Threshold Calculation Spec

This document details the numerical technique used to calculate the eligibility threshold in the staking procedure of Tetra. Taktikos allows difficulty curves to be defined with values that range from 0 to 1 in the staking threshold function phi = 1-(1-f)^alpha where alpha is the relative stake and f is the difficulty. The previous implementation in Prosomo had used a Taylor series approximation that worked well for values for f of less than about a half but started to diverge from the analytic value as f approached 1. This spurred some testing and we found that the number of terms required for the domain approaching f = 1 had to be ramped up significantly to get an accurate threshold. Clearly, a better numerical method was required and that's what is covered here.

The threshold function has an exponential term in it that needs to be evaluated at runtime by forgers and validators. The exponential form can be manipulated to take as an argument the relative stake multiplied by a base coefficient that defines the base of the exponent. The numerical evaluation of the exponentiation operation takes as an argument the coefficient*alpha where coefficient = ln(1-f). The numerical exponentiation has as a base the natural number e, such that the evaluation reads e^(coefficient*alpha). The coefficient can be calculated before the protocol execution, effectively defining each base coefficient in terms of the difficulty curve. This precompute step is no different than any other global setup and is the same as defining a given difficulty curve for a given protocol. However, if the difficulty curve adjusts in time that would necessitate the recalculation of the base coefficients as a protocol step.

Modified Lentz's Method

This method is covered in "Numerical Recipes in Fortran 77" Second Edition Section 5.2:

- William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. 1992. Numerical recipes in FORTRAN (2nd ed.): the art of scientific computing. Cambridge University Press, USA.

To enable fast calculation of the exponentiation and logarithm operations, we need a method that converges quickly, doesn't use floating-point arithmetic, and can be taken to a predetermined degree of precision. Lentz's method satisfies all of these requirements. Continued fractions typically converge more quickly than power series expansions. Any continued fraction of the following form may be evaluated using Lentz's method:

$$f(x) = b_0 + \cfrac{a_1}{b_1 + \cfrac{a_2}{b_2 + \cfrac{a_3}{b_3 + \cfrac{a_4}{b_4 + \cfrac{a_5}{b_5 + \cdots}}}}}$$

The two expressions we are interested in are:

$$e^x = \cfrac{1}{1 - \cfrac{x}{1 + x - \cfrac{x}{2 + x - \cfrac{2x}{3 + x - \cfrac{3x}{4 + x - \ddots}}}}}$$

$$\log(1+x) = \cfrac{x}{1 + \cfrac{x}{2 - x + \cfrac{2^2 x}{3 - 2x + \cfrac{3^2 x}{4 - 3x + \ddots}}}}$$

The coefficients for exp (a’s and b’s) are defined in the following Scala code:

```
/**
* Returns the exponent base natural number of the argument
* @param x argument
* @param maxIter maximum number of iterations
* @param prec desired precision
* @return exp(x)
*/
def exp(x:Ratio, maxIter:Int, prec:Long):(Ratio,Boolean,Int) = {
  def a(j:Int):Ratio = j match {
    case 0 => Ratio(0)
    case 1 => Ratio(1)
    case 2 => Ratio(-1)*x
    case _ => Ratio(-j+2)*x
  }
  def b(j:Int):Ratio = j match {
    case 0 => Ratio(0)
    case 1 => Ratio(1)
    case _ => Ratio(j-1)+x
  }
  modified_lentz_method(maxIter,prec,a,b)
}
```

The following defines the coefficients for log(1+x):

```
/**
* Returns the natural logarithm of the argument plus one
* @param x argument
* @param maxIter maximum number of iterations
* @param prec desired precision
* @return log_e(1+x)
*/
def log1p(x:Ratio,maxIter:Int,prec:Long):(Ratio,Boolean,Int) = {
  def a(j:Int):Ratio = j match {
    case 0 => Ratio(0)
    case 1 => x
    case _ => Ratio(j-1)*Ratio(j-1)*x
  }
  def b(j:Int):Ratio = j match {
    case 0 => Ratio(0)
    case 1 => Ratio(1)
    case _ => Ratio(j) - Ratio(j-1)*x
  }
  modified_lentz_method(maxIter,prec,a,b)
}
```

