hankel Documentation

Release 0.3.8

Steven Murray

Apr 02, 2019

Contents

Perform simple and accurate Hankel transformations using the method of Ogata 2005.

Hankel transforms and integrals are commonplace in any area in which Fourier Transforms are required over fields that are radially symmetric (see Wikipedia for a thorough description). They involve integrating an arbitrary function multiplied by a Bessel function of arbitrary order (of the first kind). Typical integration schemes often fall over because of the highly oscillatory nature of the transform. Ogata's quadrature method used in this package provides a fast and accurate way of performing the integration based on locating the zeros of the Bessel function.

CHAPTER 1

Features

- Accurate and fast solutions to many Hankel integrals
- Easy to use and re-use
- Arbitrary order transforms
- Built-in support for radially symmetric Fourier Transforms
- Thoroughly tested.
- Python 2 and 3 compatible.

CHAPTER 2

Quicklinks

- **Documentation:** https://hankel.readthedocs.io
- **Quickstart+Description:** Getting Started

CHAPTER 3

Installation

Either clone the repository at github.com/steven-murray/hankel and use `python setup.py install`, or simply install using `pip install hankel`.

The only dependencies are numpy, scipy and mpmath.

CHAPTER 4

References

Based on the algorithm provided in

H. Ogata, A Numerical Integration Formula Based on the Bessel Functions, Publications of the Research Institute for Mathematical Sciences, vol. 41, no. 4, pp. 949-970, 2005.

Also draws inspiration from

Fast Edge-corrected Measurement of the Two-Point Correlation Function and the Power Spectrum Szapudi, Istvan; Pan, Jun; Prunet, Simon; Budavari, Tamas (2005) The Astrophysical Journal vol. 631 (1)

CHAPTER 5

Contents

5.1 Examples

To help get you started using `hankel`, we've compiled a few extended examples. Other simple examples can be found in the API documentation for each object, or by looking at some of the tests.

5.1.1 Getting Started with Hankel

Usage and Description

Setup

This implementation is set up to allow efficient calculation of multiple functions $f(x)$. To do this, the format is class-based, with the main object taking as arguments the order of the Bessel function, and the number and size of the integration steps (see *Limitations* for discussion about how to choose these key parameters).

For any general integration or transform of a function, we perform the following setup:

[1]:
```
import hankel
from hankel import HankelTransform     # Import the basic class
print("Using hankel v{}".format(hankel.__version__))
```

```
Using hankel v0.3.7
```

[2]:
```
ht = HankelTransform(
    nu= 0,      # The order of the bessel function
    N = 120,    # Number of steps in the integration
    h = 0.03    # Proxy for "size" of steps in integration
)
```

Alternatively, each of the parameters has defaults, so you needn't pass any. The order of the bessel function will be defined by the problem at hand, while the other arguments typically require some exploration to set them optimally.

Integration

A Hankel-type integral is the integral

$$\int_0^\infty f(x)J_\nu(x)dx.$$

Having set up our transform with `nu = 0`, we may wish to perform this integral for $f(x) = 1$. To do this, we do the following:

[3]:
```
# Create a function which is identically 1.
f = lambda x : 1
ht.integrate(f)
```

[3]:
```
(1.0000000000003486, -9.838142836853752e-15)
```

The correct answer is 1, so we have done quite well. The second element of the returned result is an estimate of the error (it is the last term in the summation). The error estimate can be omitted using the argument `ret_err=False`.

We may now wish to integrate a different function, say $x/(x^2 + 1)$. We can do this directly with the same object, without re-instantiating (avoiding unnecessary recalculation):

[4]:
```
f = lambda x : x/(x**2 + 1)
ht.integrate(f)
```

[4]:
```
(0.42098875721567186, -2.6150757700135774e-17)
```

The analytic answer here is $K_0(1) = 0.4210$. The accuracy could be increased by creating `ht` with a higher number of steps `N`, and lower stepsize `h` (see *Limitations*).

Transforms

The Hankel transform is defined as

$$F_\nu(k) = \int_0^\infty f(r)J_\nu(kr)rdr.$$

We see that the Hankel-type integral is the Hankel transform of $f(r)/r$ with $k = 1$. To perform this more general transform, we must supply the k values. Again, let's use our previous function, $x/(x^2 + 1)$.

