Holo Fuel Model

Perry Kundert

August 3, 2018

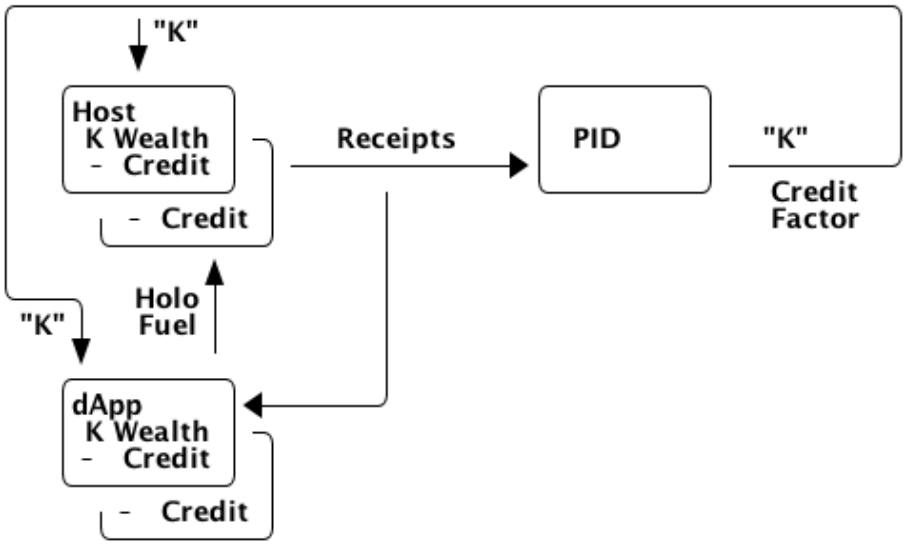
Contents

1 Holo Fuel Valuation **1**
- 1.1 The Computational Resources Basket . . . 2
  - 1.1.1 Holo Hosting Premium . . . 2
  - 1.1.2 Resource Price Stability . . . 3
- 1.2 Commodity Price Discovery . . . 3
  - 1.2.1 Recovering Commodity Basket Costs . . . 3
  - 1.2.2 Holo Hosting Receipts . . . 5
  - 1.2.3 Recovery of Commodity Valuations . . . 6
  - 1.2.4 Commodity Basket Valuation . . . 6
- 1.3 Simple Value Stability Control via PID . . . 7
  - 1.3.1 Host/dApp Pricing . . . 7
  - 1.3.2 Host/dApp Pricing Automation Approaches . . . 8
  - 1.3.3 Simple Host/dApp Pricing Model . . . 8
  - 1.3.4 Simple Credit Feedback Control . . . 12
  - 1.3.5 Agent-Centric Distributed Calculation of "K" . . . 18

2 Holo Fuel Value Stabilization **18**
- 2.1 Wealth Monetization . . . 18
  - 2.1.1 Simple Reserve Accounts . . . 19
  - 2.1.2 Simple LIFO Exchange . . . 20
  - 2.1.3 Active Exchange Flow Balancing . . . 21
  - 2.1.4 Incomplete. 21

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 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

def rnd_std_dst( sigma, mean=0, minimum=None, maximum=None ):
    """Random values with mean, in a standard distribution w/ sigma, clamped to given minimum/maximum."""
    return clamp( sigma * np.random.randn() + mean,
                  ( math.nan if minimum is None else minimum,
                    math.nan if maximum is None else maximum ))

# 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	2.45	Host	mo.
net	36.90	TB	mo.
ram	5.49	GB	mo.
data	44.06	TB	mo.
cpu	4.55	Core	mo.

