Computation of Special Functions
(Haskell)

Apollo Hogan

November 29, 2019

Contents

1 Introduction

Special functions.

2 Utility

2.1 Preamble

We start with the basic preamble.

```
{-# Language BangPatterns #-}
{-# Language FlexibleContexts #-}
{-# Language FlexibleInstances #-}
{-# Language ScopedTypeVariables #-}
{-# Language TypeFamilies #-}
-- {-# Language UndecidableSuperClasses #-}
-- {-# Language UndecidableInstances #-}
module Util where
import Data.Complex
import Data.List (zipWith5)
```

2.2 Data Types

We start by defining a convenient type synonym for complex numbers over `Double`.

```
type CDouble = Complex Double
```

Next, we define the `Value` typeclass which is useful for defining our special functions to work over both real (`Double`) values and over complex (`CDouble`) values with uniform implementations. This will also make it convenient for handling `Quad` values (later).

`class Value v` — Value

```
class (Eq v, Floating v, Fractional v, Num v,
       Enum (RealKind v), Eq (RealKind v), Floating (RealKind v),
         Fractional (RealKind v), Num (RealKind v), Ord (RealKind v),
         RealFrac (RealKind v),
       Eq (ComplexKind v), Floating (ComplexKind v), Fractional (ComplexKind v),
         Num (ComplexKind v)
      ) ⇒ Value v where
  type RealKind v :: *
  type ComplexKind v :: *
  pos_infty :: v
  neg_infty :: v
  nan :: v
  re :: v → (RealKind v)
  im :: v → (RealKind v)
  rabs :: v → (RealKind v)
  is_inf :: v → Bool
  is_nan :: v → Bool
  is_real :: v → Bool
  fromDouble :: Double → v
  fromReal :: (RealKind v) → v
  toComplex :: v → (ComplexKind v)
```

Both `Double` and `CDouble` are instances of the `Value` typeclass in the obvious ways.

`instance Value Double` — Value Double

```
instance Value Double where
  type RealKind Double = Double
  type ComplexKind Double = CDouble
  pos_infty = 1.0/0.0
  neg_infty = −1.0/0.0
  nan = 0.0/0.0
  re = id
  im = const 0
  rabs = abs
  is_inf = isInfinite
  is_nan = isNaN
  is_real _ = True
  fromDouble = id
  fromReal = id
  toComplex x = x :+ 0
```

```
instance Value CDouble
```

Value CDouble

```
instance Value CDouble where
  type RealKind CDouble = Double
  type ComplexKind CDouble = CDouble
  pos_infty = (1.0/0.0) :+ 0
  neg_infty = (−1.0/0.0) :+ 0
  nan = (0.0/0.0) :+ 0
  re = realPart
  im = imagPart
  rabs = realPart.abs
  is_inf z = (is_inf.re$z) ∨ (is_inf.im$z)
  is_nan z = (is_nan.re$z) ∨ (is_nan.im$z)
  is_real _ = False
  fromDouble x = x :+ 0
  fromReal x = x :+ 0
  toComplex = id
```

TODO: add quad versions also

2.3 Helper functions

A convenient shortcut, as we often find ourselves converting indices (or other integral values) to our computation type.

```
{-# INLINE (#) #-}
(#) :: (Integral a, Num b) ⇒ a → b
(#) = fromIntegral
```

A version of `iterate` which passes along an index also (very useful for computing terms of a power-series, for example.)

```
ixiter i x f
```

ixiter

```
{-# INLINE ixiter #-}
ixiter :: (Enum ix) ⇒ ix → a → (ix→a→a) → [a]
ixiter i x f = x:(ixiter (succ i) (f i x) f)
```

Computes the relative error in terms of decimal digits, handy for testing. Note that this fails when the exact value is zero.

$$\texttt{relerr e a} = \log_{10}\left|\frac{a-e}{e}\right|$$

```
relerr :: ∀ v.(Value v) ⇒ v → v → (RealKind v)
relerr !exact !approx = re $! logBase 10 (abs ((approx−exact)/exact))
```

2.4 Kahan summation

A useful tool is so-called Kahan summation, based on the observation that in floating-point arithmetic, one can . . .

Here `kadd t s e k` is a single step of addition, adding a term to a sum+error and passing the updated sum+error to the continuation.

```
-- kadd value oldsum olderr ---> newsum newerr
{-# INLINE kadd #-}
{-# SPECIALISE kadd :: Double -> Double -> Double -> (Double -> Double -> a) -> a #-}
kadd :: (Value v) => v -> v -> v -> (v -> v -> a) -> a
kadd t s e k =
  let y = t - e
      s' = s + y
      e' = (s' - s) - y
  in k s' e'
```

Here `ksum terms` sums a list with Kahan summation. The list is assumed to be (eventually) decreasing and the summation is terminated as soon as adding a term doesn't change the value. (Thus any zeros in the list will immediately terminate the sum.) This is typically used for power-series or asymptotic expansions. (TODO: make generic over stopping condition)

ksum terms

ksum

```
{-# SPECIALISE ksum :: [Double] -> Double #-}
{-# SPECIALISE ksum' :: [Double] -> (Double -> Double -> a) -> a #-}
ksum :: (Value v) => [v] -> v
ksum terms = ksum' terms const

ksum' :: (Value v) => [v] -> (v -> v -> a) -> a
ksum' terms k = f 0 0 terms
  where
    f !s !e [] = k s e
    f !s !e (t:terms) =
      let !y  = t - e
          !s' = s + y
          !e' = (s' - s) - y
      in if s' == s
         then k s' e'
         else f s' e' terms
```

2.5 Continued fraction evaluation

Given two sequences $\{a_n\}_{n=1}^{\infty}$ and $\{b_n\}_{n=0}^{\infty}$ we have the continued fraction

$$b_0 + \cfrac{a_1}{b_1 + \cfrac{a_2}{b_2 + \cfrac{a_3}{b_3 + \cfrac{a_4}{b_4 + \cdots}}}}$$

or

$$b_0 + a_1/(b_1 + a_2/(b_2 + a_3/(b_3 + a_4/(b_4 + \cdots))))$$

though for typesetting purposes this is often written

$$b_0 + \frac{a_1}{b_1 +}\;\frac{a_2}{b_2 +}\;\frac{a_3}{b_3 +}\;\frac{a_4}{b_4 + \cdots}$$

We conventionally notate the n'th approximant or convergent as

$$C_n = b_0 + \frac{a_1}{b_1 +}\;\frac{a_2}{b_2 +}\;\frac{a_3}{b_3 + \dots}\;\frac{a_n}{b_n}$$

2.5.1 Backwards recurrence algorithm

We can compute the n'th convergent C_n for a predetermined n by evaluating

$$u_k = b_k + \frac{a_{k+1}}{u_{k+1}}$$

for $k = n-1, n-2, \ldots, 0$, with $u_n = b_n$. Then $u_0 = C_n$.

sf_cf_back

```
sf_cf_back :: ∀ v.(Value v) ⇒ Int → [v] → [v] → v
sf_cf_back !n !as !bs =
  let !an = reverse $ take n as
      !(un:bn) = reverse $ take (n+1) bs
  in go un an bn
  where
    go :: v → [v] → [v] → v
    go !ukp1 ![] ![] = ukp1
    go !ukp1 !(a:an) !(b:bn) =
        let uk = b + a/ukp1
        in go uk an bn
```

2.5.2 Steed's algorithm

This is Steed's algorithm for evaluation of a continued fraction It evaluates the partial convergents C_n in a forward direction. This implementation will evaluate until $C_n = C_{n+1}$. TODO: describe algorithm.

sf_cf_steeds

```
sf_cf_steeds :: (Value v) ⇒ [v] → [v] → v
sf_cf_steeds (a1:as) (b0:b1:bs) =
    let !c0 = b0
        !d1 = 1/b1
        !delc1 = a1*d1
        !c1 = c0 + delc1
    in recur c1 delc1 d1 as bs
    where recur !cn_1 !delcn_1 !dn_1 !(an:as) !(bn:bs) =
            let !dn = 1/(dn_1*an+bn)
                !delcn = (bn*dn - 1)*delcn_1
                !cn = cn_1 + delcn
            in if (cn == cn_1) ∨ is_nan cn then cn else (recur cn delcn dn as bs)
```

2.5.3 Modified Lentz algorithm

An alternative algorithm for evaluating a continued fraction in a forward directions. This algorithm can be less susceptible to contamination from rounding errors. TODO: describe algorithm

sf_cf_lentz

```
sf_cf_lentz :: (Value v) ⇒ [v] → [v] → v
sf_cf_lentz as (b0:bs) =
  let !c0 = nz b0
      !e0 = c0
      !d0 = 0
  in iter c0 d0 e0 as bs
  where
    !zeta = 1e−100
    nz !x = if x==0 then zeta else x
```

sf_cf_lentz (cont)

```
    iter cn dn en (an:as) (bn:bs) =
      let !idn = nz $ bn + an*dn
          !en' = nz $ bn + an/en
          !dn' = 1 / idn
          !hn  = en' * dn'
          !cn' = cn * hn
          !delta = rabs(hn - 1)
      in if cn==cn' ∨ delta<5e-16
         then cn
         else iter cn' dn' en' as bs
```

2.6 Solving ODEs

2.6.1 Runge-Kutta IV

Solve a system of first-order ODEs using the Runge-Kutta IV method. To solve $\mathbf{y}' = \mathbf{f}(t, \mathbf{y})$ from $t = t_0$ to $t = t_n$ with initial condition $\mathbf{y}(t_0) = \mathbf{y}_0$, first choose a step-size $h > 0$. Then iteratively proceed by letting

$$\begin{aligned}
\mathbf{k}_1 &= h\mathbf{f}(t_i, \mathbf{y}_i) \\
\mathbf{k}_2 &= h\mathbf{f}(t_i + \frac{h}{2}, \mathbf{y}_i + \frac{1}{2}\mathbf{k}_1) \\
\mathbf{k}_3 &= h\mathbf{f}(t_i + \frac{h}{2}, \mathbf{y}_i + \frac{1}{2}\mathbf{k}_2) \\
\mathbf{k}_4 &= h\mathbf{f}(t_i + h, \mathbf{y}_i + \mathbf{k}_3)
\end{aligned}$$

and then

$$\begin{aligned}
t_{i+1} &= t_i + h \\
\mathbf{y}_{i+1} &= \mathbf{y}_i + \frac{1}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4)
\end{aligned}$$

sf_runge_kutta_4

```
sf_runge_kutta_4 :: ∀ v.(Value v) ⇒
    (RealKind v) → (RealKind v) → (RealKind v) → [v] → ((RealKind v)→[v]→[v]) →
[(RealKind v,[v])]
sf_runge_kutta_4 !h !t0 !tn !x0 !f = iter t0 x0 [(t0,x0)]
  where
    iter :: (RealKind v) → [v] → [(RealKind v,[v])] → [(RealKind v,[v])]
    iter !ti !xi !path
      | ti≥tn     = path
      | otherwise =
          let !h' = (min h (tn-ti))
              !h'2 = h'/2
              !h'' = fromReal h'
              !k1 = fmap (h''*) (f ti xi)
              !k2 = fmap (h''*) (f (ti+h'2) (zipWith (λx k→x+k/2) xi k1))
              !k3 = fmap (h''*) (f (ti+h'2) (zipWith (λx k→x+k/2) xi k2))
              !k4 = fmap (h''*) (f (ti+h' ) (zipWith (λx k→x+k  ) xi k3))
              !ti1 = ti + h'
              !xi1 = zipWith5 (λx k1 k2 k3 k4 → x + (k1+2*k2+2*k3+k4)/6) xi k1 k2 k3 k4
          in iter ti1 xi1 ((ti1,xi1):path)
```