First we'll import some libraries to help us visualise:

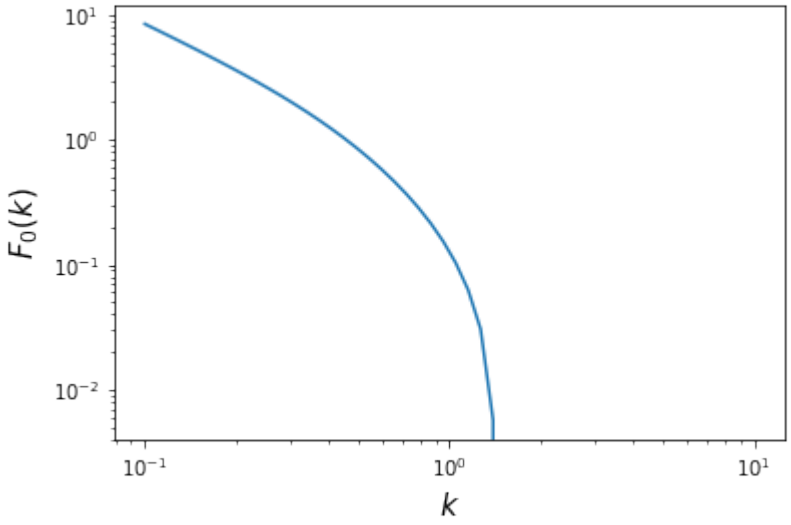
[5]:
```
import numpy as np               # Import numpy
import matplotlib.pyplot as plt
%matplotlib inline
```

Now do the transform,

[6]:
```
k = np.logspace(-1,1,50)             # Create a log-spaced array of k from 0.1 to 10.
Fk = ht.transform(f,k,ret_err=False) # Return the transform of f at k.
```

and finally plot it:

[8]:
```
plt.plot(k,Fk)
plt.xscale('log')
plt.yscale('log')
plt.ylabel(r"$F_0(k)$", fontsize=15)
plt.xlabel(r"$k$", fontsize=15)
plt.show()
```



Fourier Transforms

One of the most common applications of the Hankel transform is to solve the radially symmetric n-dimensional Fourier transform:

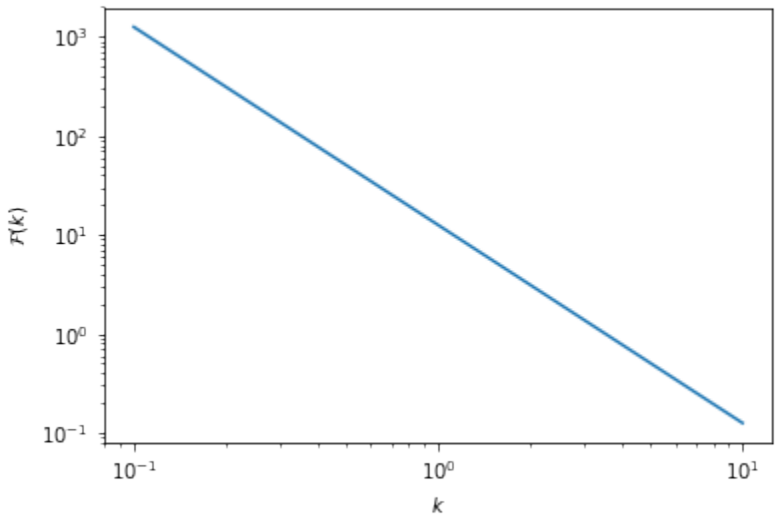
$$F(k) = \frac{(2\pi)^{n/2}}{k^{n/2-1}} \int_0^\infty r^{n/2-1} f(r) J_{n/2-1}(kr) r dr.$$

We provide a specific class to do this transform, which takes into account the various normalisations and substitutions required, and also provides the inverse transform. The procedure is similar to the basic `HankelTransform`, but we provide the number of dimensions, rather than the Bessel order directly.

Say we wish to find the Fourier transform of $f(r) = 1/r$ in 3 dimensions:

[9]:
```
# Import the Symmetric Fourier Transform class
from hankel import SymmetricFourierTransform

# Create our transform object, similar to HankelTransform,
# but with ndim specified instead of nu.
ft = SymmetricFourierTransform(ndim=3, N = 200, h = 0.03)

# Create our kernel function to be transformed.
f = lambda r : 1./r

# Perform the transform
Fk = ft.transform(f,k, ret_err=False)
```

[10]:
```
plt.plot(k,Fk)
plt.xscale('log')
plt.yscale('log')
plt.ylabel(r"$\mathcal{F}(k)$")
plt.xlabel(r"$k$")
plt.show()
```



To do the inverse transformation (which is different by a normalisation constant), merely supply `inverse=True` to the `.transform()` method.

Limitations

Efficiency

An implementation-specific limitation is that the method is not perfectly efficient in all cases. Care has been taken to make it efficient in the general sense. However, for specific orders and functions, simplifications may be made which reduce the number of trigonometric functions evaluated. For instance, for a zeroth-order spherical transform, the weights are analytically always identically 1.

Lower-Bound Convergence

Theoretically, since Bessel functions have asymptotic power-law slope of $n/2 - 1$ as $r \to 0$, the function $f(r)$ must have an asymptotic slope $> -n/2$ in this regime in order to converge. This restriction is method-independent.