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

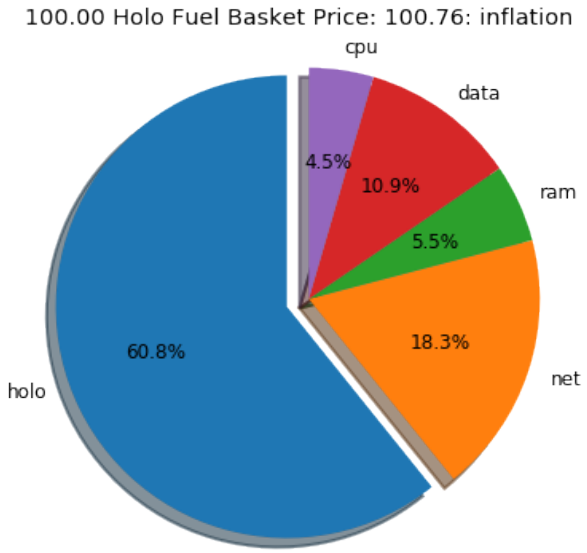
```
basket_price            = sum( basket[k] * prices[k] for k in basket )
[ [ "Holo Fuel Basket Price" ],
  None,
  [ "$%5.2f / %.2f" % ( basket_price, basket_target ) ] ]
```

Holo Fuel Basket Price
$100.76 / 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
4.44	4.54-0.10	1.0000 holo Host, 0.0339 net TB, 0.0248 ram GB, 0.0100 data TB, 0.0566 cpu Core
3.46	3.52-0.06	1.0000 holo Host, 0.0173 net TB, 0.0555 ram GB, 0.0012 data TB, 0.0161 cpu Core
4.10	4.51-0.40	1.0000 holo Host, 0.0313 net TB, 0.0340 ram GB, 0.0137 data TB, 0.0247 cpu Core
4.20	4.37-0.16	1.0000 holo Host, 0.0177 net TB, 0.0365 ram GB, 0.0198 data TB, 0.0419 cpu Core
3.76	4.01-0.25	1.0000 holo Host, 0.0171 net TB, 0.0294 ram GB, 0.0101 data TB, 0.0718 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] )[0] for k in basket } # deref numpy.array

[ [ "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):	0.5414745		
Commodity	Predicted	Actual	Error
holo	2.47	2.45	+0.691%
net	36.47	36.90	-1.166%
ram	5.54	5.49	+0.810%
data	43.80	44.06	-0.589%
cpu	4.23	4.55	-7.025%

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
$100.63 / 100.00	100.76	-0.131%

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, and thus indirectly control the credit flows (liquid credit availability, monetary velocity and relative pricing).

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.

Once approach dApp owners can use to stay within budget is to adjust their preferred pricing; lower pricing tiers access lower performing and/or higher utilization tranches of Hosts. This results in lower receipt costs for the dApp owner, but also for lower aggregate average prices for the resources – lowering the median basket prices, and hence reducing "deflation".

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.[1]

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

[1] 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. Host performance ranking is defined by aggregate perceived performance of the Host, as measured by the Clients of the dApp. Holo should sort clients into buckets by locale before assigning the appropriate Hosting tranches for its requests (ie. Hosts nearby the Client). Measurements (success and elapsed time) of requests requiring significant amounts of single resources (eg. access to lots of locally stored data, or lots of CPU) should be measured, and stored as Hosting Performance records in the Client's DHT. The Hosting Receipt's logs contain references to relevant Client Hosting Performance records. The dApp owner can then rank each Host by Client-perceived performance for each "telltale" request. A Host that is consistently below/above median performance can be Warranted for re-positioning to a different Host tranche. This could, in fact, be the primary method for moving hosts into the correct performance Tranches. New Hosts always get put into the lowest performance Tranche first, and are moved "up" when positive "Wrong Tranch" Warrants outweigh negative ones, over some period of time.

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.[2]

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). This would be roughly equivalent to the effect of a dApp auto-pricing model where the maximum Hosting performance available within the monthly credit budget is automatically selected.

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 ):
```

[2]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.

