

```

1 // SPDX-License-Identifier: AGPL-3.0-or-later
2 pragma solidity 0.7.5;
3
4 library FullMath {
5     function fullMul(uint256 x, uint256 y) private
6     pure returns (uint256 l, uint256 h) {
7         uint256 mm = mulmod(x, y, uint256(-1));
8         l = x * y;
9         h = mm - l;
10        if (mm < l) h -= 1;
11    }
12
13    function fullDiv(
14        uint256 l,
15        uint256 h,
16        uint256 d
17    ) private pure returns (uint256) {
18        uint256 pow2 = d & -d;
19        d /= pow2;
20        l /= pow2;
21        l += h * ((-pow2) / pow2 + 1);
22        uint256 r = 1;
23        r *= 2 - d * r;
24        r *= 2 - d * r;
25        r *= 2 - d * r;
26        r *= 2 - d * r;
27        r *= 2 - d * r;
28        r *= 2 - d * r;
29        r *= 2 - d * r;
30        return l * r;
31    }
32
33    function mulDiv(
34        uint256 x,
35        uint256 y,
36        uint256 d
37    ) internal pure returns (uint256) {
38        (uint256 l, uint256 h) = fullMul(x, y);
39        uint256 mm = mulmod(x, y, d);
40        if (mm > l) h -= 1;
41        l -= mm;
42        require(h < d, 'FullMath::mulDiv: overflow');
43        return fullDiv(l, h, d);
44    }
45 }
46
47 library Babylonian {
48     function sqrt(uint256 x) internal pure returns
49     (uint256) {
50         if (x == 0) return 0;
51         uint256 xx = x;
52         uint256 r = 1;
53         if (xx >= 0x100000000000000000000000000000000) {
54             xx >= 128;
55             r <= 64;
56         }
57         if (xx >= 0x1000000000000000000) {
58             xx >= 64;

```

```

1 // SPDX-License-Identifier: AGPL-3.0-or-later
2 pragma solidity 0.7.5;
3 //Only change to generate diff
4 library FullMath {
5     function fullMul(uint256 x, uint256 y) private
6     pure returns (uint256 l, uint256 h) {
7         uint256 mm = mulmod(x, y, uint256(-1));
8         l = x * y;
9         h = mm - l;
10        if (mm < l) h -= 1;
11    }
12
13    function fullDiv(
14        uint256 l,
15        uint256 h,
16        uint256 d
17    ) private pure returns (uint256) {
18        uint256 pow2 = d & -d;
19        d /= pow2;
20        l /= pow2;
21        l += h * ((-pow2) / pow2 + 1);
22        uint256 r = 1;
23        r *= 2 - d * r;
24        r *= 2 - d * r;
25        r *= 2 - d * r;
26        r *= 2 - d * r;
27        r *= 2 - d * r;
28        r *= 2 - d * r;
29        r *= 2 - d * r;
30        return l * r;
31    }
32
33    function mulDiv(
34        uint256 x,
35        uint256 y,
36        uint256 d
37    ) internal pure returns (uint256) {
38        (uint256 l, uint256 h) = fullMul(x, y);
39        uint256 mm = mulmod(x, y, d);
40        if (mm > l) h -= 1;
41        l -= mm;
42        require(h < d, 'FullMath::mulDiv: overflow');
43        return fullDiv(l, h, d);
44    }
45 }
46
47 library Babylonian {
48
49     function sqrt(uint256 x) internal pure returns
50     (uint256) {
51         if (x == 0) return 0;
52
53         uint256 xx = x;
54         uint256 r = 1;
55         if (xx >= 0x100000000000000000000000000000000) {
56             xx >= 128;
57             r <= 64;
58         }
59         if (xx >= 0x1000000000000000000) {
60             xx >= 64;