In terms of limitations of the method, they are very dependent on the form of the function chosen. Notably, functions which either have sharp features at low r, or tend towards a slope of $\sim -n/2$, will be poorly approximated in this method, and will be highly dependent on the step-size parameter, as the information at low-x will be lost between 0 and the first step. As an example consider the simple function $f(x) = 1/\sqrt{x}$ with a 1/2 order bessel function. The total integrand tends to 1 at $x = 0$, rather than 0:

[11]:
```
f = lambda x: 1/np.sqrt(x)
h = HankelTransform(0.5,120,0.03)
h.integrate(f)
```

[11]:
```
(1.2336282257874065, -2.864861354876958e-16)
```

The true answer is $\sqrt{\pi/2}$, which is a difference of about 1.6%. Modifying the step size and number of steps to gain accuracy we find:

[12]:
```
h = HankelTransform(0.5,700,0.001)
h.integrate(f)
```

[12]:
```
(1.2523045155005623, -0.0012281146007915768)
```

This has much better than percent accuracy, but uses 5 times the amount of steps. The key here is the reduction of h to "get inside" the low-x information.

Upper-Bound Convergence

Theoretically, since the Bessel functions decay as $r^{-1/2}\cos r$ as $f \to \infty$, the asymptotic logarithmic slope of $f(r)$ must be $< (1-n)/2$ in order to converge.

As the asymptotic slope approaches this value, higher and higher zeros of the Bessel function will be required to capture the convergence. Often, it will be the case that if this is so, the amplitude of the function is low at low x, so that the step-size `h` can be increased to facilitate this. Otherwise, the number of steps `N` can be increased.

For example, the 1/2-order integral supports functions that are increasing up to $f(x) = x^{0.5}$. Let's use $f(x) = x^{0.4}$ as an example of a slowly converging function, and use our "hi-res" setup from the previous section. We note that the analytic result is 0.8421449.

[20]:
```
f = lambda x : x**0.4
```

[25]:
```
%%timeit
HankelTransform(0.5,700,0.001).integrate(f)
```

```
735 µs ± 8.11 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

[26]:
```
res = HankelTransform(0.5,700,0.001).integrate(f)
print("Relative Error is: ", res[0]/0.8421449 - 1)
print("Predicted Abs. Error is: ", res[1])
```

```
Relative Error is:  -0.362600451237186
Predicted Abs. Error is:  -1.05909546212511
```

Our result is way off. Note that in this case, the error estimate itself is a good indication that we haven't reached convergence. We could try increasing `N`:

[27]:
```
h = HankelTransform(0.5,10000,0.001)
print("Relative Error is: ", h.integrate(f,ret_err=False)/0.8421449 -1)
```

```
Relative Error is:  7.133537831549575e-07
```

This is very accurate, but quite slow:

[28]:
```
%%timeit
HankelTransform(0.5,10000,0.001).integrate(f)
```

```
9.12 ms ± 211 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Alternatively, we could try increasing `h`:

[18]:
```
h = HankelTransform(0.5,700,0.03)
h.integrate(f,ret_err=False)/0.8421449 -1
```

[18]:
```
0.00045613842025526985
```

Not quite as accurate, but still far better than a percent for a tenth of the cost:

[29]:
```
%%timeit
HankelTransform(0.5,700,0.03).integrate(f)
```

```
718 µs ± 2.04 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

We explore how to determine N and h in more detail in another demo, and `hankel` provides the `get_h` function to automatically determine these values (at some computational cost).

5.1.2 Testing Forward and Inverse Hankel Transform

This is a simple demo to show how to compute the forward and inverse Hankel transform, essentially returning to the original function.

This is a common application, where often the transformed function is modified before performing the inverse transform (eg. in the calculation of a Laplacian). However, it is not explicitly supported by `hankel`, since the transformed function itself is merely encoded as a set of values, rather than a callable function. This demo shows the simplest route to closing this gap, using splines. It also highlights some of the issues one may encounter along the way.

This application is also important, as it is able to self-consistently test the accuracy of the transformation for any arbitrary function $f(r)$. The returned function should be very close to this function for an accurate transform. This idea is explored further in the demo on choosing resolution parameters.

We use the function $f(r) = 1/(r^2 + 1)$ as an example function.

[1]:
```
# Import libraries

import numpy as np
import hankel
from hankel import HankelTransform                                    # Transforms

from scipy.interpolate import InterpolatedUnivariateSpline as spline  # Spline

import matplotlib.pyplot as plt                                       # Plotting
%matplotlib inline