```
        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, mean 2 x 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; work back
        # from the desired credit budget, to the amount of wealth that would produce that at "K".
        # Of course wealth is a "stock", a budget funds a "flow", and we're conflating here. But,
        # this could represent a model where the next round of Hosting's estimated cost is budgetted
        # such that we always have at least one month of available credit to sustain it.
        self.wealth     = credit.value( basket=self.requires ) / credit.K

    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).
```

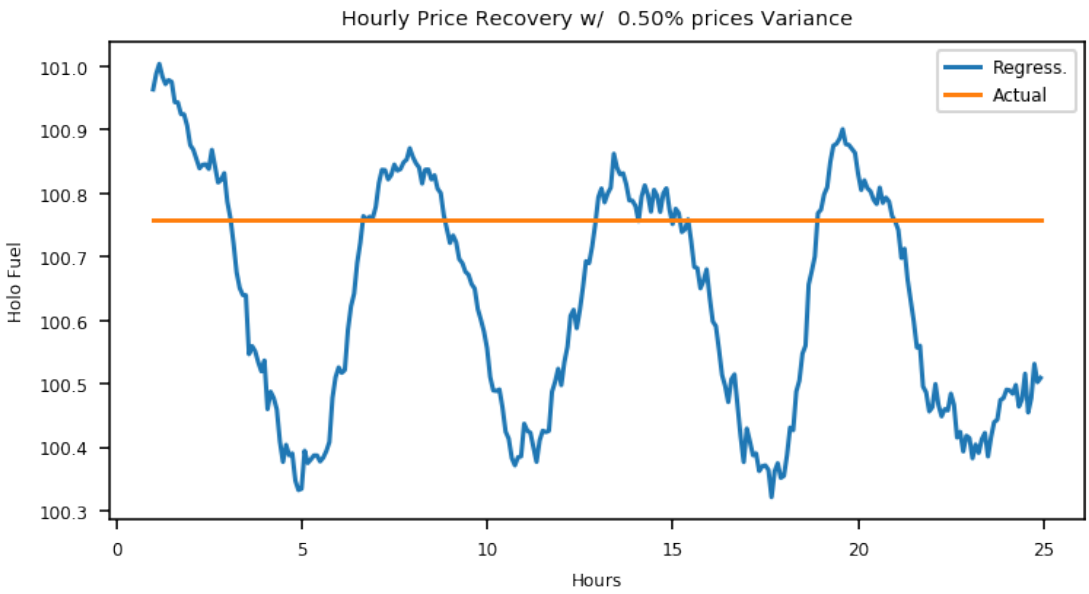
```
        """
        avail           = self.dApp.available( dt=dt )                # Credit available for this period
        used            = self.dApp.used( dt=dt, mean=used_mean, sigma=used_sigma ) # Hhosting resources used
        value           = credit.value( basket=used )                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

        result          = ( avail + value ) / 2,used
        #print( "avail: {}, value: {}, K: {!r},  result: {!r}".format( avail, value, credit.K, result ))
        return result

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] )[0] 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
23966.307685	5942.459196	119.530682	237.308268	58.870610	241.299306
Score(R^2) 0.9869816					
Predicted	2.41	36.88	5.84	44.89	4.94
Actual	2.45	36.90	5.49	44.06	4.55
Error	-1.713%	-0.074%	+6.302%	+1.876%	+8.593%
Actual Basket	100.76				
Predict Basket	100.64				
Error	-0.118%				

Lets see how well an hourly linear regression tracks the actual Basket price, in 5 minute intervals (so, 12 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            = 12 # 5 minutes
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         = [ [ r[k] for k in credit.basket ] for c,r in receipts[beg:end] ]
    costs         = [ c                               for c,r 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] )[0] 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'] )

class controller( object ):
    """Simple PID loop with Integral anti-windup, bumpless transfer."""
    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.bumpless( setpoint=setpoint, process=process, output=output, now=now )

    def bumpless( self, setpoint=None, process=None, output=None, now=None ):
        """Bumpless control transfer; compute I required to maintain steady-state output,
        and D such that a subsequent loop with idential setpoint/process won't produce a
        Differential output.
        """
        if setpoint is not None or self.setpoint is None:
            self.setpoint = setpoint or 0
        if process is not None or self.process is None:
            self.process = process or 0
        if output is not None or self.output is None:
            self.output  = output or 0

        self.now        = timer() if now is None else now
```

```
        self.P          = self.setpoint - self.process
        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.  Quench Integral
        wind-up, if the output is saturated in either direction.  Finally clamp the output drive
        to saturation limits.
        """
        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, and I moving in wrong direction
            self.D      = D
        return self.drive

    @property
    def drive( self ):
        """Limit drive by clampin graw self.output to any limits established in self.Lout"""
        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.drive, 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 underlying "median" basket price, with actual pricing modulated by available credit. Then, we'll recover this pricing from the steam of Hosting receipts, compute the current Credit Factor "K", and issue the next Hosting receipts with respect to the (newly altered) Credit Limit computed from the dApp owner's wealth and "K".