2.7 TO BE MOVED

```
sf_sqrt :: (Value v) => v -> v
sf_sqrt = sqrt
```

3 Fibonacci Numbers

A silly approach to efficient computation of Fibonacci numbers

$$f_n = f_{n-1} + f_{n-2} \qquad f_0 = 0 \qquad f_1 = 1$$

The idea is to use the closed-form solution:

$$f_n = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n + \frac{-1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$$

and note that we can work in $\mathbb{Q}[\sqrt{5}]$ with terms of the form $a + b\sqrt{5}$ with $a, b \in \mathbb{Q}$ (notice that $\frac{1}{\sqrt{5}} = \frac{\sqrt{5}}{5}$.)

$$\begin{aligned}(a + b\sqrt{5}) + (c + d\sqrt{5}) &= (a + c) + (b + d)\sqrt{5} \\ (a + b\sqrt{5}) * (c + d\sqrt{5}) &= (ac + 5bd) + (ad + bc)\sqrt{5}\end{aligned}$$

We use the `Rational` type to represent elements of $\mathbb{Q}$, which is a bit more than we actually need, as in the computations above the denominator of $\left(\frac{1\pm\sqrt{5}}{2}\right)^n$ is always, in fact, 1 or 2.

```
module Fibo (fibonacci) where
import Data.Ratio
data Q5 = Q5 Rational Rational
  deriving (Eq)
```

The number-theoretic norm $N(a + b\sqrt{5}) = a^2 - 5b^2$, though unused in our application.

```
norm (Q5 ra qa) = ra^2-5*qa^2
```

Human-friendly `Show` instantiation.

```
instance Show Q5 where
  show (Q5 ra qa) = (show ra)++"+"++(show qa)++"*sqrt(5)"
```

Implementation of the operations for typeclasses `Num` and `Fractional`. The `abs` and `signum` functions are unused, so we just give placeholder values.

```
instance Num Q5 where
  (Q5 ra qa) + (Q5 rb qb) = Q5 (ra+rb) (qa+qb)
  (Q5 ra qa) - (Q5 rb qb) = Q5 (ra-rb) (qa-qb)
  (Q5 ra qa) * (Q5 rb qb) = Q5 (ra*rb+5*qa*qb) (ra*qb+rb*qa)
  negate (Q5 ra qa) = Q5 (-ra) (-qa)
  abs a = Q5 (norm a) 0
  signum a@(Q5 ra qa) = if a==0 then 0 else Q5 (ra/(norm a)) (qa/(norm a))
  fromInteger n = Q5 (fromInteger n) 0

instance Fractional Q5 where
  recip a@(Q5 ra qa) = Q5 (ra/(norm a)) (-qa/(norm a))
  fromRational r = (Q5 r 0)
```

Finally, we define $\phi_\pm = \frac{1}{2}(1 \pm \sqrt{5})$ and $c_\pm = \pm\frac{1}{5}\sqrt{5}$ so that $f_n = c_+\phi_+^n + c_-\phi_-^n$. (We can shortcut and extract the value we want without actually computing the full expression.)

```
phip = Q5 (1%2) (1%2)
cp   = Q5 0     (1%5)
phim = Q5 (1%2) (-1%2)
cm   = Q5 0     (-1%5)
fibonacci' n = let (Q5 r q) = cp*phip^n + cm*phim^n in numerator r
fibonacci n = let (Q5 _ q) = phip^^n in numerator (2*q)
```

4 Numbers

4.1 Preamble

```
module Numbers where
import Data.Ratio
import qualified Fibo

fibonacci_number :: Int → Integer
fibonacci_number n = Fibo.fibonacci n

lucas_number :: Int → Integer
lucas_number = undefined

euler_number :: Int → Integer
euler_number = undefined

catalan_number :: Integer → Integer
catalan_number 0 = 1
catalan_number n = 2*(2*n−1)*(catalan_number (n−1))`div`(n+1)

bernoulli_number :: Int → Rational
bernoulli_number = undefined

tangent_number :: Int → Integer
tangent_number = undefined

triangular_number :: Integer → Integer
triangular_number n = n*(n+1)`div`2

factorial :: (Integral a) ⇒ a → a
factorial 0 = 1
factorial 1 = 1
factorial n = product [1..n]

binomial :: (Integral a) ⇒ a → a → a
binomial n k
    | k<0 = 0
    | n<0 = 0
    | k>n = 0
    | k==0 = 1
    | k==n = 1
    | k>n`div`2 = binomial n (n−k)
    | otherwise = (product [n−(k−1)..n]) `div` (product [1..k])
```

4.2 Stirling numbers

```
-- TODO: this is extremely inefficient approach
stirling_number_first_kind n k = s n k
  where s n k | k≤0 ∨ n≤0 = 0
        s n 1 = (−1)^(n−1)*(factorial (n−1))
        s n k = (s (n−1) (k−1)) − (n−1)*(s (n−1) k)

-- TODO: this is extremely inefficient approach
stirling_number_second_kind n k = s n k
  where s n k | k≤0 ∨ n≤0 = 0
        s n 1 = 1
        s n k = k*(s (n−1) k) + (s (n−1) (k−1))
```

5 Exponential & Logarithm

In this section, we implement the exponential function and logarithm function, as well as useful variations.

5.1 Preamble

We begin with a typical preamble.

module Exp

```
{-# Language BangPatterns #-}
{-# Language FlexibleInstances #-}
module Exp (
    sf_exp, sf_expn, sf_exp_m1, sf_exp_m1vx, sf_exp_men, sf_exp_menx,
    sf_log, sf_log_p1,
) where
import Numbers
import Util
```

5.2 Exponential

We start with implementation of the most basic special function, $exp(x)$ or e^x and variations thereof.

5.2.1 sf_exp x

For the exponential `sf_exp x` $= \exp(x)$ we use a simple series expansions

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

after first using the identity $e^{-x} = 1/e^x$ to ensure that the real part of the argument is positive. This avoids disastrous cancellation for negative arguments, (though note that for complex arguments this is not sufficient.) TODO: should do range-reduction first... TODO: maybe for complex, use explicit cis?

sf_exp x $= e^x$

```
sf_exp :: (Value v) => v -> v
sf_exp !x
  | is_inf x  = if (re x)<0 then 0 else pos_infty
  | is_nan x  = x
  | (re x)<0  = 1/(sf_exp (-x))
  | otherwise = ksum $ ixiter 1 1.0 $ \n t -> t*x/(#)n
```

5.2.2 sf_exp_m1 x

For numerical calculations, it is useful to have `sf_exp_m1 x` $= e^x - 1$ as explicitly calculating this expression will give poor results for x near 1. We use a series expansion for the calculation. Again for negative real part we reflect using $e^{-x} - 1 = -e^{-x}(e^x - 1)$. TODO: should do range-reduction first... TODO: maybe for complex, use explicit cis?

sf_exp_m1 x $= e^x - 1$

```
sf_exp_m1 :: (Value v) ⇒ v → v
sf_exp_m1 !x
  | is_inf x   = if (re x)<0 then −1 else pos_infty
  | is_nan x   = x
  | (re x)<0   = −sf_exp x ∗ sf_exp_m1 (−x)
  | otherwise = ksum $ ixiter 2 x $ λn t → t∗x/((#)n)
```

5.2.3 sf_exp_m1vx x

Similarly, it is useful to have the scaled variant `sf_exp_m1vx x` $= \frac{e^x-1}{x}$. In this case, we use a continued-fraction expansion

$$\frac{e^x-1}{x} = \frac{2}{2-x+}\,\frac{x^2/6}{1+}\,\frac{x^2/4\cdot 3\cdot 5}{1+}\,\frac{x^2/4\cdot 5\cdot 7}{1+}\,\frac{x^2/4\cdot 7\cdot 9}{1+}\cdots$$

For complex values, simple calculation is inaccurate (when $\Re z \sim 1$).

sf_exp_m1vx x $= \frac{e^x-1}{x}$

```
sf_exp_m1vx :: (Value v) ⇒ v → v
sf_exp_m1vx !x
  | is_inf x = if (re x)<0 then 0 else pos_infty
  | is_nan x = x
  | rabs(x)>(1/2) = (sf_exp x − 1)/x −− inaccurate for some complex points
  | otherwise =
      let x2 = xˆ2
      in 2/(2 − x + x2/6/(1
           + x2/(4∗(2∗3−3)∗(2∗3−1))/(1
           + x2/(4∗(2∗4−3)∗(2∗4−1))/(1
           + x2/(4∗(2∗5−3)∗(2∗5−1))/(1
           + x2/(4∗(2∗6−3)∗(2∗6−1))/(1
           + x2/(4∗(2∗7−3)∗(2∗7−1))/(1
           + x2/(4∗(2∗8−3)∗(2∗8−1))/(1
           ))))))));
```

5.2.4 sf_exp_menx n x

Compute the scaled tail of series expansion of the exponential function.

$$\texttt{sf\_exp\_menx n x} = \frac{n!}{x^n}\left(e^z - \sum_{k=0}^{n-1}\frac{x^k}{k!}\right) = \frac{n!}{x^n}\sum_{k=n}^{\infty}\frac{x^k}{k!} = n!\sum_{k=0}^{\infty}\frac{x^k}{(k+n)!}$$

We use a continued fraction expansion and using the modified Lentz algorithm for evaluation.

sf_exp_menx n z

```
sf_exp_menx :: (Value v) ⇒ Int → v → v
sf_exp_menx 0 z = sf_exp z
sf_exp_menx 1 z = sf_exp_m1vx z
sf_exp_menx n z
  | is_inf z  = if (re z)>0 then pos_infty else (0) -- TODO: verify
  | is_nan z  = z
  | otherwise = exp_menx__contfrac n z
  where
    !zeta = 1e-150
    !eps = 1e-16
    nz !z = if z==0 then zeta else z
    exp_menx__contfrac n z =
      let !fj = (#)$ n+1
          !cj = fj
          !dj = 0
          !j  = 1
      in lentz j dj cj fj
    lentz !j !dj !cj !fj =
      let !aj = if (odd j)
                then z*((#)$(j+1)`div`2)
                else -z*((#)$(n+(j`div`2)))
          bj = (#)$n+1+j
          !dj' = nz$ bj + aj*dj
          !cj' = nz$ bj + aj/cj
          !dji = 1/dj'
          !deltaj = cj'*dji
          !fj' = fj*deltaj
      in if (rabs(deltaj-1)<eps)
         then 1/(1-z/fj')
         else lentz (j+1) dji cj' fj'
```

5.2.5 sf_exp_men n x

This is the generalization of `sf_exp_m1 x`, giving the tail of the series expansion of the exponential function, for $n = 0, 1, \ldots$.

$$\texttt{sf\_exp\_men n z} = e^z - \sum_{k=0}^{n-1} \frac{z^k}{k!} = \sum_{k=n}^{\infty} \frac{z^k}{k!}$$

The special cases are: $n = 0$ gives $e^x =$ `sf_exp x` and $n = 1$ gives $e^x - 1 =$ `sf_exp_m1 x`. We compute this by calling the scaled version `sf_exp_menx` and rescaling back.

```
-- ($n=0, 1, 2, ∘ ..$)
sf_exp_men :: (Value v) ⇒ Int → v → v
sf_exp_men !n !x = (sf_exp_menx n x) * x^n / ((#)$factorial n)
```

5.2.6 sf_expn n x

```
-- Compute initial part of series for exponential, $\sum_(k=0)^n z^k/k!$
-- ($n=0,1,2,...$)
sf_expn :: (Value v) ⇒ Int → v → v
sf_expn n z
  | is_inf z  = if (re z)>0 then (1/0) else (if (odd n) then (-1/0) else (1/0))
  | is_nan z  = z
  | otherwise = expn__series n z
```

```
  where
    -- TODO: just call sf_exp when possible
    -- TODO: better handle large -ve values!
    expn__series :: (Value v) ⇒ Int → v → v
    expn__series n z = ksum $ take (n+1) $ ixiter 1 1.0 $ λk t → t*z/(#)k
```

5.3 Logarithm

5.3.1 `sf_log x`

We simply use the built-in implementation (from the `Floating` typeclass).

```
sf_log :: (Value v) ⇒ v → v
sf_log = log
```

5.3.2 `sf_log_p1 x`

The accuracy preserving `sf_log_p1 x` $= \ln 1 + x$. For values close to zero, we use a power series expansion

$$\ln(1+x) = 2\sum_{n=0}^{\infty} \frac{(\frac{x}{x+2})^{2n+1}}{2n+1}$$

and otherwise just compute it directly.