The underlying implementation of Lentz’s method using ratios is given below. The specification in Numerical Recipes assumes floating-point arithmetic but is easily adapted to rational expression evaluation. The iterative loop produces rational values with increasing size in terms of the combined number of digits in their numerator and denominator. To mitigate this a limiter routine is incorporated that adjusts the size in bytes of the rational result. There is a slight tradeoff in precision depending on the maximum byte size of the allowed ratios. This also prevents the algorithm from diverging in computation time with a maliciously chosen argument, the hypothesis being that an adversary may choose some value of alpha that causes a significant increase in the cost of validation.

```
   /**
   * The numerical technique uses a set of coefficients to calculate a 
nested fraction iteratively,
   * avoiding nested recursion and providing a much more performant 
algorithm
   * @param maxIter maximum number of iterations
   * @param prec desired precision
   * @param a a coefficients that map integers to ratios
   * @param b b coefficients that map integers to ratios
   * @param limit limit the size of the numerator and denominator
   * @param limitBytes maximum size of the byte length of the resultant 
ratio
   * @return a tuple containing:
   *         ratio approximating the nested fraction,
   *         false if method converged true otherwise,
   *         number of iterations
   */

def modified_lentz_method(
                           maxIter:Int,
                           prec:Long,
                           a: Int => Ratio,
                           b:Int => Ratio,
                           limit:Boolean = 
true,
                           limitBytes:Int = 
8192
                           ): (Ratio,Boolean,Int) = {
  val tesimal = Ratio(1,2*prec)
  val eps:Ratio = Ratio(1,prec)
  var fj:Ratio = {
    if (b(0)==Ratio(0)) tesimal
    else b(0)
  }
  var cj:Ratio = fj
  var dj:Ratio = Ratio(0)
  var deltaj = Ratio(1)
  var diff = Ratio(1)
  var error:Boolean = true  def loop(j:Int):Unit = {
    dj = b(j) + a(j) * dj
    if (dj == Ratio(0)) dj = tesimal
    cj = b(j) + a(j)/cj
    if (cj == Ratio(0)) cj = tesimal
    dj = Ratio(dj.denominator,dj.numerator)
    deltaj = cj*dj
    if (limit) {
      if (fj.numerator.toByteArray.length + fj.denominator.toByteArray.
length > limitBytes) {
        fj = reduce(fj,100*prec,100)
      }
    }
    val fjm1 = fj
    fj = fj*deltaj
    diff = fj - fjm1
    error = j match {
      case _ if j > 1 => diff.abs > eps
      case _ => true    }
  }
  var j = 1  while (j < maxIter+1 && error) {
    loop(j)
    j = j+1  }
  if (fj.denominator<0) fj = Ratio(-fj.numerator,-fj.denominator)
  (fj,error,j)
}
```

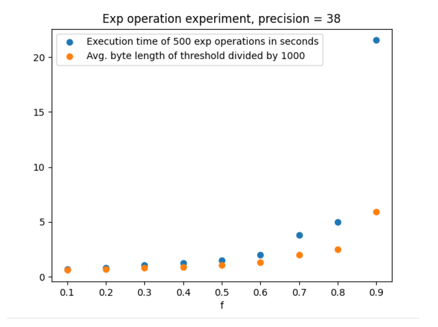
The following algorithm provides a good rational approximation of ratios with large denominators:

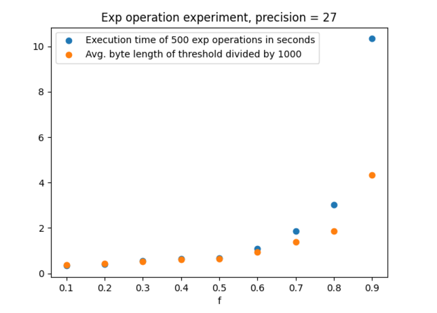
```
  /**
   * Implmentation of Farey approximation technique following this blog
post
   * https://www.johndcook.com/blog/2010/10/20/best-rational-
approximation/
   * This method reduces the byte size of ratios by finding an
   * approximate answer with a smaller numerator and denominator.
   * Returns a ratio with a denominator no greater than maxDenominator
that approximates the input ratio.
   * The approximation gets better with number of iterations.
   * I've modified the algorithm to work with numbers greater than 1
and all negative numbers
   * @param input ratio to be reduced in size
   * @param maxDenominator maximum resultant denominator
   * @param maxIter maximum number of iterations
   * @return a new ratio with smaller integer values that approximates
the input
   */
def reduce(input:Ratio, maxDenominator:Long, maxIter:Int):Ratio = {
  var x = input
  var output = input
  if (x.denominator<0) x = Ratio(-x.numerator,-x.denominator)
  val q:BigInt = x.numerator/x.denominator
  val r:BigInt = x.numerator%x.denominator
  val sign = if (x.numerator > 0 && x.denominator > 0) {
    1  } else {
    -1  }
  val absx = Ratio(r,x.denominator).abs
  var a = BigInt(0)
  var b = BigInt(1)
  var c = BigInt(1)
  var d = BigInt(1)
  val maxDenom = BigInt(maxDenominator)
  var j = 0  while (b<=maxDenom && d <= maxDenom && j<=maxIter) {
    val med = Ratio(a+c,b+d)
    if (absx == med) {
      if (b+d <=maxDenom) {
        output = Ratio(sign*(a+c),b+d)
      } else if (d>b) {
        output = Ratio(sign*c,d)
      } else {
        output = Ratio(sign*a,b)
      }
    } else if (absx > med) {
      a = a+c
      b = b+d
    } else {
      c = a+c
      d = b+d
    }
    j = j+1  }
  if (b>maxDenom) {
    output = Ratio(sign*c,d)
  } else {
    output = Ratio(sign*a,b)
  }
  output + Ratio(q)
}
```

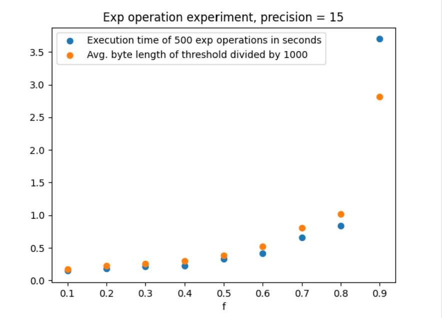
Numerical Results

Here I'll give results comparing outputs for given inputs. The precision in these routines is subject to a truncation error induced by stopping the iteration at a given threshold, either with a given tolerance or max number of iterations. We'll consider the Scala math package result computed with math.pow(1-f) as reference values to assess the accuracy of the Lentz method output. The precision between the two methods cannot be adequately compared because the Scala math power method is subject to roundoff errors and is limited by the floating-point precision. The Numerical Recipes algorithm has a truncation method designed for floating-point arithmetic that I've adapted to work with rationals.

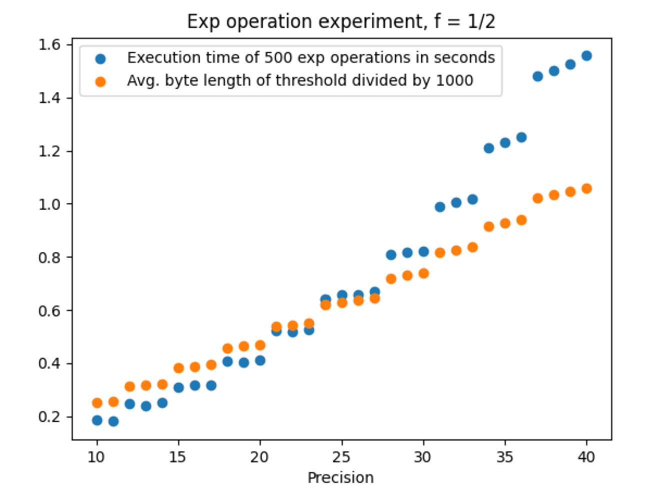
Experiments can be performed with a reference Python implementation of Lentz's method that will help identify performance characteristics of exp operation evaluation. Outputs between the Scala and Python implementations are identical but execution time in Python is easier to evaluate.

