Report on the Numerical Inversion of Laplace Tranforms of Probabilistic Functions: A Haskell Implementation

Jeremy Bradley*

Last updated: 24 July 2001†

1 Introduction

This is an experience paper[1], we document our implementation and usage of the excellent work of Abate, Whitt and Choudhury. Between them, they have written widely on the subject of inverting Laplace transforms with specific application to probability distributions for over 30 years and we found their work particularly accessible.

Briefly summarising their main publications: the Fourier-series techniques are presented in detail in [4] concentrating on the Bromwich and Post-Widder inversion formulae. Actual imperative implementations of the Euler and Post-Widder algorithms including numerical error analysis are covered in [1]. A revised inversion technique using Laguerre series expansion is presented in [2]. Finally, an excellent summary chapter of the main techniques together with probabilistic applications can be found in [3].

Whereas the algorithms implemented in [1] were originally written in UBASIC, we have reimplemented the central Euler and Post-Widder algorithms in a functional language, namely Haskell. We have also empirically found improved numerical parameters for the two algorithms across three types of probability function[2].

*Department of Computer Science, University of Durham, South Road, Durham DH1 3LE

†From the original report of 3 November 2000

[1]There also exists a short presentation documenting a restricted set of these results [6].

[2]It is not clear whether this is due to the specific Haskell architecture or whether this applies in general to other implementations.

We also show that transforms of more difficult functions (such as, discontinuous and discontinuous derivative functions) can be inverted if care is used and the features of the algorithms are understood.

Overall our experience of using these methods is very positive. What is usually thought of as a one way mathematical function can indeed be used entirely practically with some confidence that good numerical results can be obtained. This fact alone should, we feel, encourage more people to use Laplace transforms in probabilistic applications.

2 Characteristic Functions

We select three functions which are common and useful in stochastic processes, a probability density function, a cumulative distribution function and a transient state function. These cases are characteristic, it is hoped, of probabilistic functions that tail off to 0, or tend to a specific non-zero value. They also specifically represent a smooth (continuous and differentiably continuous) function—the exponential pdf; a discontinuous and differentiably discontinuous function—the Heaviside function; and a continuous but differentiably discontinuous function—our transient function.

For the latter two cases (as discussed in [4]) the algorithms are not supposed to perform well. It is suggested that convolution smoothing is used to try to smooth over the discontinuities. While this smoothing is probably a very effective technique, we still wish to know how poorly these algorithms perform and what behaviour is likely to occur if presented with such tricky inversion cases.

2.1 Exponential Distribution

For this simple case we select $f(t) = e^{-t}$ and are thus inverting $L(z) = \frac{1}{1+z}$.

In figure 7, we plot the analytic inversion (solid line) against the numerical inversion as calculated by the Euler algorithm. In figure 8, we plot the analytic inversion (solid line) against the numerical inversion as calculated by the Post-Widder algorithm. In figure 9, we plot on a log-scale the relative errors of both Euler and Post-Widder methods.

2.2 Heaviside Function

For this section we select $f(t) = H(t - 5/2)$ and are inverting $L(z) = \frac{e^{-5z/2}}{z}$.

In figure 10, we plot the analytic inversion (solid line) against the numerical inversion as calculated by the Euler algorithm. In figure 11, we plot the analytic inversion (solid line) against the numerical inversion as calculated by the Post-Widder algorithm. In figure 12, we plot on a log-scale the relative errors of both Euler and Post-Widder methods. Note that in this last case, only the error from the discontinuity at 2.5 is displayed as the function value of 0 for $0 \leq t < 2.5$ distorts the relative error calculation.

It is interesting to note the Gibbs phenomenon [7] that occurs either side of the jump in figure 10 and to a lesser extent in figure 11. We note that this cannot be removed, only pushed closer to the discontinuity by altering parameters, which we will discuss in the next section.

2.3 Transient Function

For this section we select $f(t) = \pi_{00}(t)$ where:

$$\pi_{00}(t) = \sum_{r=0}^{n} \frac{(\lambda(t - rd))^n}{n!} e^{-\lambda(t-rd)} \tag{1}$$

for $nd \leq t < (n+1)d$. We have previously calculated this to be one of the transient functions of the system in figure 1, which represents the probability that the system is in state 0 at time t, given that it started from state 0. We

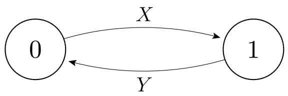

Fig. 1. A simple stochastic system where $X \sim \exp(\lambda)$ and $Y \sim \delta(d)$.

also know from prior analysis that this seemingly complicated time function for $\pi_{00}(t)$ has a relatively simple Laplace form:

$$\hat{\pi}(z) = \frac{1}{\lambda + z - \lambda e^{-dz}} \tag{2}$$

This, then, is what we set the Laplace inverters to work on.

We set $\lambda = 4$ and $d = 1$ for our plots below.

In figure 13, we plot the analytic inversion (solid line) against the numerical inversion as calculated by the Euler algorithm. In figure 14, we plot the analytic inversion (solid line) against the numerical inversion as calculated by the Post-Widder algorithm. In figure 15, we plot on a log-scale the relative errors of both Euler and Post-Widder methods.

3 Euler Truncation Error

The Euler algorithm as presented in [1] provides a calculation of the truncation error. We plot this value for each of the graphs against the actual error in figures 16, 17 and 18.

4 Using Different Numerical Parameters

In the course of experimenting with the Laplace inversion algorithms of [4], we have found that we can obtain improved results if we change some of the key parameter values in both Euler and Post-Widder methods.

It is not clear whether this is an optimisation for our particular implementation in Haskell, however it is still worth discussing.

One of the benefits of using invertible functions (to start with) on these inversion methods is that we can calculate actual errors and assess which parameter values (in this case) work best.

4.1 Euler Method

In [1] an accuracy parameter, $A = 18.2$, is suggested. We experimented with lowering this value and found that we obtained optimal accuracy across our three examples at around $A = 9.5$. Below 9.5, we saw a worsening in accuracy.

Comparisons of the different A-values for the three inversion cases are shown in figures 19, 20 and 21.

4.2 Post-Widder Method

In [1], parameter values of $m = 6$ and $E = 8$ are suggested. As before we found improved accuracy if we altered these values—specifically to $m = 4$ and $E = 2$.

Comparisons of the different m/E-values for the three inversion cases are shown in figures 22, 23 and 24.

5 Example

Finally we look at an example of a system which employs different types of distribution (figure 2).

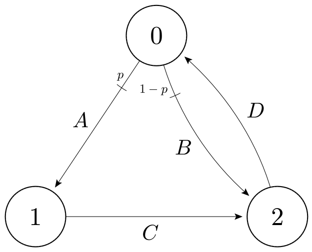

Fig. 2. Lots of distributions.