`sf_log_p1 z` $= \ln z + 1$

```
sf_log_p1 :: (Value v) ⇒ v → v
sf_log_p1 !z
  | is_nan z = z
  | (rabs z)>0.25 = sf_log (1+z)
  | otherwise = series z
  where
    series z =
      let !r = z/(z+2)
          !zr2 = r^2
          !tterms = iterate (*zr2) (r*zr2)
          !terms = zipWith (λn t → t/((#)$2*n+1)) [1..] tterms
      in 2*(ksum (r:terms))
```

A simple continued fraction implementation for $\ln 1 + z$

$$\ln(1+z) = z/(1+z/(2+z/(3+4z/(4+4z/(5+9z/(6+9z/(7+\cdots)))))))$$

Though unused for now, it seems to have decent convergence properties.

```
ln_1_z_cf z = sf_cf_steeds (z:(ts 1)) [0..]
  where ts n = (n^2*z):(n^2*z):(ts (n+1))
```

6 Gamma

6.1 Preamble

A basic preamble.

```
module Gamma (
    euler_gamma,
    factorial,
    sf_beta,
    sf_gamma,
    sf_invgamma,
    sf_lngamma,
    sf_digamma,
    bernoulli_b,
    )
where
import Exp
import Numbers(factorial)
import Trig
import Util
```

6.2 Misc

6.2.1 euler_gamma

A constant for Euler's gamma:

$$\gamma = \lim_{n\to\infty}\left(\sum_{k=1}^{n}\frac{1}{n} - \ln n\right)$$

```
euler_gamma :: (Floating a) ⇒ a
euler_gamma = 0.577215664901532860606512090082402431042159335939923598805767234884867726777664670936947063291746749
```

6.2.2 sf_beta a b

The Beta integral

$$B(a,b) = \int_0^1 t^{a-1}(1-t)^{b-1}\,dt = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}$$

implemented in terms of log-gamma

$$\texttt{sf\_beta a b} = e^{\ln\Gamma(a)+\ln\Gamma(b)-\ln\Gamma(a+b)}$$

```
sf_beta :: (Value v) ⇒ v → v → v
sf_beta a b = sf_exp $ (sf_lngamma a) + (sf_lngamma b) − (sf_lngamma$a+b)
```

6.3 Gamma

The gamma function

$$\Gamma(z) = \int_0^\infty e^{-t}t^z\,\frac{dz}{z}$$

6.3.1 sf_gamma z

The gamma function implemented using the identity $\Gamma(z) = \frac{1}{z}\Gamma(z+1)$ to increase the real part of the argument to be > 15 and then using an asymptotic expansion for log-gamma, `lngamma_asymp`, to evaluate.

sf_gamma x $= \Gamma(x)$ sf_gamma

```
sf_gamma :: (Value v) ⇒ v → v
sf_gamma x =
  redup x 1 $ λ x' t → t ∗ (sf_exp (lngamma_asymp x'))
  where redup x t k
          | (re x)>15 = k x t
          | otherwise = redup (x+1) (t/x) k
```

6.3.2 *lngamma_asymp z

The asymptotic expansion for log-gamma

$$\ln\Gamma(z) \sim (z-\frac{1}{2})\ln z - z + \frac{1}{2}\ln(2\pi) + \sum_{k=1}^{\infty}\frac{B_{2k}}{2k(2k-1)z^{2k-1}}$$

where B_n is the n'th Bernoulli number.

```
lngamma_asymp :: (Value v) ⇒ v → v
lngamma_asymp z = (z − 1/2)∗(sf_log z) − z + (1/2)∗sf_log(2∗pi) + (ksum terms)
  where terms = [b2k/(2∗k∗(2∗k−1)∗z^(2∗k'−1)) | k'←[1..10], let k=(#)k', let b2k=bernoulli_b$2∗k']
```

6.3.3 sf_invgamma z

The inverse gamma function, **sf_invgamma z** $= \frac{1}{\Gamma(z)}$.

```
sf_invgamma :: (Value v) ⇒ v → v
sf_invgamma x =
  let (x',t) = redup x 1
      lngx = lngamma_asymp x'
  in t ∗ (sf_exp$ −lngx)
  where redup x t
          | (re x)>15 = (x,t)
          | otherwise = redup (x+1) (t∗x)
```

6.3.4 sf_lngamma z

The log-gamma function, **sf_lngamma z** $= \ln\Gamma(z)$.

```
sf_lngamma :: (Value v) ⇒ v → v
sf_lngamma x =
  let (x',t) = redup x 0
      lngx = lngamma_asymp x'
  in t + lngx
  where redup x t
          | (re x)>15 = (x,t)
          | otherwise = redup (x+1) (t−sf_log x)
```

6.3.5 bernoulli_b n

The Bernoulli numbers, B_n. A simple hard-coded table, for now. (Should be moved to Numbers module and general, cached, implementation done.)

```
bernoulli_b :: (Value v) ⇒ Int → v
bernoulli_b 1 = −1/2
bernoulli_b k | k`mod`2==1 = 0
```

```
bernoulli_b 0 = 1
bernoulli_b 2 = 1/6
bernoulli_b 4 = −1/30
bernoulli_b 6 = 1/42
bernoulli_b 8 = −1/30
bernoulli_b 10 = 5/66
bernoulli_b 12 = −691/2730
bernoulli_b 14 = 7/6
bernoulli_b 16 = −3617/510
bernoulli_b 18 = 43867/798
bernoulli_b 20 = −174611/330
bernoulli_b _ = undefined
```

Spouge's approximation to the gamma function

In tests, this gave disappointing results.

```
-- Spouge's approximation (a=17?)
spouge_approx :: (Value v) ⇒ Int → v → v
spouge_approx a z' =
  let z = z' − 1
      a' = (#)a
      res = (z+a')**(z+(1/2)) * sf_exp (−(z+a'))
      sm = fromDouble$sf_sqrt(2*pi)
      terms = [(spouge_c k a') / (z+k') | k←[1..(a−1)], let k' = (#)k]
      smm = sm + ksum terms
  in res*smm
  where
    spouge_c k a = ((if k`mod`2==0 then −1 else 1) / ((#) $ factorial (k−1)))
                   * (a−((#)k))**(((#)k)−1/2) * sf_exp(a−((#)k))

spouge :: (Value v) ⇒ Int → v → v
spouge a' z' =
  let z = z' − 1
      a = fromDouble$(#)a'
      -- I don't quite understand why I can't do this:
      --q = fromReal $ (sf_sqrt(2*pi) :: (RealKind v))
      q = sf_sqrt(2*pi)
  in (z+a)**(z+1/2)*(sf_exp(−z−a))*(q + ksum (map (λk→(c a k)/(z+(#)k)) [1..(a'−1)]))
  where
    c :: (Value v) ⇒ v → Int → v
    c a k = let k' = (#)k
                sgn = if even k then −1 else 1
            in sgn*(a−k')**(k'−1/2)*(sf_exp(a−k')) / ((#)$factorial(k−1))
```

6.4 Digamma

The digamma function

$$\psi(z) = \frac{d}{dz}\ln\Gamma(z) = \frac{\Gamma'(z)}{\Gamma(z)}$$

6.4.1 `sf_digamma z`

We implement with a series expansion for $|z| <= 10$ and otherwise with an asymptotic expansion.

```
sf_digamma :: (Value v) ⇒ v → v
--sf_digamma n | is_nonposint n = Inf
sf_digamma z | (rabs z)>10 = digamma__asympt z
             | otherwise   = digamma__series z
```

The series expansion is the following

$$\psi(z) = -\gamma - \frac{1}{z} + \sum_{k=1}^{\infty} \frac{z}{k(k+z)}$$

but with Euler-Maclaurin correction terms:

$$\psi(z) = -\gamma - \frac{1}{z} + \sum_{k=1}^{n} \frac{z}{k(k+z)} + (\ln \frac{k+z}{k} - \frac{z}{2k(k=z)} + \sum_{j=1}^{p} B_{2j}(k^{-2j} - (k+z)^{-2j})$$

```
digamma_series :: (Value v) ⇒ v → v
digamma_series z =
  let res = −euler_gamma − (1/z)
      terms = map (λk→z/((#)k∗(z+(#)k))) [1..]
      corrs = map (correction.(#)) [1..]
  in summer res res terms corrs
  where
    summer :: (Value v) ⇒ v → v → [v] → [v] → v
    summer res sum (t:terms) (c:corrs) =
      let sum' = sum + t
          res' = sum' + c
      in if res==res' then res
         else summer res' sum' terms corrs
    bn1 = bernoulli_b 2
    bn2 = bernoulli_b 4
    bn3 = bernoulli_b 6
    bn4 = bernoulli_b 8
    correction k =
      (sf_log$(k+z)/k) − z/2/(k∗(k+z))
        + bn1∗(k^^(−2) − (k+z)^^(−2))
        + bn2∗(k^^(−4) − (k+z)^^(−4))
        + bn3∗(k^^(−6) − (k+z)^^(−6))
        + bn4∗(k^^(−8) − (k+z)^^(−8))
```

The asymptotic expansion (valid for $|argz| < \pi$) is the following

$$\psi(z) \sim \ln z - \frac{1}{2z} + \sum_{k=1}^{\infty} \frac{B_{2k}}{2kz^{2k}}$$

Note that our implementation will fail if the `bernoulli_b` table is exceeded. If $\Re z < \frac{1}{2}$ then we use the reflection identity to ensure $\Re z \geq \frac{1}{2}$:

$$\psi(z) - \psi(1-z) = \frac{-\pi}{\tan(\pi z)}$$

```
digamma_asympt :: (Value v) ⇒ v → v
digamma_asympt z
  | (re z)<0.5 = compute (1 − z) $ −pi/(sf_tan(pi∗z)) + (sf_log(1−z)) − 1/(2∗(1−z))
  | otherwise  = compute z $ (sf_log z) − 1/(2∗z)
    where
      compute z res =
        let z_2 = z^^(−2)
            zs = iterate (∗z_2) z_2
            terms = zipWith (λn z2n → z2n∗(bernoulli_b(2∗n+2))/(#)(2∗n+2)) [0..] zs
        in sumit res res terms
      sumit res ot (t:terms) =
        let res' = res − t
        in if res==res' ∨ (rabs t)>(rabs ot)
           then res
           else sumit res' t terms
```

7 Error function

7.1 Preamble

```
{-# Language BangPatterns #-}
-- {-# Language BlockArguments #-}
{-# Language ScopedTypeVariables #-}
module Erf (
    sf_erf,
    sf_erfc,
) where
import Exp
import Util
```

7.2 Error function

The error function is defined via

$$\operatorname{erf}(z) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-x^2}\,dx \qquad \operatorname{erf}(z)$$

and the complementary error function via

$$\operatorname{erfc}(z) = \frac{2}{\sqrt{\pi}} \int_z^\infty e^{-x^2}\,dx \qquad \operatorname{erfc}(z)$$

Thus we have the relation $\operatorname{erf}(z) + \operatorname{erfc}(z) = 1$.