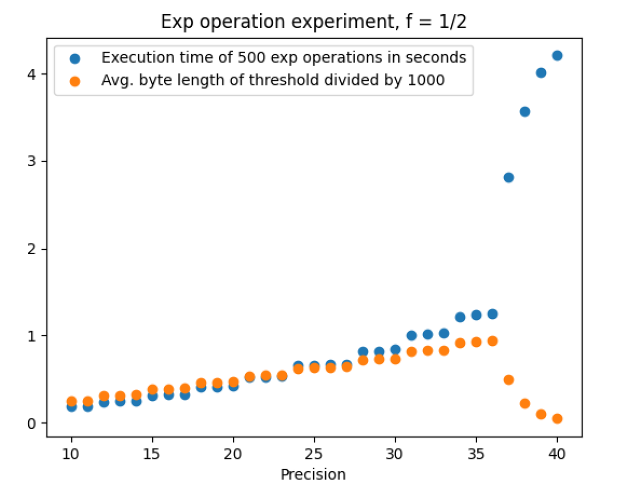




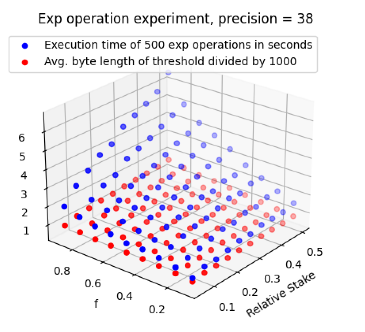


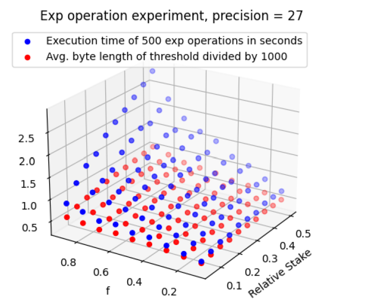
The above 3 plots each have 1/100 of the net stake as an argument in the exp operation and the total stake is set to 10^precision for each trial. To get a statistical average of byte length, each exp operation has a distinct value of stake given by 1/100 of the net stake minus the trial number. This gives a unique threshold with each trial while maintaining comparable execution time between each exp evaluation. There is a clear trend in execution time and byte length that scales with precision.

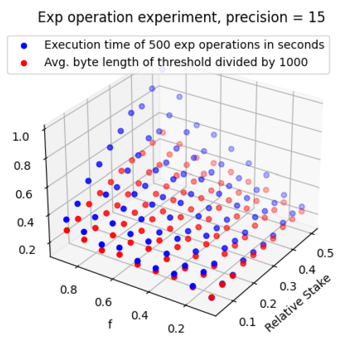


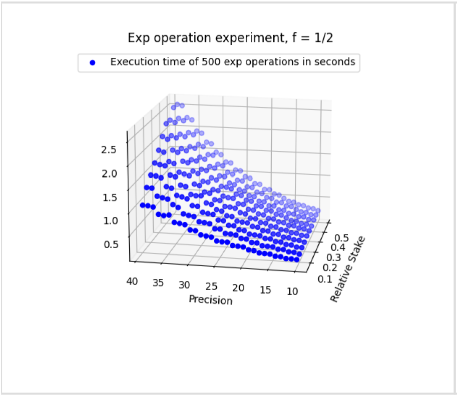


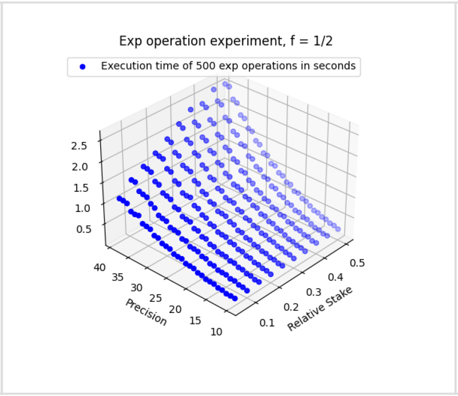
The above experiment (left) gives the execution time with respect to precision. The byte length closely correlates with the execution time. The second plot (right) shows the same regression but with the threshold size limiting routine turned on at 1024 bytes. We can see the execution time diverges as the size of the threshold decreases. The limiter routine doesn't perform well enough to justify usage at high precision.

