Consider the following price function for a bonding curve contract:

f(x

)=1/(1-z

)-1 where z

=x

/maximalAmount

It has all sorts of nice properties, such as a hard cap on the issuance of the asset without capping the amount of money it can take (since the integral of 1/x diverges to infinity), the leverage ratio converging to infinity, i.e. the ratio tied-up liquidity to market cap converges to zero as the market cap grows to infinity (and it can grow to infinity), etc.

However, if we want to implement it in a smart contract, be it EVM or eWASM based, we get into some problems. If the currently issued amount is a

and we wish to issue b

, how much do we need to pay for it? Well, the integral of f(x

) between a and a

+b

, which is F(a

+b

) - F(a

), F(x

) being the primitive function of f(x

), which in our case is -ln(1-z

), so the amount of money we need to pay to get b

tokens is ln(c

)-ln(c

-b

/maximalAmount

) where c

= 1-a

/maximalAmount

. Calculating the natural logarithm of a number on a binary computer is easiest by calculating the base 2 logarithm and multiplying by ln(2). The integer part of the base 2 logarithm is calculated by the position of the most significant set bit. Each subsequent bit of the fractional part is calculated by first shifting the number into a position where the most significant bit is just before the fractional point, squaring the number, and checking if the result is at least 2.

What if we specify the amount we pay? Let’s say p

. Solving p

= ln(c

)-ln(c

-b

/maximalAmount

) for b

we get b

= maximalAmount

- (1 - 1/exp(p

)) * c

. This is a bit easier, because exp(p

) can be calculated by multiplying the powers of e

corresponding to the set bits of p

. The squares and square roots of e

can be pre-calculated as part of the contract code.

Maybe it is even cheaper to supply both p

and q

and some witness that they indeed correspond to each other.