```

```

59         r <= 32;
60     }
61     if (xx >= 0x100000000) {
62         xx >= 32;
63         r <= 16;
64     }
65     if (xx >= 0x10000) {
66         xx >= 16;
67         r <= 8;
68     }
69     if (xx >= 0x100) {
70         xx >= 8;
71         r <= 4;
72     }
73     if (xx >= 0x10) {
74         xx >= 4;
75         r <= 2;
76     }
77     if (xx >= 0x8) {
78         r <= 1;
79     }
80     r = (r + x / r) >> 1;
81     r = (r + x / r) >> 1;
82     r = (r + x / r) >> 1;
83     r = (r + x / r) >> 1;
84     r = (r + x / r) >> 1;
85     r = (r + x / r) >> 1;
86     r = (r + x / r) >> 1; // Seven iterations s
    should be enough
87     uint256 r1 = x / r;
88     return (r < r1 ? r : r1);
89 }
90 }
91
92 library BitMath {
93
94     function mostSignificantBit(uint256 x) internal
    pure returns (uint8 r) {
95         require(x > 0, 'BitMath::mostSignificantBi
    t: zero');
96
97         if (x >= 0x100000000000000000000000000000000
    0) {
98             x >= 128;
99             r += 128;
100         }
101         if (x >= 0x1000000000000000000) {
102             x >= 64;
103             r += 64;
104         }
105         if (x >= 0x100000000) {
106             x >= 32;
107             r += 32;
108         }
109         if (x >= 0x10000) {
110             x >= 16;
111             r += 16;
112         }
113         if (x >= 0x100) {
114             x >= 8;
115             r += 8;
116         }
117         if (x >= 0x10) {
118             x >= 4;
119             r += 4;
120         }
121         if (x >= 0x4) {
122             x >= 2;

```

```

60         r <= 32;
61     }
62     if (xx >= 0x100000000) {
63         xx >= 32;
64         r <= 16;
65     }
66     if (xx >= 0x10000) {
67         xx >= 16;
68         r <= 8;
69     }
70     if (xx >= 0x100) {
71         xx >= 8;
72         r <= 4;
73     }
74     if (xx >= 0x10) {
75         xx >= 4;
76         r <= 2;
77     }
78     if (xx >= 0x8) {
79         r <= 1;
80     }
81     r = (r + x / r) >> 1;
82     r = (r + x / r) >> 1;
83     r = (r + x / r) >> 1;
84     r = (r + x / r) >> 1;
85     r = (r + x / r) >> 1;
86     r = (r + x / r) >> 1;
87     r = (r + x / r) >> 1; // Seven iterations s
    should be enough
88     uint256 r1 = x / r;
89     return (r < r1 ? r : r1);
90 }
91 }
92
93 library BitMath {
94
95     function mostSignificantBit(uint256 x) internal
    pure returns (uint8 r) {
96         require(x > 0, 'BitMath::mostSignificantBi
    t: zero');
97
98         if (x >= 0x100000000000000000000000000000000
    0) {
99             x >= 128;
100             r += 128;
101         }
102         if (x >= 0x1000000000000000000) {
103             x >= 64;
104             r += 64;
105         }
106         if (x >= 0x100000000) {
107             x >= 32;
108             r += 32;
109         }
110         if (x >= 0x10000) {
111             x >= 16;
112             r += 16;
113         }
114         if (x >= 0x100) {
115             x >= 8;
116             r += 8;
117         }
118         if (x >= 0x10) {
119             x >= 4;
120             r += 4;
121         }
122         if (x >= 0x4) {
123             x >= 2;