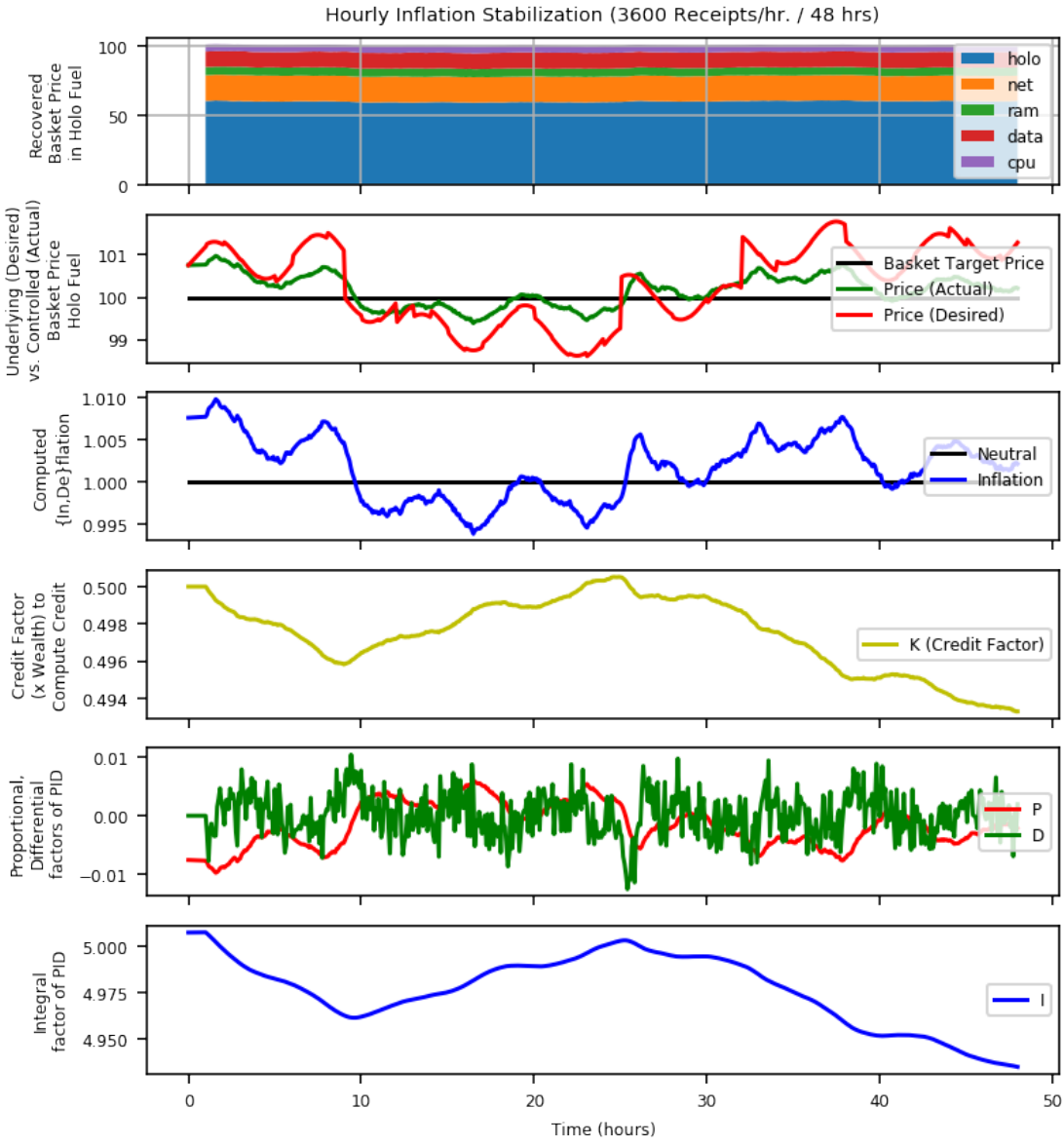
This will simulate the feedback effect on pricing of increasing and decreasing credit availability. It assumes that there is a reasonably swift pricing response to credit limit changes. If the credit lines are broadly enough distributed in the economy (every Host and dApp Owner has one), and many Hosts and dApp Owners use automation that responds to available credit for pricing decisions, this assumption should be satisfied.

```
import json
import traceback
import random

# Make random changes to the pricing of individual computational resources, to simulate
# the independent movement of commodity prices.
adva_mean               = 1.0                   # Parity
adva_sigma              = 1/100                 #  +/- 2% x standard distribution
adva_min                = 98/100                # Trending downward (ie. Moore's law)
adva_max                =102/100                # b/c 102% doesn't fully recover from 98%

class credit_sine_prices_pid_K( credit_sine_prices ):
    """Adjusts credit.K via PID, in response to prices varying randomly, and to a sine wave."""
    def __init__( self, Kpid=None, price_target=None, price_curr=None, now=None, **kwds ):
        """A current price_target (default: 100.0 ) and price_feedback (default: price_target)
        is used to initialize a PID loop.
        """
        super( credit_sine_prices_pid_K, self ).__init__( **kwds )
        self.now        = now or 0 # hours?
        # Default: 100.0 Holo Fuel / basket, defined above
        self.price_target = price_target if price_target is not None else basket_target
        # Default to 0 inflation if no price_curr given
        self.price_curr = price_curr if price_curr is not None else 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
        # TODO: compute Kpid fr. desired correction factors vs. avg target dt.
        self.K_control  = controller(
                            Kpid = Kpid or ( .1, .1, .001 ),
                        setpoint = 1.0,                 # Target is no {in,de}flation!
                         process = self.inflation,
                          output = self.K,
                             now = self.now )
        self.K_trend    = [(self.now, self.K)]
        self.PID_trend  = [(self.now, (self.K_control.P, self.K_control.I, self.K_control.D))]
        self.price_trend= [(self.now, self.value())]
        self.feedback_trend =[]

    def bumpless( self, price_curr, now ):
        """When taking control of the currency after a period of inactivity, reset the PID
        parameters to ensure a "bumpless" transfer starting from current computed inflation/K.
        """
        self.now        = now
        self.price_curr = price_curr
        self.inflation  = price_curr / self.price_target
        self.K_control.bumpless(
                        setpoint = 1.0,
```

```
                    process = self.inflation,
                     output = self.K,
                        now = now )

    def price_feedback( self, price, now, bumpless=False ):
        """Supply a computed basket price at time 'now', and compute K via PID. If we are
        assuming control (eg. after a period of inactivity), reset PID control to bumplessly
        proceed from present state; otherwise, compute K from last time quanta's computed state.
        """
        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)]
        if bumpless:
            self.bumpless( price_curr=self.price_curr, now=now )
        else:
            self.K      = self.K_control.loop(
                        process = self.inflation,
                            now = self.now )
        self.K_trend   += [(self.now, self.K)]
        self.PID_trend += [(self.now, (self.K_control.P, self.K_control.I, self.K_control.D))]
        self.price_trend += [(self.now, self.value())]

    def receipt_feedback( self, receipts, now, bumpless=False ):
        """Extract price_feedback from a sequence of receipts via linear regression.  Assumes that the
        'holo' component is a "baseline" (is assigned all static, non-varying base cost, not
        attributable to varying usage of the other computational resources); it is always a simple
        function of how much wall-clock time the Receipt represents, as a fraction of the 1 'holo'
        Host-month included in the basket.  The remaining values represent how many units (eg. GB
        'ram', TB 'storage', fraction of a 'cpu' Core's time consumed) of each computational
        resource were used by the dApp during the period of the Receipt.
        """
        items           = [ [ r[k] for k in credit.basket ] for c,r in receipts ]
        costs           = [ c                              for c,r in receipts ]
        try:
            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] )[0] for k in self.basket }
            self.price_feedback( self.value( prices=predict ), now=now, bumpless=bumpless )
            self.feedback_trend += [(self.now, { k: self.basket[k] * predict[k]
                                                 for k in self.basket })]
        except Exception as exc:
            print( "Regression failed: %s" % ( exc ))
            traceback.print_stack( file=sys.stdout )

    def advance( self ):
        """About once per integral time period (eg. hour), randomly purturb the pricing of one
        commodity in the basket.  We'll manipulate the underlying self.prices_base (which is being
        modulated systematically to produce the base commodity prices).
        """
        super( credit_sine_prices_pid_K, self ).advance()
        if int( getattr( self, 'adv_h', 0 )) != int( self.now ):
            self.adv_h = int( self.now )
            k          = random.choice( list( prices.keys() ))
            adj        = rnd_std_dst( sigma=adva_sigma, mean=adva_mean,
                                      minimum=adva_min, maximum=adva_max )
            self.prices_base[k] *= adj

# Create the credit system targetting neutral {in,de}flation of 1.0. The underlying basket and prices
# are globals, created above, randomly starting at some offset from neutral inflation.  We are varying
# the amount of credit available, essentially forcing dApp owners to opt for lower or higher tranches
# of service to stay within their available credit.

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 prices, basket
                  price_target = basket_target,
                    price_curr = credit.value() )     # Est. initial price => inflation

# Run a simulation out over a couple of days.  This will simulate a base Price of a Desired level of
# service (say, a certain Tranche of Hosts w/ a certain level of performance), but will simulate a
# withdrawal of credit from the system (eg. available to the dApp owner owners), which forces them
# to elect a lower service level (at lower prices), or gain access to a higher level of service
# (with greater available credit) and pay more.  We will also from time to time randomly adjust the
# pricing of one component of the basket relative to all others, to illustrate the effect of
# changing the supply/demand of just one portion of the computational commodities underlying Holo
# Fuel), and observe how the system responds.

hours_count             = 24 * 2