The distributions used in the transition system of figure 2 are given by:

$$A \sim \Gamma(\alpha, \lambda) \tag{3}$$
$$B \sim \text{Hyper}\left(\frac{2}{3}, \frac{1}{3}; \lambda, \mu\right) \tag{4}$$
$$C \sim \text{U}\left[\frac{1}{\lambda}, \frac{1}{\mu}\right] \tag{5}$$
$$D \sim \delta\left[\frac{1}{\mu}\right] \tag{6}$$

where:

$$L_A(z) = \left(\frac{\lambda}{\lambda + z}\right)^{\alpha} \tag{7}$$
$$L_B(z) = \frac{2}{3}\frac{\lambda}{\lambda + z} + \frac{1}{3}\frac{\mu}{\mu + z} \tag{8}$$
$$L_C(z) = \frac{\lambda\mu}{(\lambda - \mu)z}\left[e^{-z/\lambda} - e^{-z/\mu}\right] \tag{9}$$
$$L_D(z) = e^{-z/\mu} \tag{10}$$

When we come to plot the graphs, the following values will be selected for the parameters: $\alpha = 5/2$, $\lambda = 3/2$, $\mu = 1$ and $p = 3/10$. In previous work [5, section 6] we calculated the Laplace transforms of the transient functions, π_{00}, π_{11} and π_{22} which are not analytically invertible. Figures 3, 4 and 5 show the use of the Euler method. Figure 6 uses the Post-Widder technique and cross-checks figure 5 which displays a slight anomaly at the discontinuity.

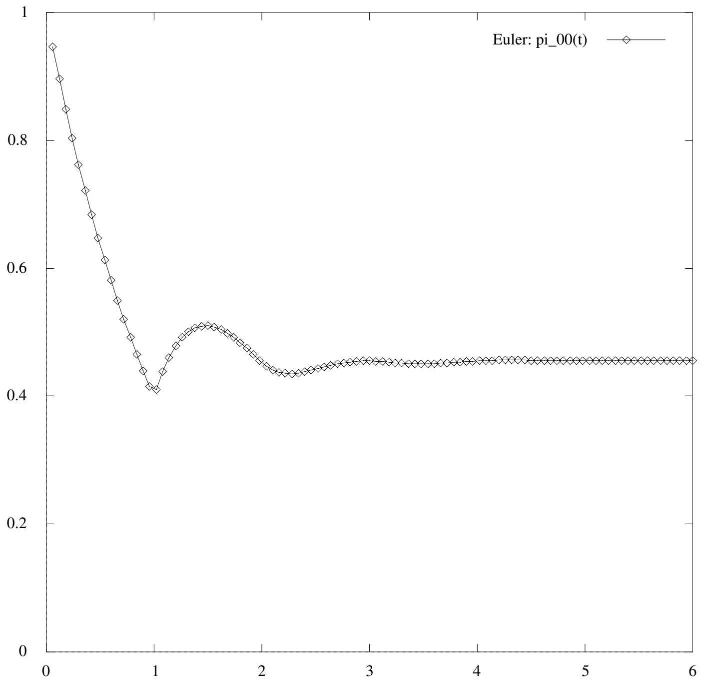

Fig. 3. Euler: Transient function $\pi_{00}(t)$.

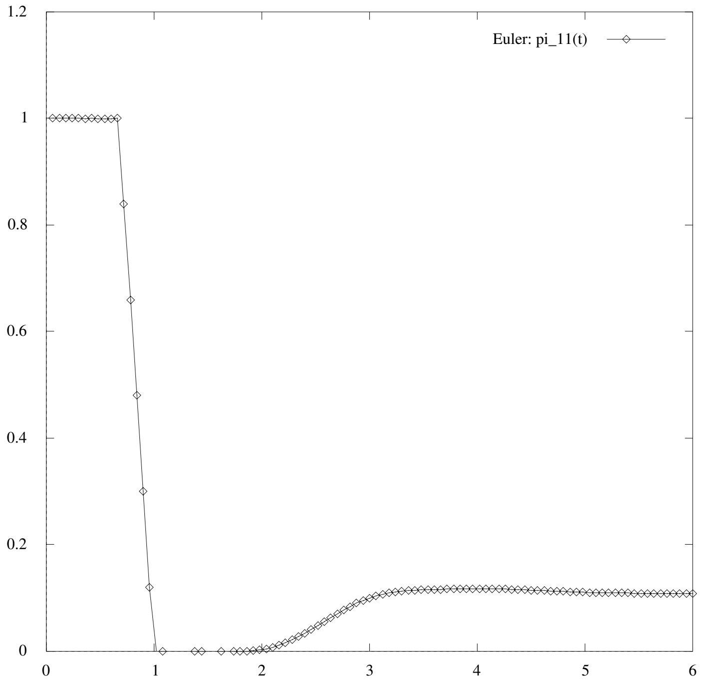

Fig. 4. Euler: Transient function $\pi_{11}(t)$.

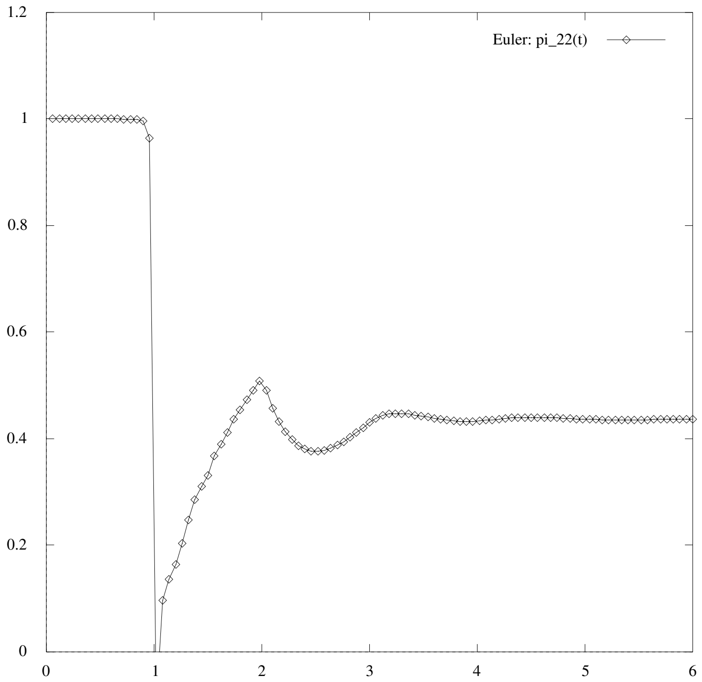

Fig. 5. Euler: Transient function $\pi_{22}(t)$.

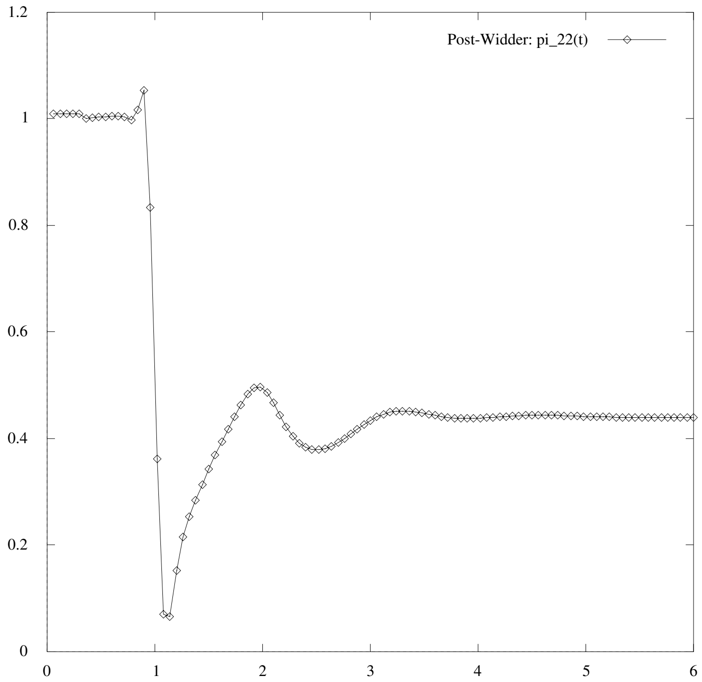

Fig. 6. Post-Widder: Transient function $\pi_{22}(t)$.