```

```

123         r += 2;
124     }
125     if (x >= 0x2) r += 1;
126 }
127 }
128
129 library FixedPoint {
130     // range: [0, 2**112 - 1]
131     // resolution: 1 / 2**112
132     struct uq112x112 {
133         uint224 _x;
134     }
135
136     // range: [0, 2**144 - 1]
137     // resolution: 1 / 2**112
138     struct uq144x112 {
139         uint256 _x;
140     }
141
142     uint8 private constant RESOLUTION = 112;
143     uint256 private constant Q112 = 0x1000000000000
0000000000000000;
144     uint256 private constant Q224 = 0x1000000000000
00000000000000000000000000000000000000000000000;
145     uint256 private constant LOWER_MASK = 0xffffffff
ffffffffffffffff; // decimal of UQ*x112 (lower
112 bits)
146
147     // decode a UQ112x112 into a uint112 by truncat
ing after the radix point
148     function decode(uq112x112 memory self) internal
pure returns (uint112) {
149         return uint112(self._x >> RESOLUTION);
150     }
151
152     // decode a uq112x112 into a uint with 18 decim
als of precision
153     function decode112with18(uq112x112 memory self)
internal pure returns (uint) {
154         return uint(self._x) / 5192296858534827;
155     }
156
157     function fraction(uint256 numerator, uint256 de
nominator) internal pure returns (uq112x112 memory)
{
158         require(denominator > 0, 'FixedPoint::fract
ion: division by zero');
159         if (numerator == 0) return FixedPoint.uq112
x112(0);
160
161         if (numerator <= uint144(-1)) {
162             uint256 result = (numerator << RESOLUTI
ON) / denominator;
163             require(result <= uint224(-1), 'FixedPo
int::fraction: overflow');
164             return uq112x112(uint224(result));
165         } else {
166             uint256 result = FullMath.mulDiv(numera
tor, Q112, denominator);
167             require(result <= uint224(-1), 'FixedPo
int::fraction: overflow');
168             return uq112x112(uint224(result));
169         }
170     }
171
172     // square root of a UQ112x112
173     // lossy between 0/1 and 40 bits
174     function sqrt(uq112x112 memory self) internal p
ure returns (uq112x112 memory) {

```

```

124         r += 2;
125     }
126     if (x >= 0x2) r += 1;
127 }
128 }
129
130 library FixedPoint {
131     // range: [0, 2**112 - 1]
132     // resolution: 1 / 2**112
133     struct uq112x112 {
134         uint224 _x;
135     }
136
137     // range: [0, 2**144 - 1]
138     // resolution: 1 / 2**112
139     struct uq144x112 {
140         uint256 _x;
141     }
142
143     uint8 private constant RESOLUTION = 112;
144     uint256 private constant Q112 = 0x1000000000000
0000000000000000;
145     uint256 private constant Q224 = 0x1000000000000
00000000000000000000000000000000000000000000000;
146     uint256 private constant LOWER_MASK = 0xffffffff
ffffffffffffffff; // decimal of UQ*x112 (lower
112 bits)
147
148     // decode a UQ112x112 into a uint112 by truncat
ing after the radix point
149     function decode(uq112x112 memory self) internal
pure returns (uint112) {
150         return uint112(self._x >> RESOLUTION);
151     }
152
153     // decode a uq112x112 into a uint with 18 decim
als of precision
154     function decode112with18(uq112x112 memory self)
internal pure returns (uint) {
155         return uint(self._x) / 5192296858534827;
156     }
157
158     function fraction(uint256 numerator, uint256 de
nominator) internal pure returns (uq112x112 memory)
{
159         require(denominator > 0, 'FixedPoint::fract
ion: division by zero');
160         if (numerator == 0) return FixedPoint.uq112
x112(0);
161
162         if (numerator <= uint144(-1)) {
163             uint256 result = (numerator << RESOLUTI
ON) / denominator;
164             require(result <= uint224(-1), 'FixedPo
int::fraction: overflow');
165             return uq112x112(uint224(result));
166         } else {
167             uint256 result = FullMath.mulDiv(numera
tor, Q112, denominator);
168             require(result <= uint224(-1), 'FixedPo
int::fraction: overflow');
169             return uq112x112(uint224(result));
170         }
171     }
172
173     // square root of a UQ112x112
174     // lossy between 0/1 and 40 bits
175     function sqrt(uq112x112 memory self) internal p
ure returns (uq112x112 memory) {