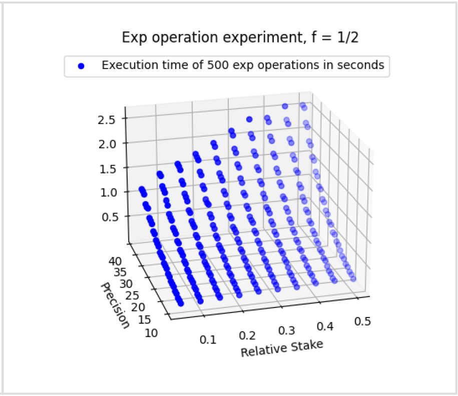




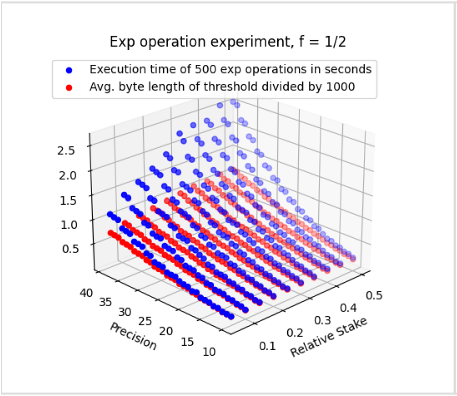


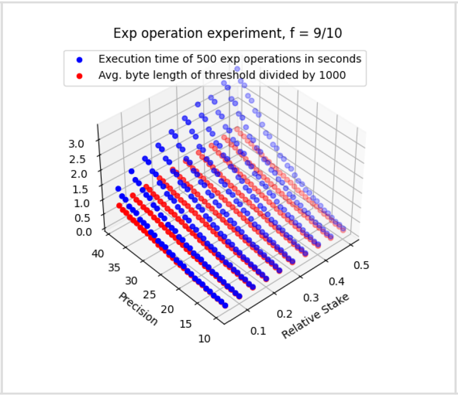


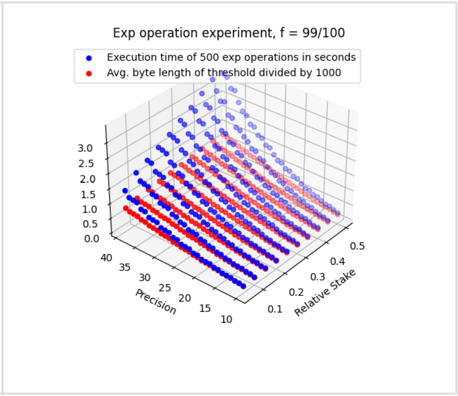


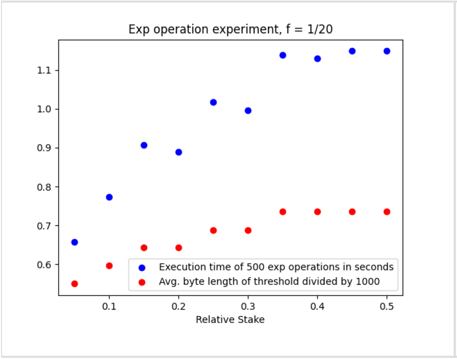


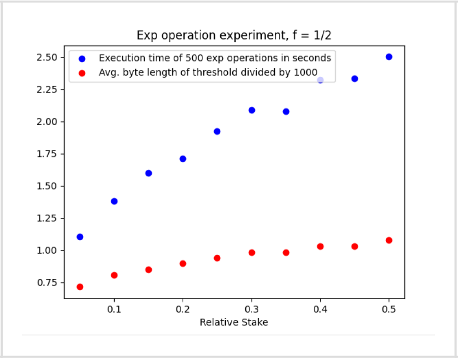
The above plots show a series of parameter sweeps for the active slots coefficient and the precision with respect to relative stake. These trends suggest exponential scaling of execution time with respect to precision and linear scaling with respect to relative stake.

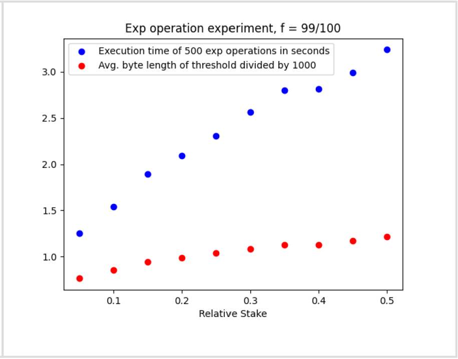












To estimate the overhead, assume an n node setup with equal stake across all nodes. The nodes have to evaluate n operations of threshold evaluation times D the number of difficulty curve states per epoch. Assuming an effective f = 1/2 and a difficulty curve with D states, the number of threshold evaluations would be on the order of n * D per epoch. With serial execution, this would translate to about 1.1 seconds * n * D / 500 operations.