6 Conclusion

Despite warnings about inverting functions with known discontinuities (mentioned earlier), we note that the Post-Widder results do agree quite closely with the Euler results, although the accuracy is in general less for Post-Widder in the standard cases. In the specific case where we had a discontinuity in the test transient function (figures 5 and 6) if any Gibbs phenomenon is present it is only to a small extent in the Post-Widder case for $t < 1$. What is useful here is the use of the two methods in conjunction to demonstrate the correct qualitative features of the solution.

References

[1] Abate, J., Choudhury, G. L., and Whitt, W. Numerical inversion of Laplace transforms of probability distributions. *ORSA Journal on Computing 7* (1995), 36–43.

[2] Abate, J., Choudhury, G. L., and Whitt, W. On the Laguerre method for numerically inverting Laplace transforms. *INFORMS Journal on Computing 8* (1996), 413–427.

[3] Abate, J., Choudhury, G. L., and Whitt, W. An introduction to numerical transform inversion and its application to probability models. In *Computational Probability*, W. Grassman, Ed. Kluwer, Boston, 1999, pp. 257–323.

[4] Abate, J., and Whitt, W. The Fourier-series method for inverting transforms of probability distributions. *Queueing Systems 10*, 1 (1992), 5–88.

[5] Bradley, J T Vowden, C. J. Report on extracting transient distributions from semi-Markov processes. Tech. Rep. 2001(8), Department of Computer Science, University of Durham, South Road, Durham, DH1 3LE, UK, July 2001.

[6] Bradley, J. T. Extracting QoS measures from Laplace transforms. Tech. Rep. 2001(7), Department of Computer Science, University of Durham, South Road, Durham, DH1 3LE, UK, July 2001.

[7] Körner, T. W. *Fourier Analysis.* Cambridge University Press, 1989.

A Graphs of Inversion Results

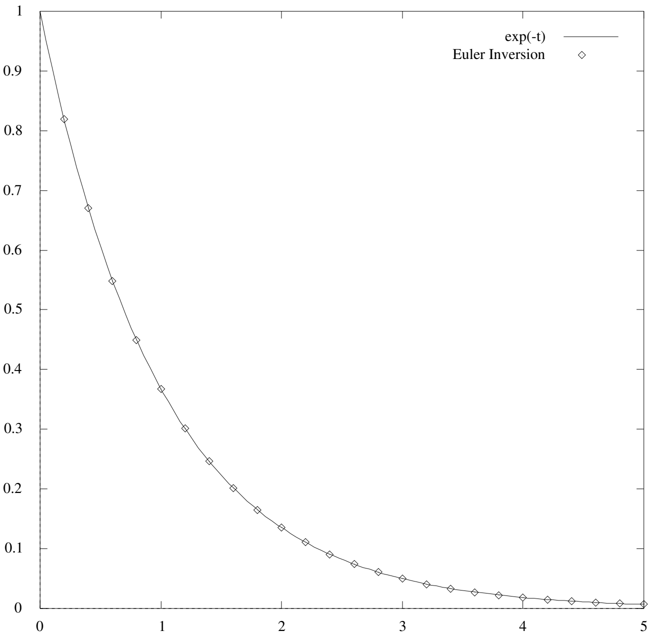

Fig. 7. Numerical Euler inversion (points) and analytic inversion (solid line).

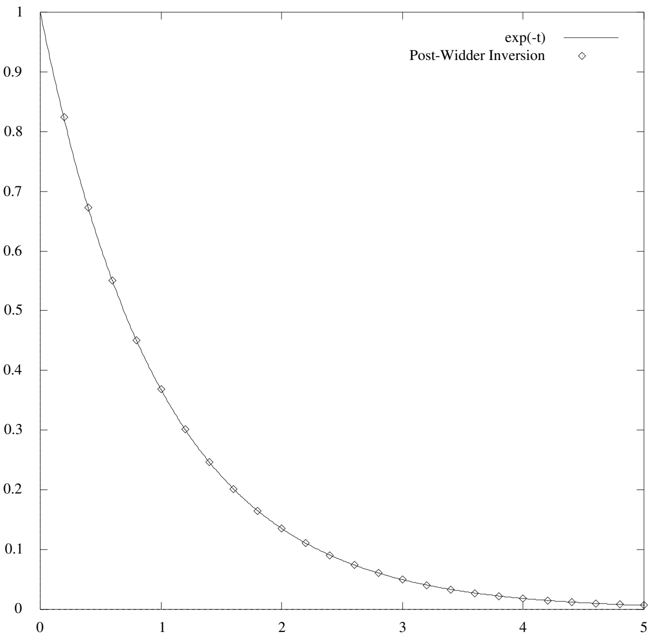

Fig. 8. Numerical Post-Widder inversion (points) and analytic inversion (solid line).

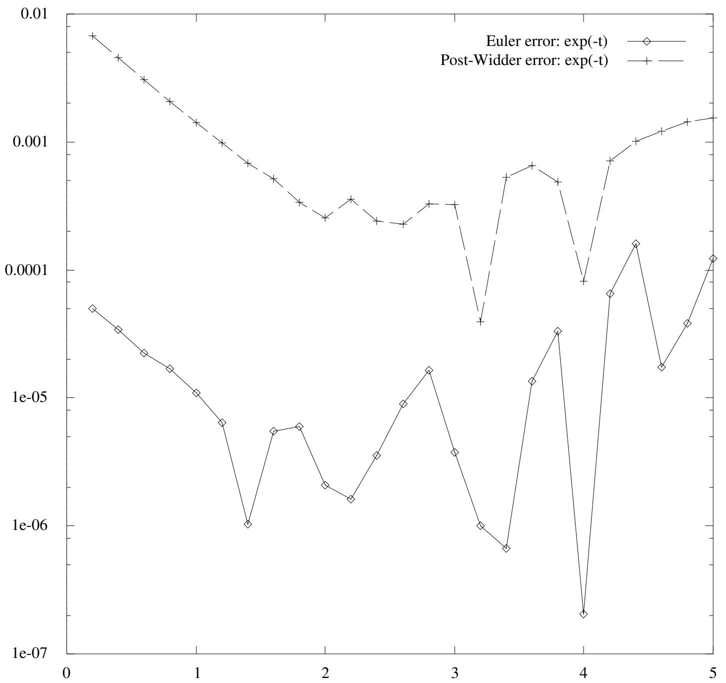

Fig. 9. Log of relative error for both Euler and Post-Widder cases.