7.2.1 `sf_erf z`

The error function `sf_erf z` $= \operatorname{erf} z$ where

$$\operatorname{erf}(z) = \frac{2}{\sqrt{\pi}} \int_{-\infty}^z e^{-x^2}\,dx$$

For $\Re z < -1$, we transform via $\operatorname{erf}(z) = -\operatorname{erf}(-z)$ and for $|z| < 1$ we use the power-series expansion, otherwise we use $\operatorname{erf} z = 1 - \operatorname{erfc} z$. (TODO: this implementation is not perfect, but workable for now.)

`sf_erf z` $= \operatorname{erf}(z)$

```
sf_erf :: (Value v) => v -> v
sf_erf z
  | (re z)<(-1) = -sf_erf(-z)
  | (rabs z)<1  = erf__series z
  | otherwise   = 1 - sf_erfc z
```

7.2.2 `sf_erfc z`

The complementary error-function `sf_erfc z` $= \operatorname{erfc} z$ where

$$\operatorname{erfc} z = 1 - \operatorname{erf} z = \frac{2}{\sqrt{\pi}} \int_z^\infty e^{-x^2}\,dx$$

For $\Re z < -1$ we transform via $\operatorname{erfc} z = 2 - \operatorname{erf}(-z)$ and if $|z| < 1$ then we use $\operatorname{erfc} z = 1 - \operatorname{erf} z$. Finally, if $|z| < 10$ we use a continued-fraction expansion and an asymptotic expansion otherwise. (TODO: there are a few issues with this implementation: For pure imaginary values and for extremely large values it seems to hang.)

sf_erfc z = erfc(z)

```
sf_erfc :: (Value v) ⇒ v → v
sf_erfc z
  | (re z)<(−1) = 2−(sf_erfc (−z))
  | (rabs z)<1  = 1−(sf_erf z)
  | (rabs z)<10 = erfc_cf_pos1 z
  | otherwise   = erfc_asymp_pos z −− TODO: hangs for very large input
```

erf_series z

The series expansion for erf z:

$$\operatorname{erf} z = \frac{2}{\sqrt{\pi}} \sum_{n=0}^{\infty} \frac{(-)^n z^{2n+1}}{n!(2n+1)}$$

There is an alternative expansion $\operatorname{erf} z = \frac{2}{\sqrt{\pi}} e^{-z^2} \sum_{n=0}^{\infty} \frac{2^n z^{2n+1}}{1 \cdot 3 \cdots (2n+1)}$, but we don't use it here. (TODO: why not?)

```
erf__series z =
  let z2 = z^2
      rts = ixiter 1 z $ λn t → (−t)∗z2/(#)n
      terms = zipWith (λn t → t/(#)(2∗n+1)) [0..] rts
  in (2/sf_sqrt pi) ∗ ksum terms
```

***sf_erf z**

This asymptotic expansion for erfc z is valid as $z \to +\infty$:

$$\operatorname{erfc} z \sim \frac{e^{-z^2}}{\sqrt{\pi}} \sum_{n=0}^{\infty} (-)^n \frac{(1/2)_m}{z^{2m+1}}$$

where the Pochhammer symbol $(1/2)_m$ is given by:

$$\left(\frac{1}{2}\right)_m = \frac{1 \cdot 3 \cdot 5 \cdots (2m-1)}{2^m} = \frac{(2m)!}{m! 2^{2m}}$$

TODO: correct the asymptotic term checking (not smallest but pre-smallest term).

```
erfc_asymp_pos z =
  let z2 = z^2
      iz2 = 1/2/z2
      terms = ixiter 1 (1/z) $ λn t → (−t∗iz2)∗(#)(2∗n−1)
      tterms = tk terms
  in (sf_exp (−z2))/(sqrt pi) ∗ ksum tterms
  where tk (a:b:cs) = if (rabs a)<(rabs b) then [a] else a:(tk$b:cs)
```

***erfc_cf_pos1 z**

A continued-fraction expansion for erfc z:

$$\sqrt{\pi} e^{z^2} \operatorname{erfc} z = \frac{z}{z^2+} \frac{1/2}{1+} \frac{1}{z^2+} \frac{3/2}{1+} \cdots$$

```
erfc_cf_pos1 z =
  let z2 = z^2
      as = z:(map fromDouble [1/2,1..])
      bs = 0:cycle [z2,1]
      cf = sf_cf_steeds as bs
  in sf_exp(-z2) / (sqrt pi) * cf
```

***erfc_cf_pos1 z**

This is an alternative continued-fraction expansion.

$$\sqrt{\pi}e^{z^2}\operatorname{erfc} z = \frac{2z}{2z^2+1-}\,\frac{1\cdot 2}{2z^2+5-}\,\frac{3\cdot 4}{2z^2+9-}\cdots$$

Unused for now.

```
erfc_cf_pos2 z =
  let z2 = z^2
      as = (2*z):(map (\n->(#)$ -(2*n+1)*(2*n+2)) [0..])
      bs = 0:(map (\n->2*z2+(#)4*n+1) [0..])
      cf = sf_cf_steeds as bs
  in sf_exp(-z2) / (sqrt pi) * cf
```

7.3 Dawson's function

Dawson's function (or Dawson's integral) is given by

$$D(z) = e^{-z^2}\int_0^z e^{t^2}\,dt = -\frac{\hat{\imath}\sqrt{\pi}}{2}e^{-x^2}\operatorname{erf}(\hat{\imath}x)$$

7.3.1 sf_dawson z

Compute Dawson's integral $D(z) = e^(-z^2)\int_0^z e^(t^2)dt$ for real z. (Correct only for reals!)

```
sf_dawson :: forall v.(Value v) => v -> v
sf_dawson z
  -- | (rabs z) < 0.5 = (toComplex$sf_exp(-z^2))*(sf_erf((toComplex z)*(0:+1)))*(sf_sqrt(pi)/2/(0:+1))
   | (im z) /= 0     = dawson__seres z
   | (rabs z) < 5    = dawson__contfrac z
   | otherwise       = dawson__contfrac2 z

dawson__seres :: (Value v) => v -> v
dawson__seres z =
  let tterms = ixiter 1 z $ \n t -> t*z^2/(#)n
      terms = zipWith (\n t->t/((#)(2*n+1))) [0..] tterms
      smm = ksum terms
  in (sf_exp(-z^2)) * smm

faddeeva__asymp :: (Value v) => v -> v
faddeeva__asymp z =
  let z' = 1/z
      terms = ixiter 1 z' $ \n t -> t*z'^2*((#)(2*n+1))/2
      smm = ksum terms
  in smm
{-
function res = seres(x)
  res = term = x;
  n = 1;
  do
```

```
      term *= x^2 / n;
      old_res = res;
      res += term / (2*n+1);
      ++n; if (n>999) break; endif
    until (res == old_res)
    res *= sf_exp(-x^2);
endfunction
-}


dawson_contfrac :: (Value v) => v -> v
dawson_contfrac z = undefined

dawson_contfrac2 :: (Value v) => v -> v
dawson_contfrac2 z = undefined

{-
function res = contfrac(x)
    eps = 1e-16;
    zeta = 1e-100;

    fj = 1;
    Cj = fj;
    Dj = 0;
    j = 1;
    do
      aj = (-1)^(rem(j,2)+1)*2*j*x^2;
      bj = 2*j+1;
      Dj = bj + aj*Dj; if (Dj==0) Dj=zeta; endif
      Cj = bj + aj/Cj; if (Cj==0) Cj=zeta; endif
      Dj = 1/Dj;
      Deltaj = Cj*Dj;
      fj *= Deltaj;
      ++j; if (j>999) break; endif
    until (abs(Deltaj-1)<eps)
    res = x/fj;
endfunction

function res = contfrac2(x)
    eps = 1e-16;
    zeta = 1e-100;

    fj = 1+2*x^2;
    Cj = fj;
    Dj = 0;
    j = 1;
    do
      aj = -4*j*x^2;
      bj = (2*j+1) + 2*x^2;
      Dj = bj + aj*Dj; if (Dj==0) Dj=zeta; endif
      Cj = bj + aj/Cj; if (Cj==0) Cj=zeta; endif
      Dj = 1/Dj;
      Deltaj = Cj*Dj;
      fj *= Deltaj;
      ++j; if (j>999) break; endif
    until (abs(Deltaj-1)<eps)
    res = x/fj;
endfunction
```

```
# from NR
# BUGGY
function res = rybicki(x)
  h = 2.0;
  n = 1;
  res = 0;
  do
    old_res = res;
    res += ( sf_exp(-(x-n*h)^2) - sf_exp(-(x+n*h)^2) )/n;
    n+=2; if (n>999) break; endif
  until (res == old_res)
  res /= sqrt(pi);
endfunction

function res = besser2(x)
  res = 0;
  n = 1;
  do
    old_res = res;
    res += (2*n+1)*sf_bessel_spher_i1(n, x^2) + (2*n+3)*sf_bessel_spher_i1(n+1, x^2);
    n +=4 ; if (n>999) break; endif
  until (res == old_res)
  res *= sf_exp(-x^2) / x;
endfunction

function res = besser(x)
  res = 0;
  n = 0;
  do
    old_res = res;
    res += (-1)^(rem(n,2)) * (sf_bessel_spher_i1(2*n, x^2) + sf_bessel_spher_i1(2*n+1, x^2));
    ++n; if (n>999) break; endif
  until (res == old_res)
  res *= x * sf_exp(-x^2);
endfunction
-}
```

8 Bessel Functions

Bessel's differential equation is:

$$z^2w'' + zw' + (z^2 - \nu^2)w = 0$$

8.1 Preamble

```
{-# Language BangPatterns #-}
{-# Language ScopedTypeVariables #-}
module Bessel where
import Gamma
import Trig
import Util
```

8.2 Bessel function J of the first kind

The Bessel functions $J_\nu(z)$ are defined as

8.2.1 sf_bessel_j nu z

Compute Bessel $J\_\nu(z)$ function

sf_bessel_j nu z $= J_\nu(z)$

```
sf_bessel_j :: (Value v) ⇒ v → v → v
sf_bessel_j nu z
  | (rabs z) < 5 = bessel_j__series nu z
  | otherwise     = bessel_j__asympt_z nu z
 —rec = recur_back(z, nu);
 —ref = recur_fore(z, nu);
 —re2 = recur_backwards(nu, z, round(abs(max(z, nu)))+21);
 —res = sys;
```