receipts                = []
for x in range( hours_count ):
    for h in hosts:
        receipts.append( h.receipt( dt=duration_hour ))
        if len( receipts ) >= hosts_count \
           and  int(  len( receipts )       * x_divs / hosts_count ) \
             != int(( len( receipts ) - 1 ) * x_divs / hosts_count ):
            # After 1st hr; About to compute next hours / x_divs' receipt! Compute and update
            # prices using last hour's receipts.  The now time (in fractional hours) is length
            hrs         = len( receipts ) / hosts_count
            #print( "After %5.2fh (%02d:%02d): %d receipts, %d K_trend (%f - %f)" % (
            #    hrs, int( hrs ), int( hrs * 60 ) % 60, len( receipts ),
            #    len( credit.K_trend ), credit.K_trend[0][0], credit.K_trend[-1][0] ))
            credit.receipt_feedback( receipts[-hosts_count:], now=hrs,
                                     bumpless=( len( receipts ) == hosts_count ))
        credit.advance() # adjust market prices algorithmically
credit.reset()
#print("K trend: %f - %f" % ( credit.K_trend[0][0], credit.K_trend[-1][0] ))

# Show how Inflation / K, Price, and PID evolve over time
fig,(ax0,ax1,ax2,ax3,ax4,ax5)= plt.subplots( 6, sharex=True, figsize=(6,7) )
#for k in credit.basket:
ax0.stackplot( [ x for x,F in credit.feedback_trend ],
               [ [ F[k] for x,F in credit.feedback_trend ] for k in credit.basket ],
               labels=list( credit.basket ) )
ax0.fmt_ydata = lambda x: '%.2f' % x
ax0.grid( True )
ax0.set_ylabel( "Recovered\nBasket Price\nin Holo Fuel" )
ax1.plot( [ 0, hours_count ], [ credit.price_target, credit.price_target ],
         "k-",    label="Basket Target Price" )
ax1.plot( [ x for x,P in credit.price_curr_trend ], [ P for x,P in credit.price_curr_trend ],
         "g-",    label="Price (Actual)" )
ax1.plot( [ x for x,P in credit.price_trend ], [ P for x,P in credit.price_trend ],
         "r-",    label="Price (Desired)" )
ax1.fmt_ydata = lambda x: '%.2f' % x
ax1.set_ylabel( "Underlying (Desired)\nvs. Controlled (Actual)\nBasket Price\nHolo Fuel" )

ax2.plot( [ 0, hours_count ], [ 1, 1 ],
         "k-",    label="Neutral" )
ax2.plot( [ x for x,I in credit.inflation_trend ], [ I for x,I in credit.inflation_trend ],
         "b-",   label="Inflation" )
ax2.set_ylabel( "Computed\n{In,De}flation" )

ax3.plot( [ x for x,K in credit.K_trend ],         [ K for x,K in credit.K_trend ],
          "y-",   label="K (Credit Factor)" )
ax3.set_ylabel( "Credit Factor\n(x Wealth) to\nCompute Credit" )

ax4.plot(  [ x for x,(P,I,D) in credit.PID_trend ],[ P for x,(P,I,D) in credit.PID_trend ],
         "r-",    label="P" )
ax4.plot(  [ x for x,(P,I,D) in credit.PID_trend ],[ D for x,(P,I,D) in credit.PID_trend ],
         "g-",    label="D" )
ax4.set_ylabel( "Proportional,\nDifferential\nfactors of PID" )
```

```
ax5.plot(  [ x for x,(P,I,D) in credit.PID_trend ],[ I for x,(P,I,D) in credit.PID_trend ],
         "b-",    label="I" )
ax5.set_ylabel( "Integral\nfactor of PID" )
ax5.set_xlabel( "Time (hours)" )
for a in ax0,ax1,ax2,ax3,ax4,ax5:
    a.legend( loc="right" )

ax0.set_title( "Hourly Inflation Stabilization ({} Receipts/hr. / {} hrs)".format(
                hosts_count, hours_count ))
plt.show()
```



