by the rate premium/discount), and purchases of Holo Fuel through exchanges at a rate proportional to the premium/discount.

```
t_last                  = -1
for t in range( 1000 ):
    dt                  = t - t_last
```