***bessel_j__series nu z**

The power-series expansion given by

$$J_\nu(z) = \left(\frac{z}{2}\right)^\nu \frac{1}{1+\nu} \sum_{k=0}^{\infty} (-)^k \frac{z^{2k}}{2^{2k} k! \Gamma(\nu + k + 1)}$$

bessel_j__series nu z

```
bessel_j__series :: (Value v) ⇒ v → v → v
bessel_j__series !nu !z =
  let !z2 = −(z/2)^2
      !terms = ixiter 1 1 $ λn t → t*z2/((#)n)/(nu+(#)n)
      !res = ksum terms
  in res * (z/2)**nu / sf_gamma(1+nu)
```

***bessel_j__asympt nu z**

Asymptotic expansion for $|z| >> \nu$ with $|arg z| < \pi$. is given by

$$J_\nu(z) \sim \left(\frac{2}{\pi z}\right)^{1/2} \left(\cos\omega \sum_{k=0}^{\infty} (-)^k \frac{a_{2k}(\nu)}{z^{2k}} - \sin\omega \sum_{k=0}^{\infty} (-)^k \frac{a_{2k+1}(\nu)}{z^{2k+1}} \right)$$

where $\omega = z - \frac{\pi\nu}{2} - \frac{\pi}{4}$ and

$$a_k(\nu) = \frac{(4\nu^2 - 1^2)(4\nu^2 - 3^2) \cdots (4\nu^2 - (2k-1)^2)}{k! 8^k}$$

(with $a_0(\nu) = 1$).

```
bessel_j__asympt_z :: ∀ v.(Value v) ⇒ v → v → v
bessel_j__asympt_z !nu !z =
  let !om = z − (nu/2 + 1/4)*pi
      !nu2 = nu^2
      !aks = ixiter 1 1 $ λk t → t*(4*nu2 − ((#)$((2*k−1)^2)))/((#)$8*k)/z
      !akse = tk $ zipWith (λk t → (−1)^k*t) [0..] (evel aks)
      !akso = tk $ zipWith (λk t → (−1)^k*t) [0..] (evel (tail aks))
  in (sf_sqrt(2/pi/z))*((sf_cos om)*(ksum akse) − (sf_sin om)*(ksum akso))
  where
    tk :: [v] → [v]
    tk (a:b:c:xs) = if (rabs b) < (rabs c) then [a] else a:(tk (b:c:xs))
    evel (a:b:cs) = a:(evel cs)
```

This approach uses the recursion in order (for large order) in a backward direction

$$J_{\nu-1}(z) = \frac{2\nu}{z} J_\nu(z) - J_{\nu+1}(z)$$

(largest to smallest) wtih ...

```
--bessel_j_recur_back :: (Value v) => Double -> v -> v
bessel_j_recur_back :: ∀ v.(Value v) => (RealKind v) -> v -> [v]
bessel_j_recur_back !nu !z =
  let !jjs = runback (nnx-2) [1.0,0.0]
      !scale = if (rabs z)<10 then (bessel_j__series nuf z) else (bessel_j__asympt_z nuf z)
      --scale2 = ((head jjs)^2) + 2*(ksum (map (^2) $ tail jjs)) -- only integral nu
  --in jjs!!(nnn) * scale / (jjs!!0)
  in map (\j -> j * scale / (jjs!!0)) (take (nnn+1) jjs)
  --in map (\j -> j/scale2) (take (nnn+1) jjs)
  where
    !nnn = truncate nu
    !nuf = fromReal $ nu - (#)nnn
    !nnx = nnn + 20
    runback :: Int -> [v] -> [v]
    runback !0 !j = j
    runback !nx !j@(jj1:jj2:jjs) =
      let !jj = jj1*2*(nuf+(#)nx)/z - jj2
      in runback (nx-1) (jj:j)

{--
-- recursion in order (forewards)
bessel_j_recur_fore :: (Value v) => Double -> v -> v
bessel_j_recur_fore !nu !z =
  let !jj1 = bessel_j_series nuf z
      !jj2 = bessel_j_series (nuf+1) z
  in loop 3 jj1 jj2
  where
    !nnn = truncate nu
    !nuf = nu - (#)nnn
    !nnx = nnn + 10
    loop :: Int -> v -> v -> v
    loop j jjm2 jjm1
      | j==(nnx+1) = jjm1
      | otherwise =
          let jjj = jjm1*2*(fromDouble(nuf+(#)j))/z - jjm2
          in loop (j+1) jjm1 jjj

{--
function res = recur_backwards(n, z, topper)
  jjp2 = zeros(size(z));
  jjp1 = ones(size(z));
  jjp2_e_ = 1e-40 * ones(size(z));
  jjp1_e_ = 1e-20 * ones(size(z));
  scale = 2*ones(size(z));
  res = zeros(size(z));
  for m= (topper-2):(-1):1
    #jj(m) = (2*nu/z)*jj(m+1) - jj(m+2);
    s_ =-jjp2;
    e_ =-jjp2_e_;
    # add high
      t_ = s_;
      y_ = ((2*m./z).*jjp1) + e_;
      s_ = t_ + y_;
```

```
      e_ = (t_ − s_) + y_;
    # add low
      t_ = s_;
      y_ = ((2*m./z).*jjp1_e_) + e_;
      s_ = t_ + y_;
      e_ = (t_ − s_) + y_;
    jjp2 = jjp1;
    jjp2_e_ = jjp1_e_;
    jjp1 = s_;
    jjp1_e_ = e_;

    if (m==n+1)
      # store the desired result,
      # but keep recursing to get scale factor
      res = jjp1;
    endif

    if (m!=1)
      scale += 2 * (s_.^2 + e_.^2 + 2*s_.*e_);
    else
      scale += 1 * (s_.^2 + e_.^2 + 2*s_.*e_);
    endif

    if (scale>1e20)
      jjp2 /= 1024;
      jjp2_e_ /= 1024;
      jjp1 /= 1024;
      jjp1_e_ /= 1024;
      res /= 1024;
      scale /= 1024^2;
    endif
  endfor
  res ./= sqrt(scale);
endfunction
—}
—}
```

9 Exponential Integral

9.1 Preamble

```
module ExpInt(
  sf_expint_ei,
  sf_expint_en,
  )
where
import Exp
import Gamma
import Util
```

9.2 Exponential integral Ei

The exponential integral $\mathrm{Ei}\, z$ is defined for $x < 0$ by

$$\mathrm{Ei}(z) = -\int_{-x}^{\infty} \frac{e^{-t}}{t}\, dt$$

It can be defined

9.2.1 sf_expint_ei z

We give only an implementation for $\Re z \geq 0$. We use a series expansion for $|z| < 40$ and an asymptotic expansion otherwise.

sf_expint_ei z $= \mathrm{Ei}(z)$

sf_expint_ei

```
sf_expint_ei :: (Value v) ⇒ v → v
sf_expint_ei z
  | (re z) < 0.0   = (0/0)   -- (NaN)
  | z == 0.0       = (−1/0)  -- (−Inf)
  | (rabs z) < 40  = expint_ei__series z
  | otherwise      = expint_ei__asymp z
```

The series expansion is given (for $x > 0$)

$$\mathrm{Ei}(x) = \gamma + \ln x + \sum_{n=1}^{\infty} \frac{x^n}{n!n}$$

We evaluate the addition of the two terms with the sum slightly differently when $\Re z < 1/2$ to reduce floating-point cancellation error slightly.

expint_ei__series z

expint_ei__se

```
expint_ei__series :: (Value v) ⇒ v → v
expint_ei__series z =
  let tterms = ixiter 2 z $ λn t → t*z/(#)n
      terms = zipWith (λ t n →t/(#)n) tterms [1..]
      res = ksum terms
  in if (re z)<0.5
     then sf_log(z * sf_exp(euler_gamma + res))
     else res + sf_log(z) + euler_gamma
```

The asymptotic expansion as $x \to +\infty$ is

$$\mathrm{Ei}(x) \sim \frac{e^x}{x} \sum_{n=0}^{\infty} \frac{n!}{x^n}$$

expint_ei__asymp z

expint_ei__as

```
expint_ei__asymp :: (Value v) ⇒ v → v
expint_ei__asymp z =
  let terms = tk $ ixiter 1 1.0 $ λn t → t/z*(#)n
      res = ksum terms
  in res * (sf_exp z) / z
  where tk (a:b:cs) = if (rabs a)<(rabs b) then [a] else a:(tk$b:cs)
```

9.3 Exponential integral E_n

The exponential integrals $E_n(z)$ are defined as

$$E_n(z) = z^{n-1} \int_z^{\infty} \frac{e^{-t}}{t^n}\, dt$$

They satisfy the following relations:

$$E_0(z) = \frac{e^{-z}}{z}$$
$$E_{n+1}(z) = \int_z^\infty E_n(t)\,dt$$

And they can be expressed in terms of incomplete gamma functions:

$$E_n(z) = z^{n-1}\Gamma(1-n,z)$$

(which also gives a generalization for non-integer n).

9.3.1 sf_expint_en n z

sf_expint_en n z = $E_n(z)$ sf_expint_en

```
sf_expint_en :: (Value v) ⇒ Int → v → v
sf_expint_en n z | (re z)<0 = (0/0) −− (NaN) TODO: confirm this
                 | z == 0   = (1/(#)(n−1)) −− TODO: confirm this
sf_expint_en 0 z = sf_exp(−z) / z
sf_expint_en 1 z = expint_en__1 z
sf_expint_en n z | (rabs z) ≤ 1.0 = expint_en__series n z
                 | otherwise = expint_en__contfrac n z
```

We use this series expansion for $E_1(z)$:

$$E_1(z) = -\gamma - \ln z + \sum_{k=1}^{\infty}(-)^k \frac{z^k}{k!k}$$

(Note that this will not be good for large values of z.)

```
expint_en__1 :: (Value v) ⇒ v → v
expint_en__1 z =
  let r0 = −euler_gamma − (sf_log z)
      tterms = ixiter 2 (z) $ λk t → −t∗z/(#)k
      terms = zipWith (λ t k → t/(#)k) tterms [1..]
  in ksum (r0:terms)

−− assume n≥ 2, z≤1
expint_en__series :: (Value v) ⇒ Int → v → v
expint_en__series n z =
  let n' = (#)n
      res = (−(sf_log z) + (sf_digamma n')) ∗ (−z)^(n−1)/(#)(factorial$n−1) + 1/(n'−1)
      terms' = ixiter 2 (−z) (λm t → −t∗z/(#)m)
      terms = map (λ(m,t)→(−t)/(#)(m−(n−1))) $ filter ((/=(n−1)) ∘ fst) $ zip [1..] terms'
  in ksum (res:terms)

−− assume n≥ 2, z>1
−− modified Lentz algorithm
expint_en__contfrac :: (Value v) ⇒ Int → v → v
expint_en__contfrac n z =
  let fj = zeta
      cj = fj
      dj = 0
```

```
    j = 1
    n' = (#)n
in lentz j cj dj fj
where
  zeta = 1e-100
  eps = 5e-16
  nz x = if x==0 then zeta else x
  lentz j cj dj fj =
    let aj = (#) $ if j==1 then 1 else -(j-1)*(n+j-2)
        bj = z + (#)(n + 2*(j-1))
        dj' = nz $ bj + aj*dj
        cj' = nz $ bj + aj/cj
        dji = 1/dj'
        delta = cj'*dji
        fj' = fj*delta
    in if (rabs$delta-1)<eps
       then fj' * sf_exp(-z)
       else lentz (j+1) cj' dji fj'
```

10 AGM

10.1 Preamble

module AGM

```
module AGM (
    sf_agm,
    sf_agm',
    )
where
import Util
```

10.2 AGM

Gauss' arithmetic-geometric mean or AGM of two numbers is defined as the limit $\mathrm{agm}(\alpha,\beta) = \lim_n \alpha_n = \lim_n \beta_n$ where we define

$$\begin{aligned}\alpha_{n+1} &= \frac{\alpha_n+\beta_n}{2}\\ \beta_{n+1} &= \sqrt{\alpha_n\cdot\beta_n}\end{aligned}$$

(Note that we need real values to be positive for this to make sense.)