```

```

175         if (self._x <= uint144(-1)) {
176             return uq112x112(uint224(Babylonian.sqr
t(uint256(self._x) << 112)));
177         }
178
179         uint8 safeShiftBits = 255 - BitMath.mostSig
nificantBit(self._x);
180         safeShiftBits -= safeShiftBits % 2;
181         return uq112x112(uint224(Babylonian.sqrt(ui
nt256(self._x) << safeShiftBits) << ((112 - safeShi
ftBits) / 2)));
182     }
183 }
184
185 library LowGasSafeMath {
186     /// @notice Returns x + y, reverts if sum overf
lows uint256
187     /// @param x The augend
188     /// @param y The addend
189     /// @return z The sum of x and y
190     function add(uint256 x, uint256 y) internal pur
e returns (uint256 z) {
191         require((z = x + y) >= x);
192     }
193
194     function add32(uint32 x, uint32 y) internal pur
e returns (uint32 z) {
195         require((z = x + y) >= x);
196     }
197
198     /// @notice Returns x - y, reverts if underflow
s
199     /// @param x The minuend
200     /// @param y The subtrahend
201     /// @return z The difference of x and y
202     function sub(uint256 x, uint256 y) internal pur
e returns (uint256 z) {
203         require((z = x - y) <= x);
204     }
205
206     function sub32(uint32 x, uint32 y) internal pur
e returns (uint32 z) {
207         require((z = x - y) <= x);
208     }
209
210     /// @notice Returns x * y, reverts if overflows
211     /// @param x The multiplicand
212     /// @param y The multiplier
213     /// @return z The product of x and y
214     function mul(uint256 x, uint256 y) internal pur
e returns (uint256 z) {
215         require(x == 0 || (z = x * y) / x == y);
216     }
217
218     /// @notice Returns x + y, reverts if overflows
or underflows
219     /// @param x The augend
220     /// @param y The addend
221     /// @return z The sum of x and y
222     function add(int256 x, int256 y) internal pure
returns (int256 z) {
223         require((z = x + y) >= x == (y >= 0));
224     }
225
226     /// @notice Returns x - y, reverts if overflows
or underflows
227     /// @param x The minuend
228     /// @param y The subtrahend
229     /// @return z The difference of x and y

```

```

176         if (self._x <= uint144(-1)) {
177             return uq112x112(uint224(Babylonian.sqr
t(uint256(self._x) << 112)));
178         }
179
180         uint8 safeShiftBits = 255 - BitMath.mostSig
nificantBit(self._x);
181         safeShiftBits -= safeShiftBits % 2;
182         return uq112x112(uint224(Babylonian.sqrt(ui
nt256(self._x) << safeShiftBits) << ((112 - safeShi
ftBits) / 2)));
183     }
184 }
185
186 library LowGasSafeMath {
187     /// @notice Returns x + y, reverts if sum overf
lows uint256
188     /// @param x The augend
189     /// @param y The addend
190     /// @return z The sum of x and y
191     function add(uint256 x, uint256 y) internal pur
e returns (uint256 z) {
192         require((z = x + y) >= x);
193     }
194
195     function add32(uint32 x, uint32 y) internal pur
e returns (uint32 z) {
196         require((z = x + y) >= x);
197     }
198
199     /// @notice Returns x - y, reverts if underflow
s
200     /// @param x The minuend
201     /// @param y The subtrahend
202     /// @return z The difference of x and y
203     function sub(uint256 x, uint256 y) internal pur
e returns (uint256 z) {
204         require((z = x - y) <= x);
205     }
206
207     function sub32(uint32 x, uint32 y) internal pur
e returns (uint32 z) {
208         require((z = x - y) <= x);
209     }
210
211     /// @notice Returns x * y, reverts if overflows
212     /// @param x The multiplicand
213     /// @param y The multiplier
214     /// @return z The product of x and y
215     function mul(uint256 x, uint256 y) internal pur
e returns (uint256 z) {
216         require(x == 0 || (z = x * y) / x == y);
217     }
218
219     /// @notice Returns x + y, reverts if overflows
or underflows
220     /// @param x The augend
221     /// @param y The addend
222     /// @return z The sum of x and y
223     function add(int256 x, int256 y) internal pure
returns (int256 z) {
224         require((z = x + y) >= x == (y >= 0));
225     }
226
227     /// @notice Returns x - y, reverts if overflows
or underflows
228     /// @param x The minuend
229     /// @param y The subtrahend
230     /// @return z The difference of x and y