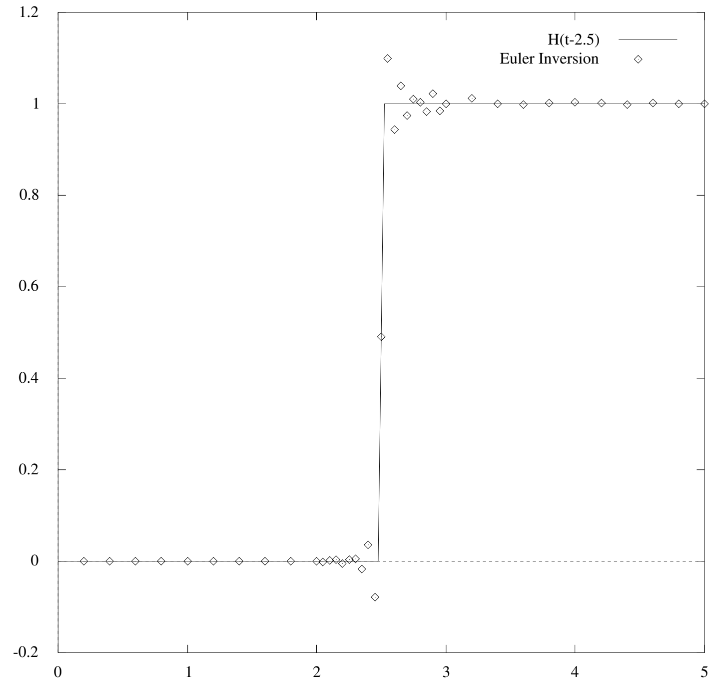

Fig. 10. Numerical Euler inversion (points) and analytic inversion (solid line).

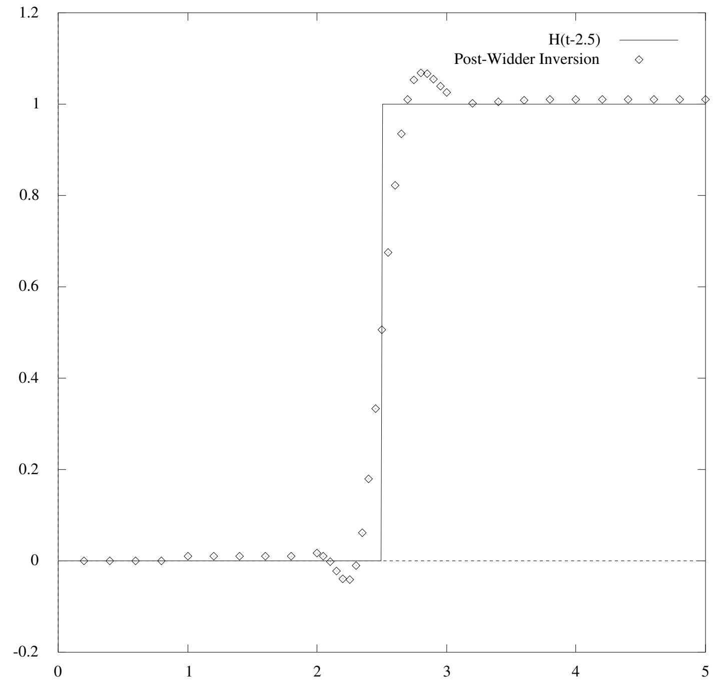

Fig. 11. Numerical Post-Widder inversion (points) and analytic inversion (solid line).

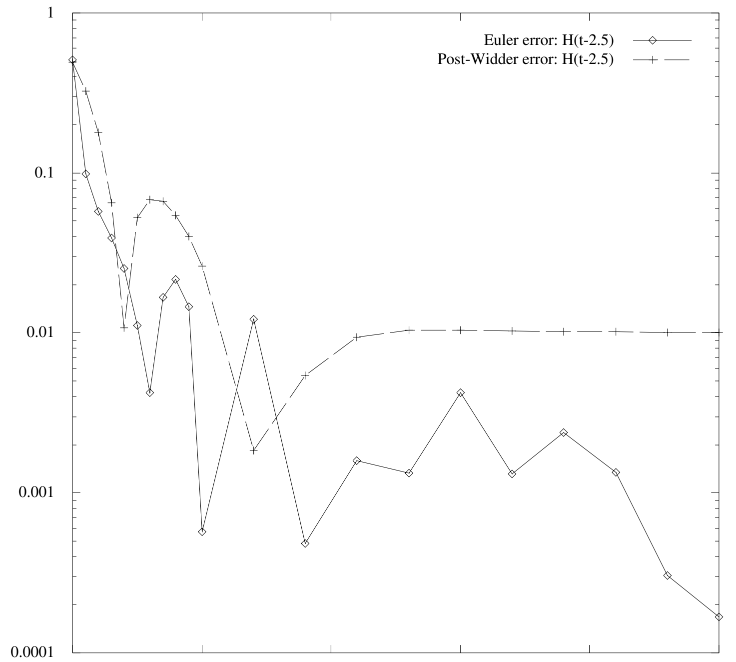

Fig. 12. Log of relative error for both Euler and Post-Widder cases.

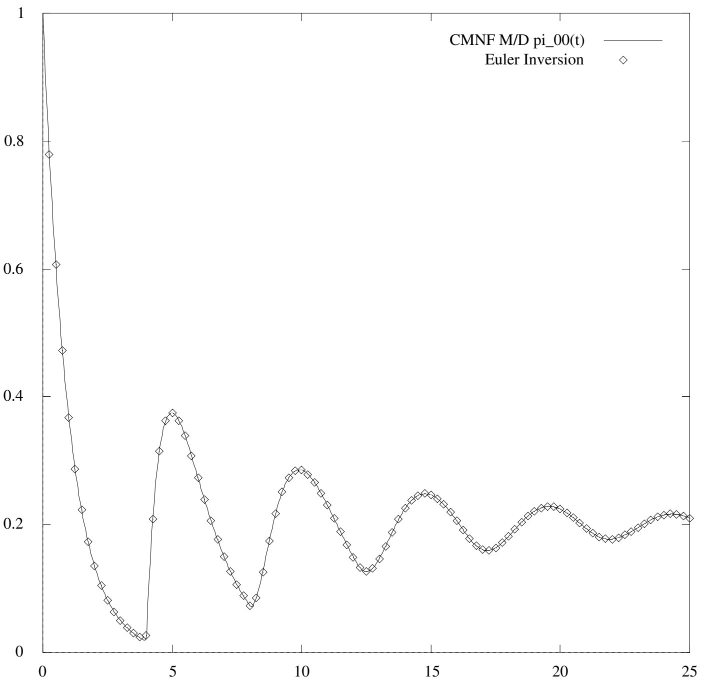

Fig. 13. Numerical Euler inversion (points) and analytic inversion (solid line).