10.2.1 sf_agm alpha beta

Here we compute the AGM via the definition and return the full arrays of intermediate values $([\alpha_n],[\beta_n],[\gamma_n])$, where $\gamma_n = \frac{\alpha_n-\beta_n}{2}$. (The iteration converges quadratically so this is an efficient approach.)

sf_agm alpha beta $= \mathrm{agm}(\alpha,\beta)$ sf_agm

```
sf_agm :: (Value v) => v -> v -> ([v],[v],[v])
sf_agm alpha beta = agm [alpha] [beta] [alpha-beta]
  where agm as@(a:_) bs@(b:_) cs@(c:_) =
```

sf_agm alpha beta $= \mathrm{agm}(\alpha, \beta)$ **(cont)** sf_agm

```
        if c==0 then (as,bs,cs)
        else let a' = (a+b)/2
                 b' = sf_sqrt (a*b)
                 c' = (a-b)/2
             in if c'==c then (as,bs,cs)
                else agm (a':as) (b':bs) (c':cs)
```

10.2.2 sf_agm' alpha beta

Here we return simply the value `sf_agm' a b` $= \mathrm{agm}(a, b)$.

sf_agm' z $= \mathrm{agm}\, z$

```
sf_agm' :: (Value v) => v -> v -> v
sf_agm' alpha beta = agm alpha beta ((alpha-beta)/2)
  --let (as,_,_) = sf_agm alpha beta in head as
  where agm a b 0 = a
        agm a b c =
          let a' = (a+b)/2
              b' = sf_sqrt (a*b)
              c' = (a-b)/2
          in agm a' b' c'
```

```
sf_agm_c0 :: (Value v) => v -> v -> v -> ([v],[v],[v])
sf_agm_c0 alpha beta c0 = undefined
```

11 Airy

The Airy functions Ai and Bi, standard solutions of the ode $y'' - zy = 0$.

11.1 Preamble

A basic preamble.

```
module Airy (sf_airy_ai, sf_airy_bi) where
import Gamma
import Util
```

11.2 Ai

11.2.1 sf_airy_ai z

For now, just use a simple series expansion.

```
sf_airy_ai :: (Value v) => v -> v
sf_airy_ai z = airy_ai_series z
```

Initial conditions $\mathrm{Ai}(0) = 3^{-2/3}\frac{1}{\Gamma(2/3)}$ and $\mathrm{Ai}'(0) = -3^{-1/3}\frac{1}{\Gamma(1/3)}$

```
ai0 :: (Value v) ⇒ v
ai0 = 3**(−2/3)/sf_gamma(2/3)

ai'0 :: (Value v) ⇒ v
ai'0 = −3**(−1/3)/sf_gamma(1/3)
```

Series expansion, where $n!!! = \max(n,1)$ for $n \le 2$ and otherwise $n!!! = n\cdot(n-3)!!!$:

$$\mathrm{Ai}(z) = \mathrm{Ai}(0)\left(\sum_{n=0}^{\infty}\frac{(3n-2)!!!}{(3n)!}z^{3n}\right) + \mathrm{Ai}'(0)\left(\frac{(3n-1)!!!}{(3n+1)!}z^{3n+1}\right)$$

```
airy_ai_series z =
    let z3 = z^3
        aiterms  = ixiter 0 1 $ λn t → t*z3*((#)$3*n+1)/((#)$(3*n+1)*(3*n+2)*(3*n+3))
        ai'terms = ixiter 0 z $ λn t → t*z3*((#)$3*n+2)/((#)$(3*n+2)*(3*n+3)*(3*n+4))
    in ai0 * (ksum aiterms) + ai'0 * (ksum ai'terms)
```

11.3 Bi

11.3.1 `sf_airy_bi z`

For now, just use a simple series expansion.

```
sf_airy_bi :: (Value v) ⇒ v → v
sf_airy_bi z = airy_bi_series z
```

Initial conditions $\mathrm{Bi}(0) = 3^{-1/6}\frac{1}{\Gamma(2/3)}$ and $\mathrm{Bi}'(0) = 3^{1/6}\frac{1}{\Gamma(1/3)}$

```
bi0 :: (Value v) ⇒ v
bi0 = 3**(−1/6)/sf_gamma(2/3)

bi'0 :: (Value v) ⇒ v
bi'0 = 3**(1/6)/sf_gamma(1/3)
```

Series expansion, where $n!!! = \max(n,1)$ for $n \le 2$ and otherwise $n!!! = n\cdot(n-3)!!!$:

$$\mathrm{Bi}(z) = \mathrm{Bi}(0)\left(\sum_{n=0}^{\infty}\frac{(3n-2)!!!}{(3n)!}z^{3n}\right) + \mathrm{Bi}'(0)\left(\frac{(3n-1)!!!}{(3n+1)!}z^{3n+1}\right)$$

```
airy_bi_series z =
    let z3 = z^3
        biterms  = ixiter 0 1 $ λn t → t*z3*((#)$3*n+1)/((#)$(3*n+1)*(3*n+2)*(3*n+3))
        bi'terms = ixiter 0 z $ λn t → t*z3*((#)$3*n+2)/((#)$(3*n+2)*(3*n+3)*(3*n+4))
    in bi0 * (ksum biterms) + bi'0 * (ksum bi'terms)
```

12 Riemann zeta function

12.1 Preamble

```
{−# Language BangPatterns #−}
module Zeta (sf_zeta, sf_zeta_m1) where
import Gamma
import Trig
import Util
```

12.2 Zeta

The Riemann zeta function is defined by power series for $\Re z > 1$

$$\zeta(z) = \sum_{n=1}^{\infty} n^{-z}$$

and defined by analytic continuation elsewhere.

12.2.1 sf_zeta z

Compute the Riemann zeta function **sf_zeta z** $= \zeta(z)$ where

sf_zeta z $= \zeta(z)$

```
sf_zeta :: (Value v) ⇒ v → v
sf_zeta z
  | z==1       = (1/0)
  | (re z)<0   = 2 * (2*pi)**(z−1) * (sf_sin$pi*z/2) * (sf_gamma$1−z) * (sf_zeta$1−z)
  | otherwise = zeta_series 1.0 z
```

12.2.2 sf_zeta_m1 z

For numerical purposes, it is useful to have **sf_zeta_m1 z** $= \zeta(z) - 1$.

sf_zeta_m1 z $= \zeta(z) - 1$

```
sf_zeta_m1 :: (Value v) ⇒ v → v
sf_zeta_m1 z
  | z==1       = (1/0)
  | (re z)<0   = 2 * (2*pi)**(z−1) * (sf_sin$pi*z/2) * (sf_gamma$1−z) * (sf_zeta$1−z) − 1
−− TODO:
  | otherwise = zeta_series 0.0 z
```

***zeta_series i z**

We use the simple series expansion for $\zeta(z)$ with an Euler-Maclaurin correction:

$$\zeta(z) = \sum_{n=1}^{N} \frac{1}{n^z} + \sum_{k=1}^{p} \cdots$$

zeta_series init z =

```
zeta_series :: (Value v) ⇒ v → v → v
zeta_series !init !z =
  let terms = map (λn→((#)n)**(−z)) [2..]
      corrs = map correction [2..]
  in summer terms corrs init 0.0 0.0
  where
    −−TODO: convert to use kahan summer
    summer !(t:ts) !(c:cs) !s !e !r =
```

```
zeta_series init z =  (cont)

      let !y = t + e
          !s' = s + y
          !e' = (s - s') + y
          !r' = s' + c + e'
      in if r==r' then r'
         else summer ts cs s' e' r'
    !zz1 = z/12
    !zz2 = z*(z+1)*(z+2)/720
    !zz3 = z*(z+1)*(z+2)*(z+3)*(z+4)/30240
    !zz4 = z*(z+1)*(z+2)*(z+3)*(z+4)*(z+5)*(z+6)/1209600
    !zz5 = z*(z+1)*(z+2)*(z+3)*(z+4)*(z+5)*(z+6)*(z+7)*(z+8)/239500800
    correction !n' =
      let n=(#)n'
      in n**(1-z)/(z-1) - n**(-z)/2
         + n**(-z-1)*zz1 - n**(-z-3)*zz2 + n**(-z-5)*zz3
         - n**(-z-7)*zz4 + n**(-z-9)*zz5
```

13 Elliptic functions

13.1 Preamble

```
{-# Language BangPatterns #-}
module Elliptic where
import AGM
import Exp
import Trig
import Util
```

$2^{-2/3}$

```
two23 :: Double
!two23 = 0.62996052494743658238
```

13.2 Elliptic integral of the first kind

Assume that $1-\sin^2\phi, 1-k^2\sin^2\phi \in \mathbb{C}\setminus(-\infty,0]$ except that one of them may be 0.

The elliptic integral of the first kind is defined by

$$F(\phi,k) = \int_0^\phi \frac{d\theta}{\sqrt{1-k^2\sin^2\theta}} = \int_0^{\sin\phi} \frac{dt}{\sqrt{1-t^2}\sqrt{1-k^2t^2}}$$

The complete integral is given by $\phi=\pi/2$:

$$K(k) = F(\pi/2,k) =$$

13.2.1 `sf_elliptic_k k`

Compute the complete elliptic integral of the first kind $K(k)$ To evaluate this, we use the AGM relation

$$K(k) = \frac{\pi}{2\,\mathrm{agm}(1,k')} \qquad \text{where } k' = \sqrt{1-k^2} \qquad K(k)$$

TODO: UNTESTED!

sf_elliptic_k k $= K(k)$

```
sf_elliptic_k :: Double → Double
sf_elliptic_k k =
  let an = sf_agm' 1.0 (sf_sqrt $ 1.0-k^2)
  in pi/(2*an)
```

13.2.2 sf_elliptic_f phi k

Compute the (incomplete) elliptic integral of the first kind $F(\phi, k)$. To evaluate, we use an ascending Landen transformation:

$$F(\phi, k) = \frac{2}{1+k} F(\phi_2, k_2) \qquad \text{where } k_2 = \frac{2\sqrt{k}}{1+k} \text{ and } 2\phi_2 = \phi + \arcsin(k \sin\phi) \tag{$F(\phi,k)$}$$

Note that $0 < k < 1$ and $0 < \phi \leq \pi/2$ imply $k < k_2 < 1$ and $\phi_2 < \phi$. We iterate this transformation until we reach $k = 1$ and use the special case

$$F(\phi, 1) = \mathrm{gud}^{-1}(\phi)$$

(Where $\mathrm{gud}^{-1}(\phi)$ is the inverse Gudermannian function (TODO)). TODO: UNTESTED!

sf_elliptic_f phi k $= F(\phi, k)$

```
sf_elliptic_f :: Double → Double → Double
sf_elliptic_f phi k
  | k==0 = phi
  | k==1 = sf_log((1 + (sf_sin phi)) / (1 - (sf_sin phi))) / 2
        -- quad(@(t)(1/sqrt(1-k^2*sin(t)^2)), 0, phi)
  | phi==0 = 0
  | otherwise =
      ascending_landen phi k 1 $ λ phi' res' →
        res' * sf_log((1 + (sf_sin phi)) / (1 - (sf_sin phi))) / 2
  where
    ascending_landen phi k res kont =
      let k' = 2 * (sf_sqrt k) / (1 + k)
          phi' = (phi + (asin (k*(sin phi))))/2
          res' = res * 2/(1+k)
      in if k'==1 then kont phi' res
         else ascending_landen phi' k' res' kont
    --function res = agm_method(phi, k)
    --  [an,bn,cn,phin] = sf_agm(1.0, sqrt(1 - k^2), phi, k);
    --  res = phin(end) / (2^(length(phin)-1) * an(end));
    --endfunction
```

13.3 Elliptic integral of the second kind

Assume that $1 - \sin^2\phi, 1 - k^2\sin^2\phi \in \mathbb{C} \setminus (-\infty, 0]$ except that one of them may be 0.