Test Vectors

The following Python 3 code stands as the reference implementation of Lentz's method:

```python
from fractions import Fraction

def modified_lentz_method(max_iter: int, prec: int, a, b):
    big_factor = pow(10, prec + 10)
    tiny_factor = Fraction(1, big_factor)
    truncation_error = Fraction(1, pow(10, prec + 1))
    fj = b(0)
    if fj == Fraction(0):
        fj = tiny_factor
    cj = fj
    dj = Fraction(0)
    error = True
    iii = 1
    while iii < max_iter + 1 and error:
        dj = b(iii) + a(iii) * dj
        if dj == Fraction(0):
            dj = tiny_factor
        cj = b(iii) + a(iii) / cj
        if cj == Fraction(0):
            cj = tiny_factor
        dj = Fraction(dj.denominator, dj.numerator)
        deltaj = cj * dj
        fj = fj * deltaj
        if iii > 1:
            error = (deltaj - Fraction(1)).__abs__() > truncation_error
        iii = iii + 1
    return fj
```

```
def exp(x: Fraction, max_iter: int, prec: int):
    def a(j: int):
        if j == 0:
            return Fraction(0)
        elif j == 1:
            return Fraction(1)
        elif j == 2:
            return Fraction(-1) * x
        else:
            return Fraction(-j + 2) * x

    def b(j: int):
        if j == 0:
            return Fraction(0)
        elif j == 1:
            return Fraction(1)
        else:
            return Fraction(j - 1) + x

    if x == Fraction(0):
        return Fraction(1)
    else:
        return modified_lentz_method(max_iter, prec, a, b)
```

```python
def log1p(x: Fraction, max_iter: int, prec: int):
    def a(j: int):
        if j == 0:
            return Fraction(0)
        elif j == 1:
            return x
        else:
            return Fraction(j - 1) * Fraction(j - 1) * x

    def b(j: int):
        if j == 0:
            return Fraction(0)
        elif j == 1:
            return Fraction(1)
        else:
            return Fraction(j) - Fraction(j - 1) * x

    if x == Fraction(0):
        return Fraction(1)
    else:
        return modified_lentz_method(max_iter, prec, a, b)
```

```
def rational_approximation(arg: Fraction, max_denominator: int,
max_iter: int):
    q = arg.numerator // arg.denominator
    r = arg.numerator % arg.denominator
    if arg < 0:
        sign = -1
    else:
        sign = 1
    x = Fraction(r, arg.denominator).__abs__()
    a = 0
    b = 1
    c = 1
    d = 1
    j = 0
    not_done = True
    output = Fraction(0)
    while b <= max_denominator and d <= max_denominator and j <=
max_iter and not_done:
        med = Fraction(a + c, b + d)
        if x == med:
            if b + d <= max_denominator:
                output = Fraction(sign * (a + c), b + d)
            elif d > b:
                output = Fraction(sign * c, d)
            else:
                output = Fraction(sign * a, b)
            not_done = False
        elif x > med:
            a = a + c
            b = b + d
        else:
            c = a + c
            d = b + d
        j = j + 1
    if not_done:
        if b > max_denominator:
            output = Fraction(sign * c, d)
        else:
            output = Fraction(sign * a, b)
    return Fraction(q * output.denominator + output.numerator, output.
denominator)
```

Test vectors are produced with the following code:

```
def print_frac(frac: Fraction):
    print(float(frac.numerator) / float(frac.denominator))

res = exp(Fraction(1), 10000, 38)
print(math.e)
print_frac(res)
print(res.__str__())
print(rational_approximation(res, 100000, 10000).__str__())
print((log1p(Fraction(-1, 2), 10000, 38)).__str__())
```

The ouput:

```
2.718281828459045
2.7182818284590455
16741996018770930131393432724658409631834030151373346536139479961949388177
/61590361394799619493881770000000000000000000000000000000000000000000
49171/18089
-1376717581725210806236771084653234861246883840172888672612535907978222457208968281235594169021560465023149349862853
205651
/1986183627859607825534949681982241794324288474463155455659529989441716224000000000000000000000000000000000000000
000000
```