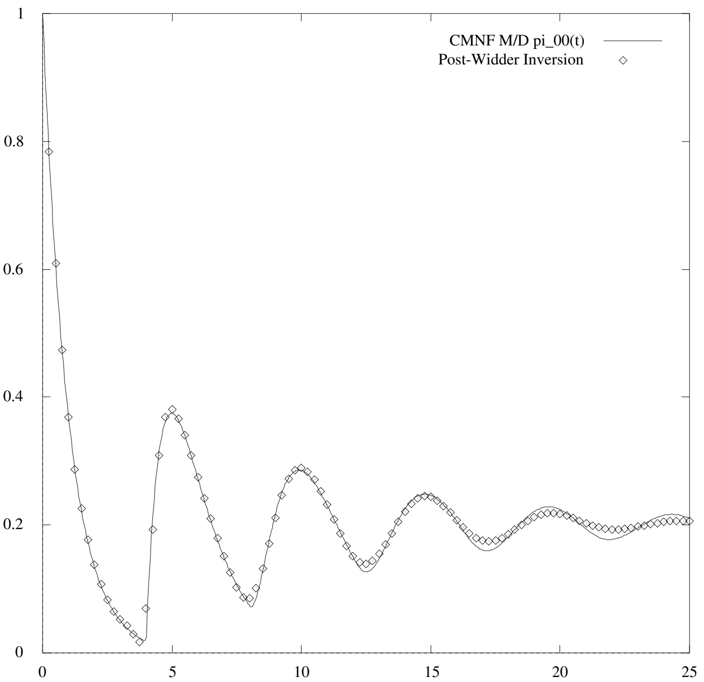

Fig. 14. Numerical Post-Widder inversion (points) and analytic inversion (solid line).

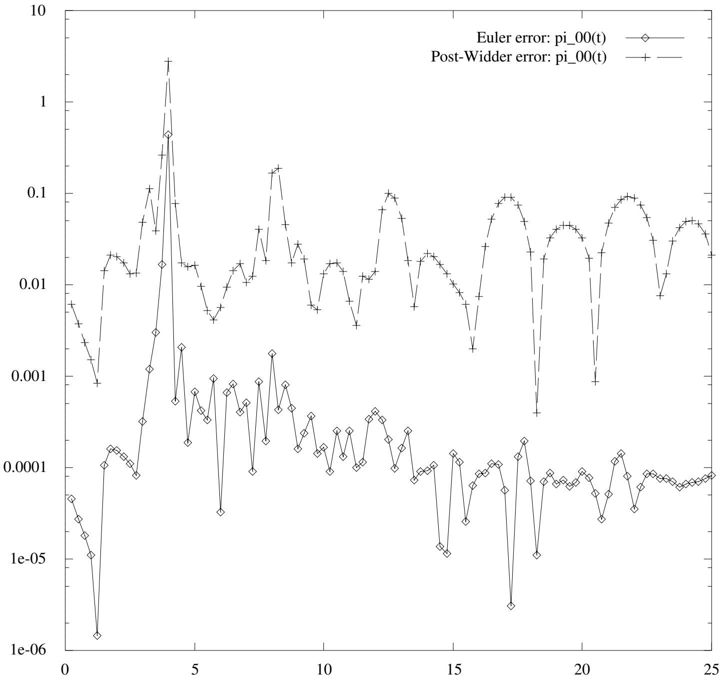

Fig. 15. Log of relative error for both Euler and Post-Widder cases.

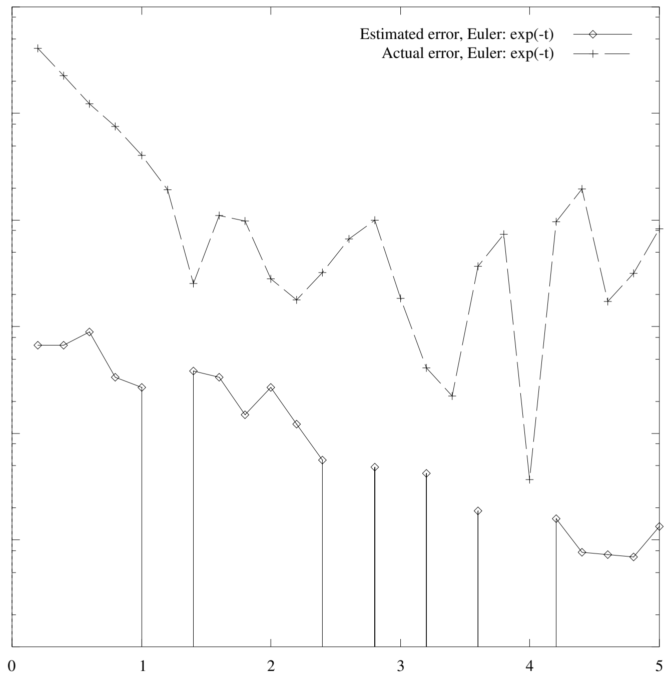

Fig. 16. Actual error against predicted Euler truncation error for the exponential case.

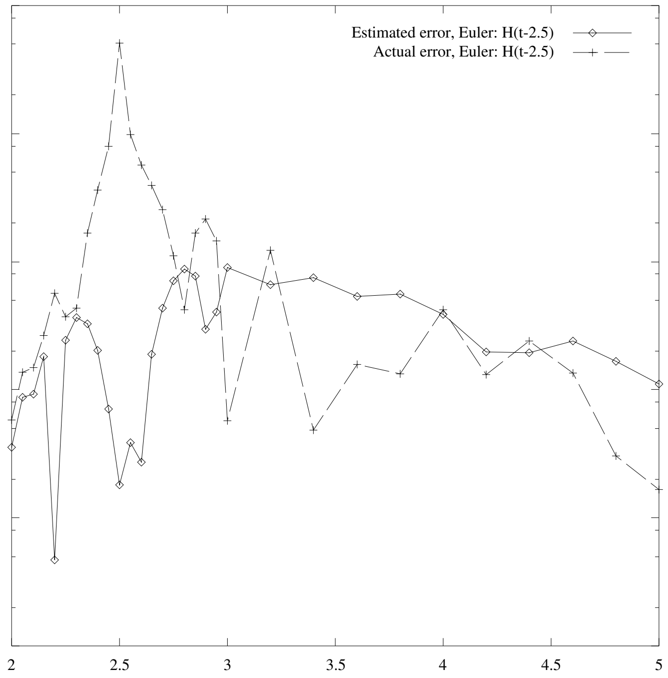

Fig. 17. Actual error against predicted Euler truncation error for the Heaviside case.

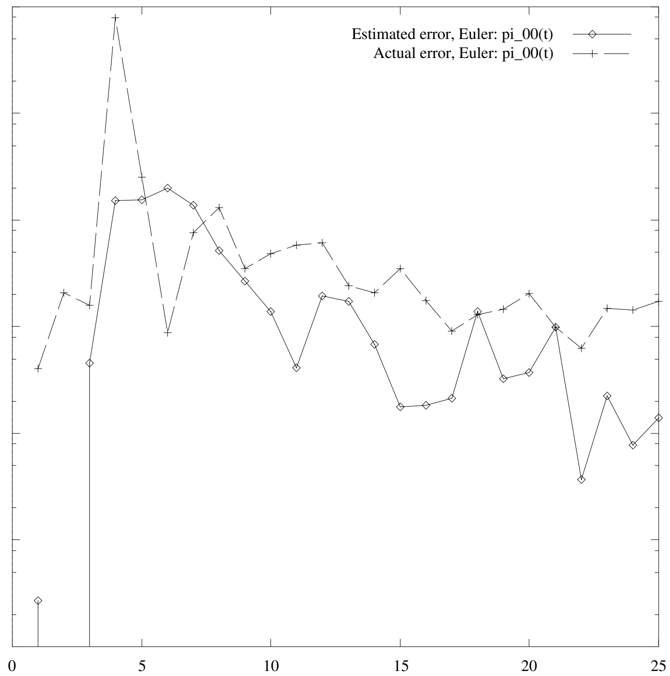

Fig. 18. Actual error against predicted Euler truncation error for the transient case.