Legendre's (incomplete) elliptic integral of the second kind is defined via

$$E(\phi, k) = \int_0^\phi \sqrt{1 - k^2 \sin^2\theta}\, d\theta = \int_0^{\sin\phi} \frac{\sqrt{1-k^2t^2}}{\sqrt{1-t^2}}\, dt$$

The complete integral is

$$E(k) = E(\pi/2, k) =$$

13.3.1 `sf_elliptic_e k`

Compute the complete elliptic integral of the second kind $E(k)$. We evaluate this with an agm-based approach:

$$\ldots$$

TODO: UNTESTED!

`sf_elliptic_e k` $= E(k)$

```
sf_elliptic_e :: Double → Double
sf_elliptic_e k =
  let phi = k
      (as,bs,cs') = sf_agm 1.0 (sf_sqrt (1.0 - k^20))
      cs = k:(tail.reverse$cs')
      res = foldl (-) 2 (map (λ(i,c)→2^(i-1)*c^2) (zip [1..] cs))
  in res * pi/(4*(head as))
```

13.3.2 `sf_elliptic_e_ic phi k`

Compute the incomplete elliptic integral of the second kind $E(\phi, k)$ We evaluate this with an ascending Landen transformation:

$$\ldots$$

TODO: UNTESTED! (Note: could try direct quadrature of the integral, also there is an AGM-based method).

`sf_elliptic_e_ic phi k` $= E(\phi, k)$

```
sf_elliptic_e_ic :: Double → Double → Double
sf_elliptic_e_ic phi k
   | k==1 = sf_sin phi
   | k==0 = phi
   | otherwise = ascending_landen phi k
  where
    ascending_landen phi 1 = sin phi
    ascending_landen phi k =
      let !k' = 2*(sf_sqrt k) / (k+1)
          !phi' = (phi + (sf_asin (k*(sf_sin phi))))/2
      in (1+k)*(ascending_landen phi' k') + (1-k)*(sf_elliptic_f phi' k') - k*(sf_sin phi)
```

13.4 Elliptic integral of the third kind

We define Legendre's (incomplete) elliptic integral of the third kind via

$$\Pi(\phi, \alpha^2, k) = \int_0^{\phi} \frac{d\theta}{\sqrt{1 - k^2 \sin^2\theta}(1 - \alpha^2 \sin^2\theta)} = \int_0^{\sin\phi} \frac{dt}{\sqrt{1-t^2}\sqrt{1-k^2t^2}(1-\alpha^2t^2)}$$

The complete integral of the third kind is given by

$$\Pi(\alpha^2, k) = \Pi(\pi/2, \alpha^2, k) =$$

13.4.1 `sf_elliptic_pi c k`

Compute the complete elliptic integral of the third kind ($c = \alpha^2$ in DLMF notation) for real values only $0 < k < 1$, $0 < c < 1$. Uses agm-based approach. (Could also try numerical quadrature `quad(@(t)(1.0/(1-c*sf_sin(t)^2)/sqrt`
TODO: mostly untested

`sf_elliptic_pi c k` $= \Pi(c, k)$

```
sf_elliptic_pi :: Double → Double → Double
sf_elliptic_pi c k = complete_agm k c
  where
    -- -\infty < k^2 < 1
    -- -\infty < c < 1
    complete_agm k c =
      let (ans,gns,_) = sf_agm 1 (sf_sqrt (1.0-k^2))
          pn1 = sf_sqrt (1-c)
          qn1 = 1
          an1 = last ans
          gn1 = last gns
          en1 = (pn1^2 - an1*gn1) / (pn1^2 + an1*gn1)
      in iter pn1 en1 (reverse ans) (reverse gns) [qn1]

    iter pnm1 enm1 [an] [gn] qns = pi/(4*an) * (2 + c/(1-c)*(ksum qns))
    iter pnm1 enm1 (anm1:an:ans) (gnm1:gn:gns) (qnm1:qns) =
      let pn = (pnm1^2 + anm1*gnm1)/(2*pnm1)
          en = (pn^2 - an*gn) / (pn^2 + an*gn)
          qn = qnm1 * enm1/2
      in iter pn en (an:ans) (gn:gns) (qn:qnm1:qns)
```

13.4.2 `sf_elliptic_pi_ic phi c k`

`sf_elliptic_pi_ic phi c k` $= \Pi(\phi, c, k)$

```
sf_elliptic_pi_ic :: Double → Double → Double → Double
sf_elliptic_pi_ic 0 c k = 0.0
sf_elliptic_pi_ic phi c k = gauss_transform k c phi
  where
    gauss_transform k c phi =
      if (sf_sqrt (1-k^2))==1
      then let cp=sf_sqrt(1-c)
           in sf_atan(cp*(sf_tan phi)) / cp
      else if (1-k^2/c)==0 -- special case else rho below is zero...
      then ((sf_elliptic_e_ic phi k) - c*(sf_cos phi)*(sf_sin phi)
                / sqrt(1-c*(sf_sin phi)^2))/(1-c)
      else let kp = sf_sqrt (1-k^2)
               k' = (1 - kp) / (1 + kp)
               delta = sf_sqrt(1-k^2*(sf_sin phi)^2)
               psi' = sf_asin((1+kp)*(sf_sin phi) / (1+delta))
               rho = sf_sqrt(1 - (k^2/c))
               c' = c*(1+rho)^2/(1+kp)^2
               xi = (sf_csc phi)^2
               newgt = gauss_transform k' c' psi'
           in (4/(1+kp)*newgt + (rho-1)*(sf_elliptic_f phi k)
                - (sf_elliptic_rc (xi-1) (xi-c)))/rho
```

13.5 Elliptic integral of Legendre's type

The (incomplete) elliptic integral of Legendre's type is defined by

$$D(\phi,k) = \int_0^{\phi} \frac{\sin^2\theta}{\sqrt{1-k^2\sin^2\theta}}\,d\theta = \int_0^{\sin\phi} \frac{t^2}{\sqrt{1-t^2}\sqrt{1-k^2t^2}}\,dt$$

This can be expressed as $D(\phi,k) = (F(\phi,k) - E(\phi,k))/k^2$.

The complete elliptic integral of Legendre's type is

$$D(k) = D(\pi/2,k) = (K(k) - E(k))/k^2$$

13.5.1 `sf_elliptic_d_ic phi k`

We simply reduce to $F(\phi,k)$ and $E(\phi,k)$.

`sf_elliptic_d_ic phi k` $= D(\phi,k)$

```
sf_elliptic_d_ic :: Double → Double → Double
sf_elliptic_d_ic phi k = ((sf_elliptic_f phi k) − (sf_elliptic_e_ic phi k)) / (k^2)
```

13.5.2 `sf_elliptic_d_ic phi k`

We simply reduce to $K(k)$ and $E(k)$.

`sf_elliptic_d k` $= D(k)$

```
sf_elliptic_d :: Double → Double
sf_elliptic_d k = ((sf_elliptic_k k) − (sf_elliptic_e k)) / (k^2)
```

13.6 Burlisch's elliptic integrals

DLMF: "Bulirschs integrals are linear combinations of Legendres integrals that are chosen to facilitate computational application of Bartkys transformation"

13.6.1 `sf_elliptic_cel kc p a b`

Compute Burlisch's elliptic integral where $p \neq 0$, $k_c \neq 0$.

$$cel(k_c,p,a,b) = \int_0^{\pi/2} \frac{a\cos^2\theta + b\sin^2\theta}{\cos^2\theta + p\sin^2\theta} \frac{1}{\sqrt{\cos^2\theta + k_c^2\sin^2\theta}}\,d\theta \qquad cel(k_c,p,a,b)$$

TODO: UNTESTED!

`sf_elliptic_cel kc p a b` $= cel(k_c,p,a,b)$

```
sf_elliptic_cel :: Double → Double → Double → Double → Double
sf_elliptic_cel kc p a b = a ∗ (sf_elliptic_rf 0 (kc^2) 1) + (b−p∗a)/3 ∗
(sf_elliptic_rj 0 (kc^2) 1 p)
```

13.6.2 `sf_elliptic_el1 x kc`

Compute Burlisch's elliptic integral

$$el_1(x, k_c) =$$

TODO: UNTESTED!

`sf_elliptic_el1 k kc` $= el_1(x, k_c)$

```
sf_elliptic_el1 :: Double → Double → Double
sf_elliptic_el1 x kc =
  --sf_elliptic_f (atan x) (sf_sqrt(1-kc^2))
  let r = 1/x^2
  in sf_elliptic_rf r (r+kc^2) (r+1)
```

13.6.3 `sf_elliptic_el2 x kc a b`

Compute Burlisch's elliptic integral

$$el_2(x, k_c, a, b) = \int_0^{\arctan x} \frac{a + b\tan^2\theta}{\sqrt{(1+\tan^2\theta)(1+k_c^2\tan^2\theta)}}\,d\theta$$

TODO: UNTESTED!

`sf_elliptic_el2 x kc a b` $= el_2(x, k_c, a, b)$

```
sf_elliptic_el2 :: Double → Double → Double → Double → Double
sf_elliptic_el2 x kc a b =
  let r = 1/x^2
  in a * (sf_elliptic_el1 x kc) + (b-a)/3 * (sf_elliptic_rd r  (r+kc^2) (r+1))
```

13.6.4 `sf_elliptic_el3 x kc p`

Compute the Burlisch's elliptic integral

$$el_3(x, k_c, p) = \int_0^{\arctan x} \frac{d\theta}{(\cos^2\theta + p\sin^2\theta)\sqrt{\cos^2\theta + k_c^2\sin^2\theta}}$$

TODO: UNTESTED!

`sf_elliptic_el3 x kc p` $= el_3(x, k_c, p)$

```
sf_elliptic_el3 :: Double → Double → Double → Double
sf_elliptic_el3 x kc p =
  -- sf_elliptic_pi(atan(x), 1-p, sf_sqrt(1-kc.^2));
  let r = 1/x^2
  in (sf_elliptic_el1 x kc) + (1-p)/3 * (sf_elliptic_rj r (r+kc^2) (r+1) (r+p))
```

13.7 Symmetric elliptic integrals

13.7.1 `sf_elliptic_rc x y`

Compute the symmetric elliptic integral $R_C(x,y)$ for real parameters. Let $x \in \mathbb{C} \setminus (-\infty, 0)$, $y \in \mathbb{C} \setminus \{0\}$, then we define

$$R_C(x,y) = \frac{1}{2}\int_0^\infty \frac{dt}{\sqrt{t+x}(t+y)}$$

(where the Cauchy principal value is taken if $y < 0$.) TODO: UNTESTED!