- The scale of the PID loop Integral factor is such that it takes about 24 hours to adjust credit lines, even for large Inflation swings. We do not want to adjust Host/Owner credit lines abruptly.

- We are using linear regression over the last hour of Hosting receipt data to recover prices.

Each receipt represents some prior time period, so the PID loop is operating on pricing approximations from data hours old, so P and D response is not likely to be very helpful.

- Linear regression can wildly miscalculate attribution, especially if thin data with outliers is used. This is another reason that the long-term Integral of error under the curve is more desirable; it minimizes the effect of occasional dramatic miscalculations of basket pricing.

1.3.5 Agent-Centric Distributed Calculation of "K"

No single agent, or group of colluding agents, must have influence over "K".

- The Holo Tranche dApp
  - Knows all Hosts within the Tranche (transfers Hosts moving from/to adjacent Tranches)
  - Collects Hosting receipts from its Host, on behalf of the entire Tranche of similar Hosts.
  - Computes Holo Fuel commodity prices and Basket price on a certain interval (eg. 5m.), over a longer interval of Hosting receipt data (eg. 1hr.)[3]
- The Holo dApp
  - Knows all Tranches providing Holo Hosting (N-dimensional array of Tranches)
  - Collects commodity and Basket price computations from each Holo Tranche (eventually consistent)
  - Computes median

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).

Holo Fuel is created through wealth-backed credit lines, which are adjusted dynamically to increase and decrease liquid credit availability, offsetting deflation and inflation.

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 (a stock, or supply) available to actually be spent (a flow) varies. Since this available liquid credit is typically 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.

[3]The time frames are short enough that the prices should be considered "constant" during the period; ie. won't contain Hosting reports priced under very different economic climates. This means that a simple multiple linear regression over the data, producing a linear model of the price (the dependent variable) as a function of the commodities (explanatory variables) is sufficient. From that model, we can predict the individual commodity prices, and then multiply by the amounts of each commodity in the Holo Fuel Basket to arrive at the Basket price.

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. The attachment of wealth in the form of Hosting capacity, or a dApp owner's demonstrated ability to pay, can be directly measured by the monetary system (as demonstrated above in 1.3.4, above).

Other types of wealth such as Fiat currency can be attached, but are not directly measured within the Holo system. Therefore, we must dynamically respond to both changes in the value of these relative to each-other, and relative to Holo Fuel – without intrinsic knowledge of either their relative values, or absolute value vs. Holo Fuel.