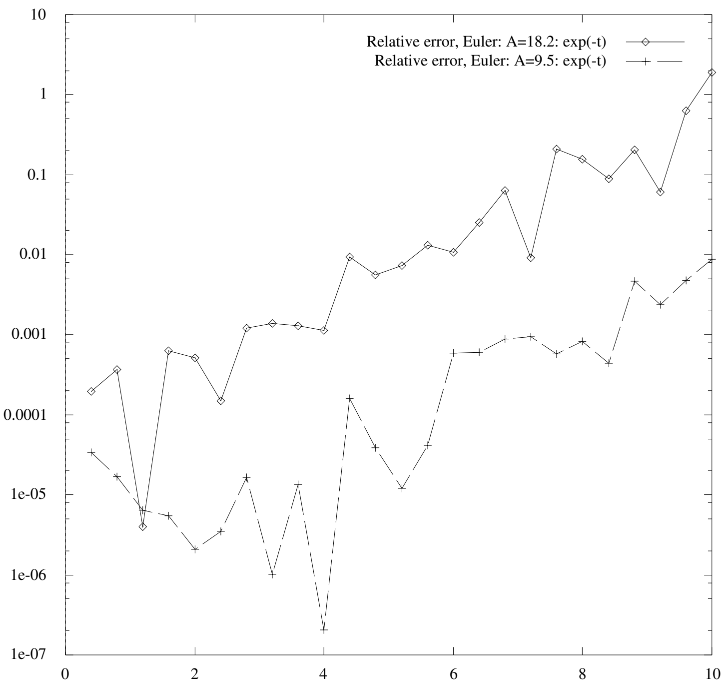

Fig. 19. Log of relative error in exponential case for Euler inversion using different parameter values.

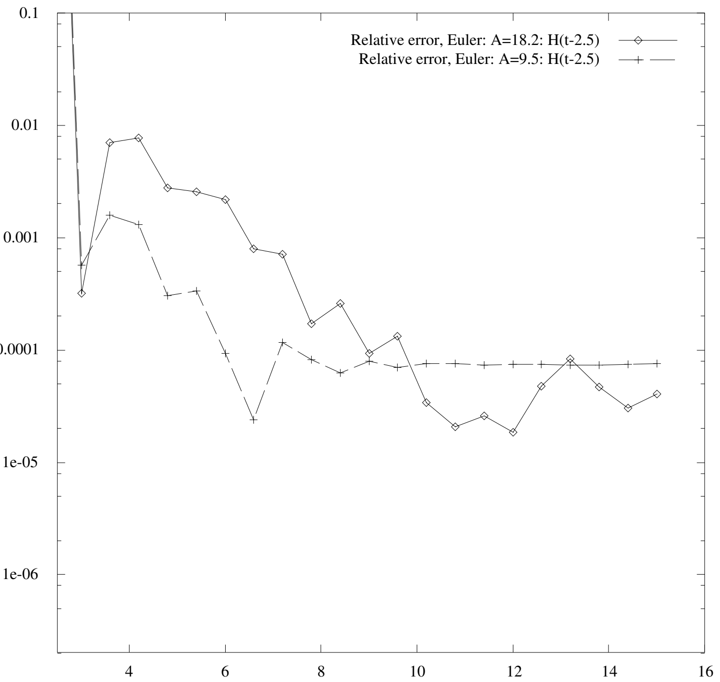

Fig. 20. Log of relative error in Heaviside case for Euler inversion using different parameter values.

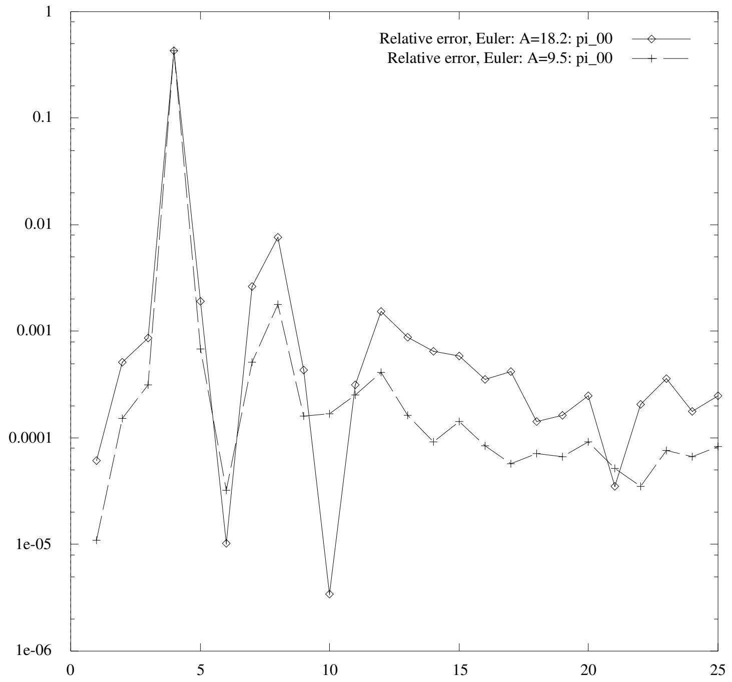

Fig. 21. Log of relative error in transient function case for Euler inversion using different parameter values.

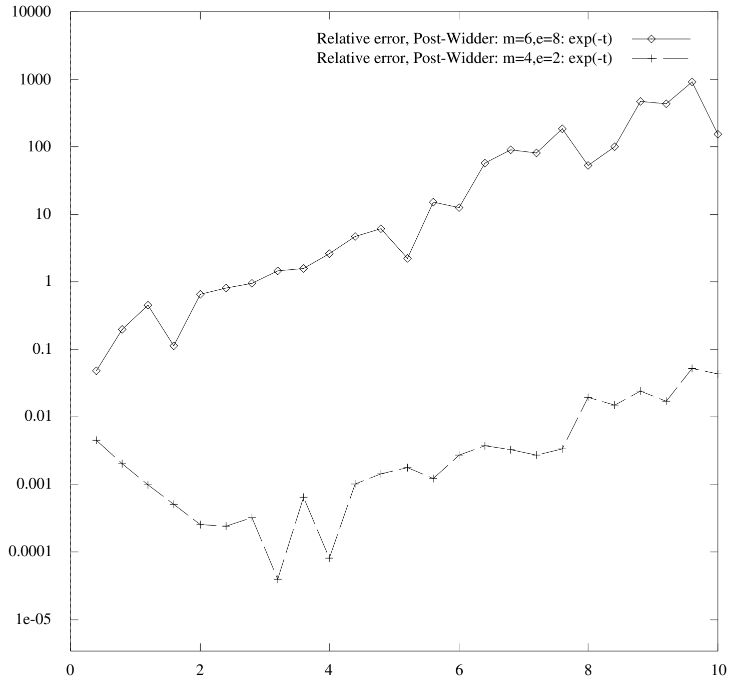

Fig. 22. Log of relative error in exponential case for Post-Widder inversion using different parameter values.