print("Using hankel v{}".format(hankel.__version__))
```

```
Using hankel v0.3.7
```

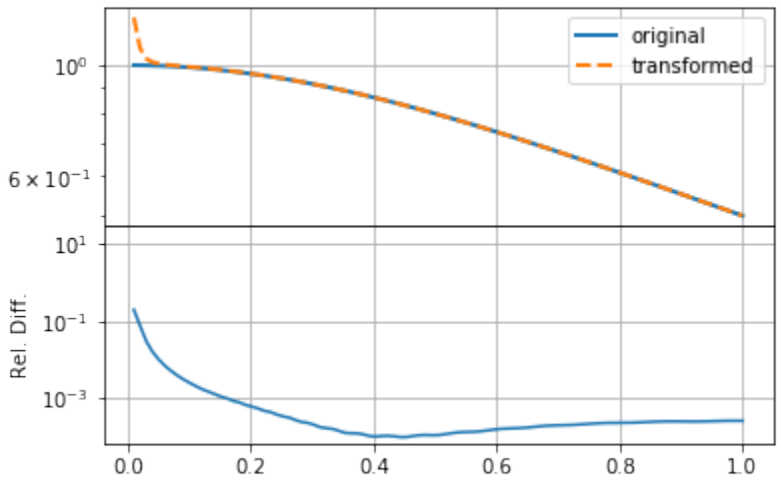
[53]:
```
# Define grid

r = np.linspace(0,1,100)[1:]       # Define a physical grid
k = np.logspace(-3,2,100)           # Define a spectral grid
```

[54]:
```
# Compute Forward Hankel transform

f    = lambda r : 1/(r**2 + 1)                    # Sample Function
h    = HankelTransform(nu=0,N=1000,h=0.005)       # Create the HankelTransform␣
↪instance, order zero
hhat = h.transform(f,k,ret_err=False)             # Return the transform of f at k.
```

[55]:
```
# Compute Inverse Hankel transform

hhat_sp = spline(k, hhat)                          # Define a spline to␣
↪approximate transform
f_new = h.transform(hhat_sp, r, False, inverse=True)  # Compute the inverse transform
```

[56]:
```
def mkplot(f_new):
    fig,ax = plt.subplots(2,1,sharex=True,gridspec_kw={"hspace":0})

    ax[0].semilogy(r,f(r), linewidth=2,label='original')
    ax[0].semilogy(r,f_new,ls='--',linewidth=2,label='transformed')
    ax[0].grid(True)
    ax[0].legend(loc='best')

    ax[1].plot(r,np.abs(f(r)/f_new-1))
    ax[1].set_yscale('log')
    ax[1].set_ylim(None,30)
    ax[1].grid(True)
    ax[1].set_ylabel("Rel. Diff.")
    plt.show()
```

[57]:
```
# Plot the original function and the transformed functions
mkplot(f_new)
```



The result here is reasonable, though not very accurate, especially at small r.

In practice, there are three aspects that affect the accuracy of the round-trip transformed function, other than the features of the function itself:

1. the value of `N`, which controls the the upper limit of the integral (and must be high enough for convergence),
2. the value of `h`, which controls the resolution of the array used to do integration. Most importantly, controls the position of the *first sample* of the integrand. In a function such as $1/r$, or something steeper, this must be small to capture the large amount of information at low r.
3. the resolution/range of k, which is used to define the function which is inverse-transformed.

As a simple empirical exercise, we modify each of these three below to estimate their importance in this particular transform (see the demo on choosing resolution parameters for a more thorough investigation):

Changing N

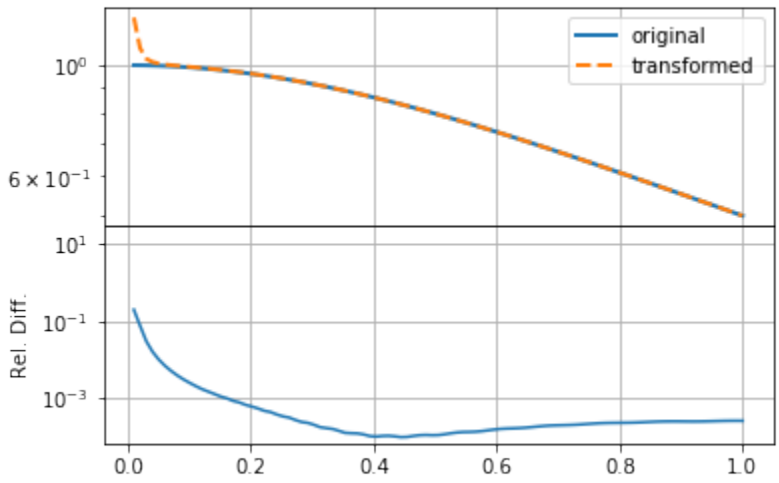
[58]:
```
h    = HankelTransform(nu=0,N=5000,h=0.005)        # Create the HankelTransform␣
↪instance, order zero
hhat = h.transform(f,k,ret_err=False)              # Return the transform of f at k.
```

(continues on next page)

(continued from previous page)