2.1.1 Simple Reserve Accounts

The Reserve Accounts provide the interface between external currencies (eg. USD$, HOT ERC20 Tokens) and Holo Fuel. Consider a simplistic Reserve Account design:

Deposits to the reserve creates Holo Fuel credit limit (debt) at a current rate of exchange (TBD; eg. Book Value + premium/discount). The corresponding Holo Fuel credits created are deposited to the recipient's account. The currency remains in the Reserve Account, and a negative amount of Holo Fuel created is added to the Exchanges' Holo Fuel credit balance. When Holo Fuel is bought or sold for theses reserve currencies over time, an Book Value (average $ per Holo Fuel) emerges.

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

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 minimizing 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 Book Value 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 Book Value.

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
```

(Incomplete...)

2.1.2 Simple LIFO Exchange

Holo Fuel is an accounting mechanism, representing a pre-purchased amount of Hosting, within the Holo system. It is not a traditional currency, nor is it a commodity. By ensuring value-stability, one unit of Holo Fuel is guaranteed to purchase 1 basket of computational resources within the system. It is much more similar to an Apple iTunes card balance, redeemable later for various goods and services at some expected value (eg. about USD$1 per iTunes card unit).

This means that, necessarily, the exchange rates between Holo Fuel and various external currencies must be dynamically controlled to maintain this invariant. Failing to do so will destroy Holo as a viable system; who would purchase Holo Fuel, if A) the price paid is unfair (eg. vs. other currencies), B) the value of it fluctuates wildly over time (eg. can't purchase the amount of Hosting required when used later), or C) cannot be sold by Hosting service owners, later, for a reasonably stable real value in some external currency?

Therefore, allowing arbitrary exchange in/out of Holo Fuel at some arbitrary historical exchange rate (eg. the rate the last person bought in at) is probably not viable. When real exchange rates drop (ie. due to currency values fluctuating), internal Holo Fuel holders would be encouraged to sell Holo Fuel for say USD$ at a premium vs. its actual current market value.

While this would certainly maintain the narrative that Holo Fuel should be treated as an accounting system rather than a currency, it is unnecessary; no real company would allow its reserves to be looted simply because external exchange rates happened to change. Neither will the Holo system.

2.1.3 Active Exchange Flow Balancing

In order to detect and respond to both A) Exchange currency value fluctuations, and B) maintain the desired level of inward/outward flow of Holo Fuel required to maintain value stability, we need to control:

1. The relative exchange activity level between the various exchanges

   Increasing inflow through one Exchange, while increasing outflow through other Exchange(s) indicates a pricing fluctuation in that Exchange currency.

   The premium of the Exchange rate vs. the Book Value should rise/fall, until net inflow/outflow is roughly equivalent through all Exchanges, taking into account their overall relative activity levels. For example, USD$ and EUR$ will be expected to do more volume that ILS$, but sudden increases in activity indicate corrective action (a PD loop response). Over the long term, a primarily PI loop would detect the correct overall relative activity level.

2. The offset between Buy and Sell price

   If a single Buy/Sell price is offered, a certain equilibrium amount of each should occur at that price. However, we do not always want that equilibrium amount. Sometimes, the underlying Holo Fuel credit system wants to see net redemption of Holo Fuel in inflationary times, or net creation during deflation.

   Another control adjust the Buy/Sell differential, attempting to reach the correct equilibrium; a certain target ratio of buying to selling (eg. 3 Holo Fuel bought for every 1 sold), leading to net increase/decrease in Holo Fuel supply.

3. Limits on the absolute buy/sell amounts

   Even when equilibrium inflow between Exchanges is reached, and a target buy/sell ratio is reached, the absolute rates of purchase (creation) or redemption of Holo Fuel must be controlled. We neither wish to empty our Reserve accounts, or allow a sudden inflow of massive amounts of Holo Fuel credit.

   As the amount of transactions increases within Holo Fuel, the capacity for inflow/outflow increases. The size and number of buy/sell transactions allowed on each exchange per unit time will be limited to a certain percentage of the Monetary Velocity of the Holo Fuel system.

2.1.4 Incomplete...