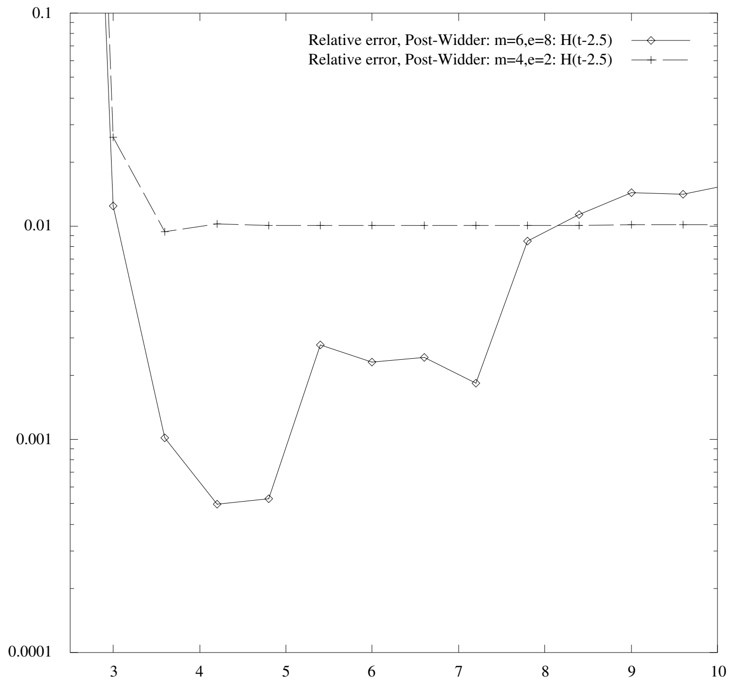

Fig. 23. Log of relative error in Heaviside case for Post-Widder inversion using different parameter values.

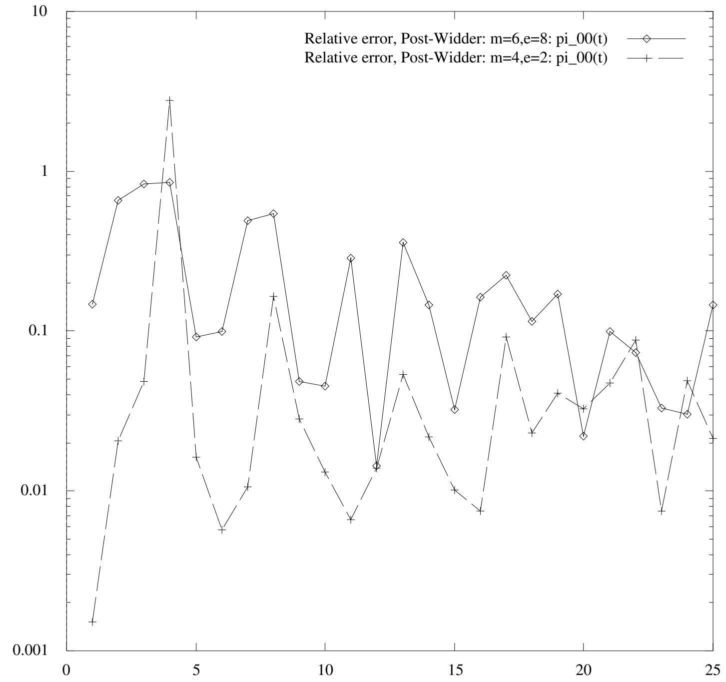

Fig. 24. Log of relative error in transient function case for Post-Widder inversion using different parameter values.

B Source Code

B.1 LaplaceInvert.hs

```
module LaplaceInvert (
    LaplaceT, TimeF, DomainSpec, InvAlgorithm(..), invertLaplace
    ) where

-- Matrix v0.1a (HS98); (c)010701; jtb.
--(c) 2001 Jeremy Bradley, University of Durham

import Matrix (Matrix)
import Utils (factorial, choose, isNothing, (?))
import Complex
import List (transpose)
import Maybe (fromJust)

type LaplaceT = (Complex Double -> Complex Double)
type TimeF = (Double -> Double)
type Domain = [Double]
type DomainSpec = [((Double, Double), Int)]
data InvAlgorithm = Euler | PostWidder -- | Laguerre

-- general calling interface

invertLaplace :: InvAlgorithm -> LaplaceT -> Maybe TimeF
    -> DomainSpec -> Matrix Double
invertLaplace algo f_z mf dm
    = let
        ts = makeDomain dm
        f_t = fromJust mf
        xs = [f_t t | t <- ts]
    in case (algo) of
        Euler -> if (isNothing mf)
            then transpose mtx
            else transpose mtx' where
                mtx = eulerInv f_z ts
                mtx' = mtx ++ (xs:makeErrorLists mtx xs)
        PostWidder -> if (isNothing mf)
```

```
            then transpose mtx
            else transpose mtx' where
                mtx = postWidderInv f_z ts
                mtx' = mtx ++ (xs:makeErrorLists mtx xs)
--      Laguerre -> if (isNothing mf)
--          then transpose mtx
--          else transpose mtx' where
--              mtx = laguerreInv f_z ts
--              mtx' = mtx ++ (xs:makeErrorLists mtx xs)


-- generic domain specification -> domain list
makeDomain :: DomainSpec -> Domain
makeDomain = concat . map getDomain where
    getDomain :: ((Double, Double), Int) -> Domain
    getDomain ((tR1,tR2),step)
        =[tR1+(tR2-tR1)*(fromInt u)/(fromInt step)|u<-[1..step]]

-- obtain absolute and relative errors of numerical inversion
makeErrorLists :: Matrix Double -> [Double] -> Matrix Double
makeErrorLists mtx xs = (as:[rs]) where
    as = map (\(x,y) -> abs (x-y)) $ zip xs is
        -- absolute error
    rs = map (\(x,y) -> (y==0) ? (x/0.00001, x/y)) $ zip as xs
        -- relative error
    is = head $ tail mtx    -- invertion list

-- EULER algorithm

-- Direct translation of the formulae from Abate & Whitt 1991
-- [used to check efficient FILT version: checked 001031]

-- s_n :: Int -> Double -> Double -> Double
-- s_n n a_ t = exp(0.5*a_)/t * (s1 + s2) where
--     s1 = 0.5 * (realPart $ f_z (0.5*a_/t :+ 0))
--     s2 = sum [(-1)^k * (a_k k a_ t) | k <- [1..n]]
--
-- e_mnt :: Int -> Int -> Double -> Double -> Double
-- e_mnt m n a_ t
--    = sum [fromInt (choose m k)*(s_n (n+k) a_ t)|k<-[0..m]]/2^m
```

```
--
-- e_err :: Int -> Int -> Double -> Double -> Double
-- e_err m n a_ t = abs ((e_mnt m (n+1) a_ t) - (e_mnt m n a_ t))


eulerInv :: LaplaceT -> Domain -> Matrix Double
eulerInv f_z ts = (ts:xs:[es]) where
    (xs,es) = unzip [filt f_z m n a_ t | t <- ts]
    m = 11; n = 45; a_ = 9.5
    -- was: a_=18.2 but 9.5 seems to work better

filt :: LaplaceT -> Int -> Int -> Double -> Double
        -> (Double, Double)
filt f_z m n a_ t = (g e_1, g $ abs (e_2-e_1)) where
    ak = ((0.5*(realPart $ f_z (0.5*a_/t :+ 0))
        :[(-1)^k * (a_k f_z k a_ t)| k <- [1..n+m+1]])
    sn = drop n $ scanl1 (+) ak
    (e_1, e_2) = (f (take (m+1) sn), f (drop 1 sn))
    u_ = exp(0.5*a_)/t
    c_ = [fromInt $ choose m k | k <- [0..m]]
    -- local funclets
    f :: [Double] -> Double
    f xs = sum (zipWith (*) c_ xs)
    g :: Double -> Double
    g = ((u_/2^m)*)

a_k :: LaplaceT -> Int -> Double -> Double -> Double
a_k f_z k a_ t = realPart $ f_z (0.5*a_/t :+ (fromInt k)*pi/t)


-- POST-WIDDER algorithm

postWidderInv :: LaplaceT -> Domain -> Matrix Double
postWidderInv f_z ts = (ts:[xs]) where
    xs = [f_jm f_z wks j m e_ t | t <- ts]
    j = 40; m = 4; e_ = 2
    wks = [w_km k m | k <- [1..m]]

f_jm :: LaplaceT -> [Double] -> Int -> Int -> Int
    -> Double -> Double
```

```
f_jm f_z wks j m e_ t = (sum $ zipWith (*) wks fns)/(2*t) where
    fns = [f_n f_z (j*k) e_ t | k <- [1..m]]

f_n :: LaplaceT -> Int -> Int -> Double -> Double
f_n f_z n e_ t = u_ * (f1 + f2 + 2*sm) where
    u_ = n1'/(n' * r^n)
    r = 10**(-(fromInt e_)/(2*n')
        -- why "2*n'": not explained in paper
    f1 = realPart $ f_z (n1'*(1-r)/t :+ 0)
    f2 = (-1)^n * (realPart $ f_z (n1'*(1+r)/t :+ 0))
    sm = sum [t_k k | k <- [1..n-1]]
    n' = fromInt n
    n1' = fromInt $ n + 1
    v_ = fromDouble $ n1'/t
    h_ = pi / n'
    t_k :: Int -> Double
    t_k k = (-1)^k * (realPart $ f_z (v_*(1-z))) where
        z = (fromDouble r) * exp (0 :+ h_*(fromInt k))

w_km :: Int -> Int -> Double
w_km k m = (fromInt $ (-1)^(m-k) * (k^(m-1)))
        / (fromInt $ (factorial $ k-1) * (factorial $ m-k))
```