```
hhat_sp = spline(k, hhat)                                  # Define a spline to␣
↪approximate transform
f_new = h.transform(hhat_sp, r, False, inverse=True)   # Compute the inverse transform
```

[60]:
```
mkplot(f_new)
```

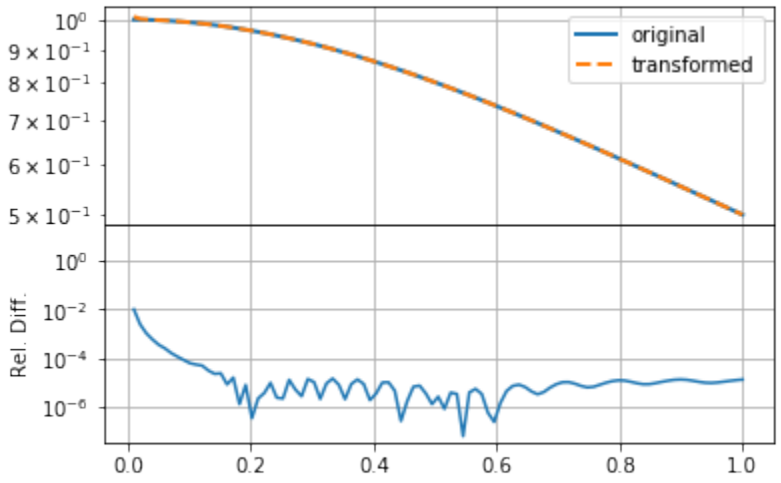


Increasing N in this case does not seem to make the solution any better.

Changing h

[61]:
```
h     = HankelTransform(nu=0,N=5000,h=0.001)         # Create the HankelTransform␣
↪instance, order zero
hhat = h.transform(f,k,ret_err=False)                # Return the transform of f at k.

hhat_sp = spline(k, hhat)                                  # Define a spline to␣
↪approximate transform
f_new = h.transform(hhat_sp, r, False, inverse=True)   # Compute the inverse transform
```

[62]:
```
mkplot(f_new)
```

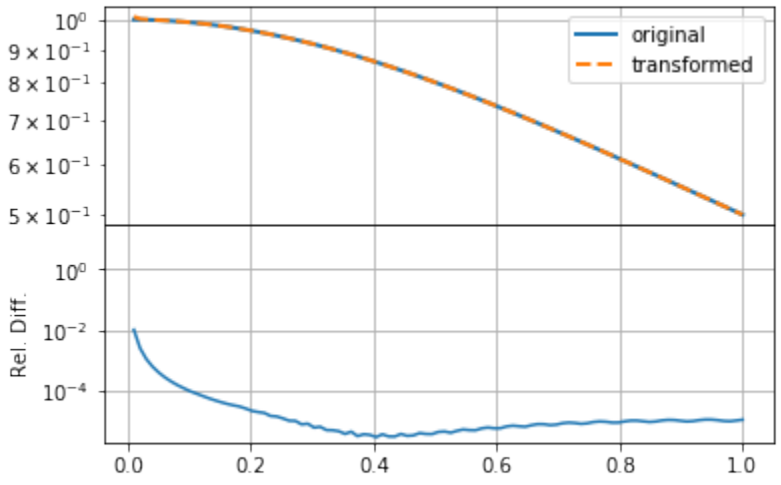


Clearly changing h makes much more of a difference, reducing the relative error by an order of magnitude. In this case, we ensured that the product hN remained the same (as the original example), so that the upper bound on the summation remained approximately the same. This indicates that either $f(r)$ or $F(k)$ has high-resolution information.

Changing k

[63]:
```
k = np.logspace(-3,3,200)                                  # Define a spectral grid

h    = HankelTransform(nu=0,N=5000,h=0.001)                # Create the HankelTransform␣
↪instance, order zero
hhat = h.transform(f,k,ret_err=False)                      # Return the transform of f at k.