```

```

230     function sub(int256 x, int256 y) internal pure
returns (int256 z) {
231         require((z = x - y) <= x == (y >= 0));
232     }
233
234     function div(uint256 x, uint256 y) internal pur
e returns(uint256 z){
235         require(y > 0);
236         z=x/y;
237     }
238
239     function sqrtt(uint256 a) internal pure returns
(uint c) {
240         if (a > 3) {
241             c = a;
242             uint b = add( div( a, 2), 1 );
243             while (b < c) {
244                 c = b;
245                 b = div( add( div( a, b ), b), 2 );
246             }
247         } else if (a != 0) {
248             c = 1;
249         }
250     }
251 }
252
253 interface IERC20 {
254     function decimals() external view returns (uint
8);
255 }
256
257 interface IUniswapV2ERC20 {
258     function totalSupply() external view returns (u
int);
259 }
260
261 interface IUniswapV2Pair is IUniswapV2ERC20 {
262     function getReserves() external view returns (u
int112 reserve0, uint112 reserve1, uint32 blockTime
stampLast);
263     function token0() external view returns ( addre
ss );
264     function token1() external view returns ( addre
ss );
265 }
266
267 interface IBondingCalculator {
268     function valuation( address pair_, uint amount_ )
external view returns ( uint _value );
269 }
270
271 contract TimeBondingCalculator is IBondingCalculato
r {
272
273     using FixedPoint for *;
274     using LowGasSafeMath for uint;
275     using LowGasSafeMath for uint112;
276
277     IERC20 public immutable Time;
278
279     constructor( address _Time ) {
280         require( _Time != address(0) );
281         Time = IERC20(_Time);
282     }
283
284     function getKValue( address _pair ) public view
returns( uint k_ ) {
285         uint token0 = IERC20( IUniswapV2Pair( _pair
).token0() ).decimals();

```

```

231     function sub(int256 x, int256 y) internal pure
returns (int256 z) {
232         require((z = x - y) <= x == (y >= 0));
233     }
234
235     function div(uint256 x, uint256 y) internal pur
e returns(uint256 z){
236         require(y > 0);
237         z=x/y;
238     }
239
240     function sqrtt(uint256 a) internal pure returns
(uint c) {
241         if (a > 3) {
242             c = a;
243             uint b = add( div( a, 2), 1 );
244             while (b < c) {
245                 c = b;
246                 b = div( add( div( a, b ), b), 2 );
247             }
248         } else if (a != 0) {
249             c = 1;
250         }
251     }
252 }
253
254 interface IERC20 {
255     function decimals() external view returns (uint
8);
256 }
257
258 interface IUniswapV2ERC20 {
259     function totalSupply() external view returns (u
int);
260 }
261
262 interface IUniswapV2Pair is IUniswapV2ERC20 {
263     function getReserves() external view returns (u
int112 reserve0, uint112 reserve1, uint32 blockTime
stampLast);
264     function token0() external view returns ( addre
ss );
265     function token1() external view returns ( addre
ss );
266 }
267
268 interface IBondingCalculator {
269     function valuation( address pair_, uint amount_ )
external view returns ( uint _value );
270 }
271
272 contract TimeBondingCalculator is IBondingCalculato
r {
273
274     using FixedPoint for *;
275     using LowGasSafeMath for uint;
276     using LowGasSafeMath for uint112;
277
278     IERC20 public immutable Time;
279
280     constructor( address _Time ) {
281         require( _Time != address(0) );
282         Time = IERC20(_Time);
283     }
284
285     function getKValue( address _pair ) public view
returns( uint k_ ) {
286         uint token0 = IERC20( IUniswapV2Pair( _pair
).token0() ).decimals();