B.2 Matrix.hs

```
module Matrix (
    Matrix, showMatrix, joinMatrices, showMatrixFixedCol,
    showTableFixedCol
    ) where

-- Matrix v0.02 (HS98); (c)001101; jtb.
--(c) 2001 Jeremy Bradley, University of Durham

import Utils (ColJustify(..),ColWidth,showTable,(?))
import List (transpose)


type Matrix a = [[a]]
```

```
-- Turns A and B into a single matrix (A|B)
joinMatrices :: Matrix a -> Matrix a -> Matrix a
joinMatrices xss yss = map (\(xs,ys) -> xs ++ ys) (zip xss yss)

-- Completely automated but probably slow
showMatrix :: Show a => Matrix a -> String
showMatrix xss = showTable cs ss where
    ns = map maximum $ map (map (length.show)) (transpose xss)
    cs = [((0,""),1,L)]
        ++ [((1,""),n,C) | n <- ns]
        ++ [((0,""),2,R)]
    ss = map (\s -> ["("] ++ s ++ [")"]) $ map (map show) xss


-- If you know the width of your elements best to use this one
showMatrixFixedCol :: Show a => Int -> Matrix a -> String
showMatrixFixedCol n xss = showTable cs ss where
    cs = [((0,""),1,L)]
        ++ (take (maximum $ map length xss)
            $ repeat ((1,""),n,R))
        ++ [((0,""),2,R)]
    ss = map (\s -> ["("] ++ s ++ [")"]) $ map (map show) xss

-- If you know the width of your elements best to use this one
showTableFixedCol :: Show a => Int -> Matrix a -> String
showTableFixedCol n xss = showTable cs ss where
    cs = take (maximum $ map length xss) $ repeat ((1,""),n,L)
    ss = map (map show) xss
```

B.3 Utils.hs

```
module Utils (
    selectTerm, replaceTerm, isNothing, nSpace, nS,(?),
    showCol, showTable, ColWidth, ColJustify(..), choose,
    factorial
    ) where

-- Utils v0.72 (HS98); (c)001101; jtb.
--(c) 2001 Jeremy Bradley, University of Durham
```

```
import Maybe (isJust)

-- selects the first item from a list (if it exists)
-- satisfying f and presents the item and the list\item
selectTerm :: (a -> Bool) -> [a] -> Maybe (a, [a])
selectTerm f ys
    | null ys2 = Nothing
    | otherwise = Just (head ys2, ys1 ++ tail ys2) where
        (ys1, ys2) = span (not.f) ys

-- replaces elems in list xs that pass f with y
replaceTerm :: (a -> Bool) -> a -> [a] -> [a]
replaceTerm f y xs = map (\x -> f x ? (y,x)) xs

isNothing :: Maybe a -> Bool
isNothing = not.isJust

(?) :: Bool -> (a, a) -> a
(?) bl e = if bl then (fst e) else (snd e)

-- String Related


nS = nSpace --alias
nSpace :: Int -> String
nSpace n
    | n < 0     = "" -- error "Utils.nSpace: negative argument"
    | otherwise = concat $ take n $ repeat " "



-- Maths funclets

factorial :: Int -> Int
factorial 0 = 1
factorial 1 = 1
factorial 2 = 2
factorial 3 = 6
factorial 4 = 24
factorial n = n * (factorial (n-1))
```

```
choose  :: Int -> Int -> Int
choose n r = div (factorial n) ((factorial r)*(factorial (n-r)))


-- Column display routines

data ColJustify = L | C | R
type ColWidth = Int

-- If string s exceeds column width then it is
--    allowed to overflow
showCol :: ColWidth -> ColJustify -> String -> String
showCol w j s
    | length s > w = case j of
        R -> (w > 0) ? ("$" ++ (drop (1 + (length s) - w) s), "")
        _ -> (w > 0) ? ((take (w - 1) s) ++ "$", "")
    | otherwise = case j of
        L -> s ++ (nS $ w - length s)
        C -> sp1 ++ s ++ sp2 where
            sp = w - length s
            sp1 = nS (div sp 2)
            sp2 = (mod sp 2 == 1) ? (sp1 ++ " ", sp1)
        R -> (nS $ w - length s) ++ s

-- 1st cw param indicates separation between cols,
--   2nd is width of column itself
showTable :: [((ColWidth, String), ColWidth, ColJustify)]
    -> [[String]] -> String
showTable _ [] = ""
showTable forms (xs:xss) = (showRow $ zip forms xs)
        ++ (showTable forms xss) where
    showRow::[(((ColWidth, String),ColWidth,ColJustify),String)]
        -> String
    showRow [] = "\n"
    showRow ((((ws, cdv), wc, just), str):cs)
        = (showCol ws C cdv) ++ (showCol wc just str)
            ++ (showRow cs)
```