hhat_sp = spline(k, hhat)                                      # Define a spline to␣
↪approximate transform
f_new = h.transform(hhat_sp, r, False, inverse=True)   # Compute the inverse transform
```

[64]:
```
mkplot(f_new)
```



Having set k to extend to smaller scales, as well as increasing its resolution, has not helped the accuracy of the transformation, though it does seemed to have smoothed out small-scale wiggles in the transformation. Thus we

can see that the predominant problem in this case was that the integration grid was not fine enough to capture the small-scale information in the function (either $f(r)$ or $F(k)$).

Any procedure like this will have to explore the values of these three parameters in order to set them as efficiently and accurately as possible.

5.1.3 Choosing Resolution Parameters

The only real choices to be made when using `hankel` are the choice of resolution parameters N and h. Roughly speaking, h controls the quadrature bin width, while N controls the number of these bins, ideally simulating infinity. Here we identify some rules of thumb for choosing these parameters so that desired precision can be attained.

For ease of reference, we state our problem explicitly. We'll deal first with the simple Hankel integral, moving onto a transformation, and Symmetric FT in later sections. For an input function $f(x)$, and transform of order ν, we are required to solve the Hankel integral

$$\int_0^\infty f(x) J_\nu(x) dr. \tag{5.1}$$

The O5 method approximates the integral as

$$\hat{f}(K) = \pi \sum_{k=1}^{N} w_{\nu k} f\left(y_{\nu k}\right) J_\nu(y_{\nu k}) \psi'(h r_{\nu k}), \tag{5.2}$$

where

$$y_{\nu k} = \pi \psi(h r_{\nu k})/h \tag{5.3}$$

$$\psi(t) = t \tanh(\pi \sinh(t)/2) \tag{5.4}$$

$$\psi'(t) = \frac{\pi t \cosh(t) + \sinh(\pi \sinh(t))}{1 + \cosh(\pi \sinh(t))} \tag{5.5}$$

$$w_{\nu k} = \frac{Y_\nu(\pi r_{\nu k})}{J_{\nu+1}(\pi r_{\nu k})}. \tag{5.6}$$

Here $Y_\nu(x)$ is a Bessel function of the second kind, and $r_{\nu k}$ are the roots of $J_\nu(\pi r)$.

Simple Hankel Integral

Choosing N given h

Choosing a good value of N given h is a reasonably simple task. The benefit of the O5 method is that the successive nodes approach the roots of the Bessel function double-exponentially. This means that at some term k in the series, the Bessel function term in the sum approaches zero, and for reasonably low k.

This is because for large t, $\psi(t) \approx t$, so that $y_{\nu k} \approx \pi r_{\nu k}$, which are the roots ($r$ are the roots scaled by π). Thus we can expect that a plot of the values of $J_\nu(y_{\nu k})$ should fall to zero, and they should do this approximately identically as a function of $h r_{\nu k}$.

[1]:
```
from scipy.special import yv,jv
from scipy.integrate import simps
from mpmath import fp as mpm
import numpy as np

import matplotlib.pyplot as plt
import matplotlib as mpl
%matplotlib inline
```

(continues on next page)

(continued from previous page)

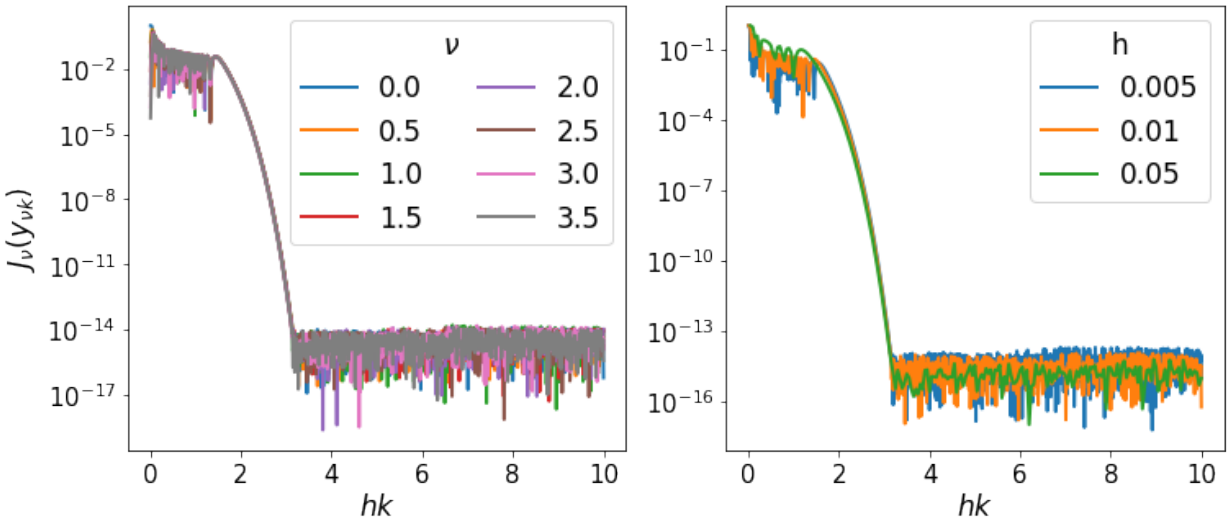
```
%load_ext autoreload
%autoreload 2

from hankel import HankelTransform, SymmetricFourierTransform
import hankel
print("Using hankel v{}".format(hankel.__version__))
```