`sf_elliptic_rc x y` $= R_C(x,y)$

```
-- x≥ 0, y!=0
sf_elliptic_rc :: Double → Double → Double
sf_elliptic_rc x y
  | 0==x  ∧ x<y  = 1/sf_sqrt(y-x) * sf_acos(sf_sqrt(x/y))
  | 0<x   ∧ x<y  = 1/sf_sqrt(y-x) * sf_atan(sf_sqrt((y-x)/x))
  | 0<y   ∧ y<x  = 1/sf_sqrt(x-y) * sf_atanh(sf_sqrt((x-y)/x))
                 -- = 1/sf_sqrt(x-y) * sf_log((sf_sqrt(x) + sf_sqrt(x-y))/sf_sqrt(y))
  | y<0   ∧ 0≤x  = 1/sf_sqrt(x-y) * sf_log((sf_sqrt(x)+sf_sqrt(x-y))/sf_sqrt(-y))
                 -- = 1/sf_sqrt(x-y) * sf_atanh(sf_sqrt(x/(x-y)))
                 -- = sf_sqrt(x/(x-y)) * (sf_elliptic_rc (x-y) (-y))
  | x == y       = 1/(sf_sqrt x)
  | otherwise    = error "sf_elliptic_rc: domain error"
```

13.7.2 `sf_elliptic_rd x y z`

Compute the symmetric elliptic integral $R_D(x,y,z)$ TODO: UNTESTED!

`sf_elliptic_rc x y z` $= R_D(x,y,z)$

```
-- x,y,z>0
sf_elliptic_rd :: Double → Double → Double → Double
sf_elliptic_rd x y z = let (x',s) = (iter x y z 0.0) in (x'**(-3/2) + s)
  where
    iter x y z s =
      let lam = sf_sqrt(x*y) + sf_sqrt(y*z) + sf_sqrt(z*x);
          s' = s + 3/sf_sqrt(z)/(z+lam);
          x' = (x+lam)*two23
          y' = (y+lam)*two23
          z' = (z+lam)*two23
          mu = (x+y+z)/3;
          eps = foldl1 max (map (λt→abs(1-t/mu)) [x,y,z])
      in if eps<2e-16 ∨ [x,y,z]==[x',y',z'] then (x',s')
         else iter x' y' z' s'
```

13.7.3 `sf_elliptic_rf x y z`

Compute the symmetric elliptic integral of the first kind

$$R_F(x,y,z) = \frac{1}{2}\int_0^\infty \frac{dt}{\sqrt{t+x}\sqrt{t+y}\sqrt{t+z}}$$

TODO: UNTESTED!

sf_elliptic_rf x y z $= R_F(x, y, z)$

```
-- x,y,z>0
sf_elliptic_rf :: Double → Double → Double → Double
sf_elliptic_rf x y z = 1/(sf_sqrt $ iter x y z)
  where
    iter x y z =
      let lam = (sf_sqrt $ x*y) + (sf_sqrt $ y*z) + (sf_sqrt $ z*x)
          mu = (x+y+z)/3
          eps = foldl1 max $ map (λa→abs(1-a/mu)) [x,y,z]
          x' = (x+lam)/4
          y' = (y+lam)/4
          z' = (z+lam)/4
      in if (eps<1e-16) ∨ ([x,y,z]==[x',y',z'])
         then x
         else iter x' y' z'
```

13.7.4 `sf_elliptic_rg x y z`

Compute the symmetric elliptic integral

$$R_G(x, y, z) = \frac{1}{4\pi}\int_0^{2\pi}\int_0^{\pi}\sqrt{x\sin^2\theta\cos^2\phi + y\sin^2\theta\sin^2\phi + z\cos^2\theta}\,\sin\theta\,d\theta\,d\phi$$

TODO: UNTESTED!

sf_elliptic_rg x y z $= R_G(x, y, z)$

```
-- x,y,z>0
sf_elliptic_rg :: Double → Double → Double → Double
sf_elliptic_rg x y z
  | x>y = sf_elliptic_rg y x z
  | x>z = sf_elliptic_rg z y x
  | y>z = sf_elliptic_rg x z y
  | otherwise =
    let !a0 = sqrt (z-x)
        !c0 = sqrt (y-x)
        !h0 = sqrt z
        !t0 = sqrt x
        !(an,tn,cn_sum,hn_sum) = iter 1 a0 t0 c0 (c0^2/2) h0 0
    in ((t0^2 + theta*cn_sum)*(sf_elliptic_rc (tn^2+theta*an^2) tn^2) + h0 + hn_sum)/2
    where
      theta = 1
      iter n an tn cn cn_sum hn hn_sum =
        let an' = (an + sf_sqrt(an^2 - cn^2))/2
            tn' = (tn + sf_sqrt(tn^2 + theta*cn^2))/2
            cn' = cn^2/(2*an')/2
            cn_sum' = cn_sum + 2^((#)n-1)*cn'^2
            hn' = hn*tn'/sf_sqrt(tn'^2+theta*cn'^2)
            hn_sum' = hn_sum + 2^n*(hn' - hn)
            n' = n + 1
        in if cn^2==0 then (an,tn,cn_sum,hn_sum)
           else iter n' an' tn' cn' hn_sum' hn' hn_sum'
```

13.7.5 `sf_elliptic_rj x y z p`

Compute the symmetric elliptic integral

$$R_J(x, y, z, p) = \frac{3}{2}\int_0^\infty \frac{dt}{\sqrt{t+x}\sqrt{t+y}\sqrt{t+z}(t+p)}$$

TODO: UNTESTED!

`sf_elliptic_rj x y z p` $= R_J(x, y, z, p)$

```
-- x,y,z>0
sf_elliptic_rj :: Double -> Double -> Double -> Double -> Double
sf_elliptic_rj x y z p =
  let (x',smm,scale) = iter x y z p 0.0 1.0
  in scale*x'**(-3/2) + smm
  where
    iter x y z p smm scale =
      let lam = sf_sqrt(x*y) + sf_sqrt(y*z) + sf_sqrt(z*x)
          alpha = p*(sf_sqrt(x)+sf_sqrt(y)+sf_sqrt(z)) + sf_sqrt(x*y*z)
          beta = sf_sqrt(p)*(p+lam)
          smm' = smm + (if (abs(1 - alpha^2/beta^2) < 5e-16)
                   then
                     -- optimization to reduce external calls
                     scale*3/alpha;
                   else
                     scale*3*(sf_elliptic_rc (alpha^2) (beta^2))
                   )
          mu = (x+y+z+p)/4
          eps = foldl1 max (map (\t->abs(1-t/mu)) [x,y,z,p])
          x' = (x+lam)*two23/mu
          y' = (y+lam)*two23/mu
          z' = (z+lam)*two23/mu
          p' = (p+lam)*two23/mu
          scale' = scale * (mu**(-3/2))
      in if eps<1e-16 || [x,y,z,p]==[x',y',z',p'] || smm'==smm
         then (x',smm',scale')
         else iter x' y' z' p' smm' scale'
```

14 Spence

Spence's integral for $z \geq 0$ is

$$S(z) = -\int_1^z \frac{\ln t}{t-1}\,dt = -\int_0^{z-1} \frac{ln(1+u)}{z}\,dz$$

and we extend the function via analytic continuation. Spence's function $S(z)$ is related to the dilogarithm function via $S(z) = \mathrm{Li}_2(1-z)$.

14.1 Preamble

```
module Spence (sf_spence) where
import Exp
import Util
```

A useful constant `pi2_6` $= \frac{\pi^2}{6}$

```
pi2_6 :: (Value v) ⇒ v
pi2_6 = pi^2/6
```

14.2 sf_spence z

Compute Spence's integral `sf_spence z` $= S(z)$. We use a variety of transformations to to allow efficient computation with a series.

$$\begin{aligned}
\mathrm{Li}_2(z) + \mathrm{Li}_2(\frac{z}{z-1}) &= -\frac{1}{2}(\ln(1-z))^2 \qquad z \in \mathbb{C} \setminus [1,\infty) \\
\mathrm{Li}_2(z) + \mathrm{Li}_2(\frac{1}{z}) &= -\frac{\pi^2}{6} - \frac{1}{2}(\ln(-z))^2 \qquad z \in \mathbb{C} \setminus [0,\infty) \\
\mathrm{Li}_2(z) + \mathrm{Li}_2(1-z) &= \frac{\pi^2}{6} - \ln(z)\ln(1-z) \qquad 0 < z < 1
\end{aligned}$$

(TODO: this code has not be solidly retested after conversion, especially verify complex.)

`sf_spence z` $= \mathrm{Li}_2(z)$

```
sf_spence :: (Value v) ⇒ v → v
sf_spence z
  | is_nan z      = z
  | (re z)<0      = 0/0
  | z == 0        = pi2_6
  | (rabs z)<0.5 = (series z) + (pi2_6 − (sf_log z)*(sf_log (1−z)))
  | (rabs z)<1.0 = −(series (1−z))
  | (rabs z)<2.5 = (series ((z−1)/z)) − (sf_log z)^2/2
  | otherwise    = (series (1/(1−z))) − pi2_6 − (sf_log (z−1))^2/2
```

*series z

The series expansion used for Spence's integral:

$$\texttt{series z} = -\sum_{k=1}^{\infty} \frac{z^k}{k^2}$$

```
series z =
  let zk = iterate (*z) z
      terms = zipWith (λ t k → −t/(#)k^2) zk [1..]
  in ksum terms
```

15 Lommel functions

15.1 Preamble

```
module Lommel (
  sf_lommel_s,
  sf_lommel_s2,
) where
import Util
```

–TODO: These are completely untested!

15.2 First Lommel function

For $\mu \pm \nu \neq \pm 1, \pm 3, \pm 5, \cdots$ we define the first Lommel function `sf_lommel_s mu nu z` $= S_{\mu,\nu}(z)$ via series-expansion:

$$S_{\mu,\nu}(z) = \frac{z^{mu+1}}{(\mu+1)^2 - \nu^2} \sum_{k=0}^{\infty} t_k$$

where

$$t_0 = 1 \qquad t_k = t_{k-1} \frac{-z^2}{(\mu+2k+1)^2 - \nu^2}$$

15.2.1 `sf_lommel_s mu nu z`

`sf_lommel_s mu nu z` $= S_{\mu,\nu}(z)$

```
sf_lommel_s mu nu z =
  let terms = ixiter 1 1.0 $ λ k t → −t*z^2 / ((mu+((#)$2*k+1))^2 − nu^2)
      res = ksum terms
  in res * z**(mu+1) / ((mu+1)^2 − nu^2)
```

15.3 Second Lommel function

For $\mu \pm \nu \neq \pm 1, \pm 3, \pm 5, \cdots$ the second Lommel function `sf_lommel_s2 mu nu z` $= s_{\mu,\nu}(z)$ is given via an asymptotic expansion:

$$s_{\mu,\nu}(z) \sim \sum_{k=0}^{\infty} u_k$$

where

$$u_0 = 1 \qquad u_k = u_{k-1} \frac{-(\mu-2k+1)^2 - \nu^2}{z^2}$$

15.3.1 `sf_lommel_s2 mu nu z`

`sf_lommel_s2 mu nu z` $= s_{\mu,\nu}(z)$

```
sf_lommel_s2 mu nu z =
  let tterms = ixiter 1 1.0 $ λ k t → −t*((mu−((#)$2*k+1))^2 − nu^2) / z^2
      terms = tk tterms
      res = ksum terms
  in res
  where tk (a:b:cs) = if (rabs a)<(rabs b) then [a] else a:(tk$b:cs)
```