```

```

286         uint token1 = IERC20( IUniswapV2Pair( _pair
287         ).token1() ).decimals();
287         uint pairDecimals = IERC20( _pair ).decimals();
288
288         (uint reserve0, uint reserve1, ) = IUniswap
289         V2Pair( _pair ).getReserves();
290         if (token0.add(token1) < pairDecimals)
291         {
292             uint decimals = pairDecimals.sub(token
293             0.add(token1));
294             k_ = reserve0.mul(reserve1).mul( 10 **
295             decimals );
296         }
297         else {
298             uint decimals = token0.add(token1).sub
299             (pairDecimals);
300             k_ = reserve0.mul(reserve1).div( 10 **
301             decimals );
302         }
303     }
304     function getTotalValue( address _pair ) public
305     view returns ( uint _value ) {
306         _value = getKValue( _pair ).sqrt().mul(2);
307     }
308     function valuation( address _pair, uint amount_
309     ) external view override returns ( uint _value ) {
310         uint totalValue = getTotalValue( _pair );
311         uint totalSupply = IUniswapV2Pair( _pair ).
312         totalSupply();
313         _value = totalValue.mul( FixedPoint.fraction(
314         amount_, totalSupply ).decode112with18() ).div(
315         1e18 );
316     }
317     function markdown( address _pair ) external view
318     returns ( uint ) {
319         ( uint reserve0, uint reserve1, ) = IUniswapV2Pair( _pair ).getReserves();
320
321         uint reserve;
322         if ( IUniswapV2Pair( _pair ).token0() == address(Time) ) {
323             reserve = reserve1;
324         } else {
325             require(IUniswapV2Pair( _pair ).token1
326             () == address(Time), "not a Time lp pair");
327             reserve = reserve0;
328         }
329         return reserve.mul( 2 * ( 10 ** Time.decimals() ) ).div( getTotalValue( _pair ) );
330     }
331 }
332

```

```

287         uint token1 = IERC20( IUniswapV2Pair( _pair
288         ).token1() ).decimals();
289         uint pairDecimals = IERC20( _pair ).decimals();
290
290         (uint reserve0, uint reserve1, ) = IUniswap
291         V2Pair( _pair ).getReserves();
292         if (token0.add(token1) < pairDecimals)
293         {
294             uint decimals = pairDecimals.sub(token
295             0.add(token1));
296             k_ = reserve0.mul(reserve1).mul( 10 **
297             decimals );
298         }
299         else {
300             uint decimals = token0.add(token1).sub
301             (pairDecimals);
302             k_ = reserve0.mul(reserve1).div( 10 **
303             decimals );
304         }
305     }
306     function getTotalValue( address _pair ) public
307     view returns ( uint _value ) {
308         _value = getKValue( _pair ).sqrt().mul(2);
309     }
310     function valuation( address _pair, uint amount_
311     ) external view override returns ( uint _value ) {
312         uint totalValue = getTotalValue( _pair );
313         uint totalSupply = IUniswapV2Pair( _pair ).
314         totalSupply();
315         _value = totalValue.mul( FixedPoint.fraction(
316         amount_, totalSupply ).decode112with18() ).div(
317         1e18 );
318     }
319     function markdown( address _pair ) external view
320     returns ( uint ) {
321         ( uint reserve0, uint reserve1, ) = IUniswapV2Pair( _pair ).getReserves();
322
323         uint reserve;
324         if ( IUniswapV2Pair( _pair ).token0() == address(Time) ) {
325             reserve = reserve1;
326         } else {
327             require(IUniswapV2Pair( _pair ).token1
328             () == address(Time), "not a Time lp pair");
329             reserve = reserve0;
330         }
331         return reserve.mul( 2 * ( 10 ** Time.decimals() ) ).div( getTotalValue( _pair ) );
332     }
333 }
334

```