```
Using hankel v0.3.7
```

[2]:
```
mpl.rcParams['lines.linewidth'] = 2
mpl.rcParams['xtick.labelsize'] = 15
mpl.rcParams['ytick.labelsize'] = 15
mpl.rcParams['font.size'] = 17
mpl.rcParams['axes.titlesize'] = 14
```

We test our assertion by plotting these values for a range of ν and h:

[3]:
```
fig,ax = plt.subplots(1,2,figsize=(12,5), subplot_kw={"yscale":'log'})
for nu in np.arange(0,4,0.5):
    ht= HankelTransform(nu=nu,N=1000, h = 0.01)
    ax[0].plot(ht._h*np.arange(1,1001), np.abs(jv(ht._nu, ht.x)), label=str(nu))

for h in [0.005,0.01,0.05]:
    ht= HankelTransform(nu=0,N=10/h, h = h)
    ax[1].plot(ht._h*np.arange(1,10/h+1), np.abs(jv(ht._nu, ht.x)), label=str(h))

ax[0].legend(ncol=2, title=r"$\nu$")
ax[1].legend(title='h')
ax[0].set_ylabel(r"$J_\nu(y_{\nu k})$")
ax[0].set_xlabel(r"$hk$")
ax[1].set_xlabel(r"$hk$");
```



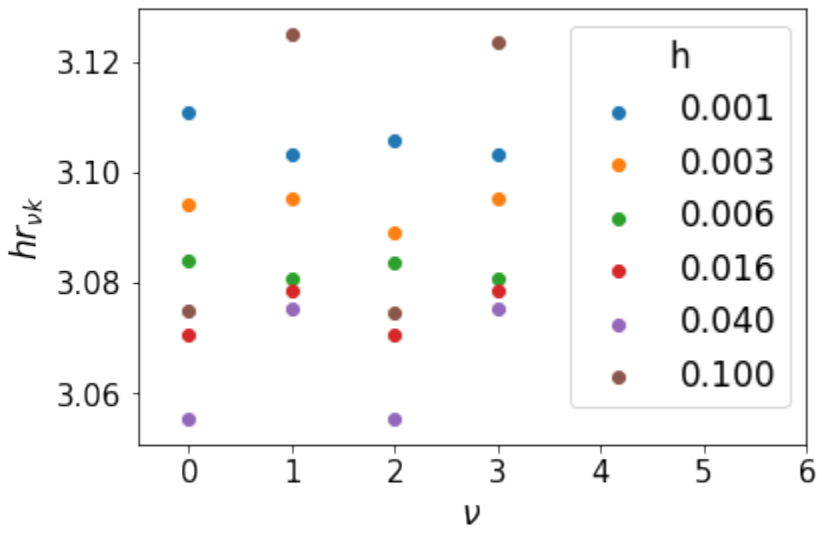
Interestingly, the fall-off is very similar across a range of both ν and h. We can compute where approximately the fall-off is completed:

[4]:
```
for i,nu in enumerate( np.arange(0,4)):
```

(continues on next page)

(continued from previous page)

```
    for j,h in enumerate(np.logspace(-3,-1,6)):
        ht= HankelTransform(nu=nu,N=int(3.2/h), h = h)
        plt.scatter(nu,ht._h*ht._zeros[np.where(np.abs(jv(ht._nu, ht.x))<1e-
↪13)[0][0]],color="C%s"%j,label="%.3f"%h if not i else None)

plt.xlabel(r"$\nu$")
plt.ylabel(r"$hr_{\nu k}$")
plt.xlim(-0.5,6)
plt.legend(title="h");
```



Clearly, we can cut the summation at $hr_{\nu k} = \pi$ without losing any precision. We do not want to sum further than this for two reasons: firstly, it is inefficient to do so, and secondly, we could be adding unnecessary numerical noise.

Now, let's assume that N is reasonably large, so that the Bessel function is close to its asymptotic limit, in which

$$r_{\nu k} = k - \frac{\pi\nu}{2} - \frac{\pi}{4} \approx k. \tag{5.7}$$

Then we merely set $hr_{\nu k} = hN = 3.2$, i.e. $N = \pi/h$.

It may be a reasonable question to ask whether we could set N significantly lower than this limit. The function $f(x)$ may converge faster than the Bessel function itself, in which case the limit could be reduced. In this regard it is useful to keep in mind that the sum extends to $r \sim \pi N$, if N is reasonably large. If $f(x)$ falls to zero for $x \ll \pi^2/h$, then it is probably reasonable to use a lower value of N.

However, for simplicity, for the rest of our analysis, we consider N to be set by this relation, and change h to modify N.

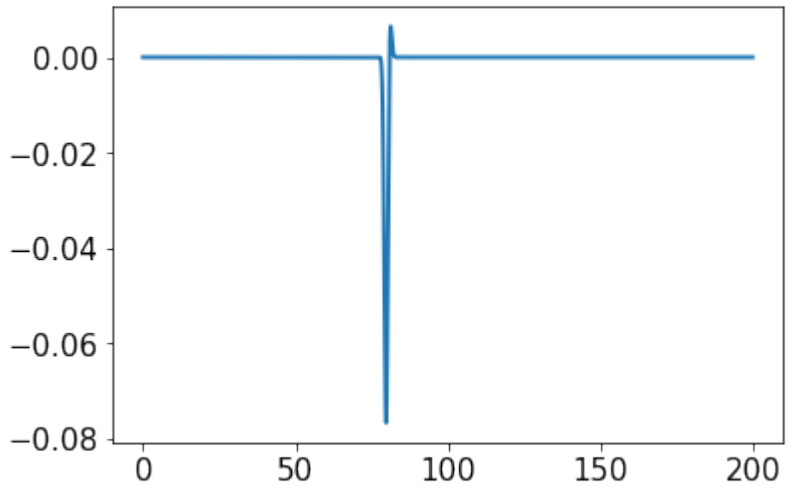
Choosing h

O5 give a rather involved proof of an upper limit on the residual error of their method as a function of h. Unfortunately, evaluating the upper limit is non-trivial, and we pursue a more tractable approach here, namely iteratively modifying h until convergence is reached.

As an example, let's take a sharp Gaussian, $f(x) = e^{-(x-80)^2}$ with $\nu = 0$:

[5]:
```
x = np.linspace(0,200.,1000000)
ff = lambda x : np.exp(-(x-80.)**2/1.)
plt.plot(x,ff(x) * jv(0,x))

res80 = simps(ff(x) * jv(0,x),x)
print("Integral is: ", res80)
```

```
Integral is:  -0.09651170657186205
```



[6]:
```
print("h\tN\txmax\t#nodes in peak\t Rel. Err.")
print("-----------------------------------------------")

for h in np.logspace(-4,0,10):
    N = int(np.pi/h)
    ht = HankelTransform(nu=0, h=h, N=N)
    G = ht.G(ff,h)
    ans,cum  = ht.integrate(f= ff,ret_cumsum=True,ret_err=False)
    print(f"{h:.2e}\t{N}\t{np.pi*N:.2e}\t{np.sum(np.logical_and(ht.x>78,ht.x<82))}\t
↪{ans/res80 - 1:.2e}")
```

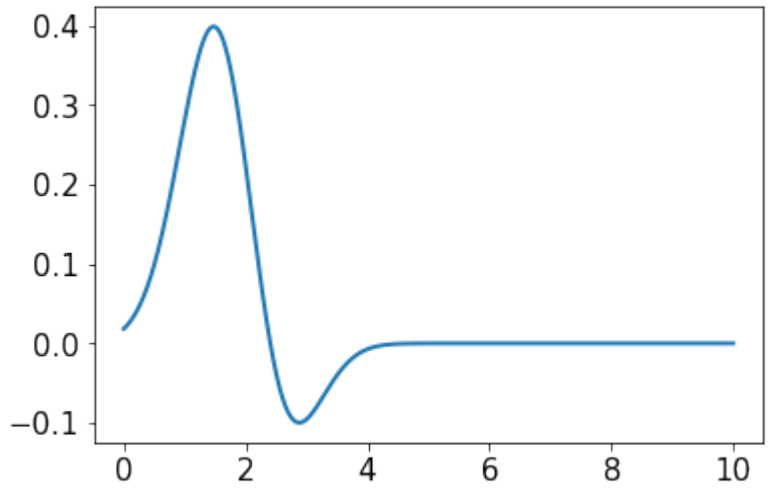
```
h       N       xmax    #nodes in peak  Rel. Err.
------------------------------
1.00e-04        31415   9.87e+04        10      -1.01e-14
2.78e-04        11290   3.55e+04        6       2.10e-08
7.74e-04        4057    1.27e+04        4       -5.55e-03
2.15e-03        1458    4.58e+03        3       2.76e-01
5.99e-03        524     1.65e+03        1       1.00e+00
1.67e-02        188     5.91e+02        1       -1.25e+00
4.64e-02        67      2.10e+02        1       1.38e+00
1.29e-01        24      7.54e+01        0       -1.00e+00
3.59e-01        8       2.51e+01        0       -1.00e+00
1.00e+00        3       9.42e+00        0       -1.00e+00
```

In the above example we see that only for very large values of h was the criteria of negative derivative not met. However, the criteria of covering the interesting region with nodes was only met by the smallest 5 values of h, each of which yields a good value of the integral.

Doing the same example, but moving the Gaussian closer to zero yields a different answer:

[7]:
```
x = np.linspace(0,10,10000)
ff = lambda x : np.exp(-(x-2)**2)
plt.plot(x,ff(x) * jv(0,x))

res = simps(ff(x) * jv(0,x),x)
print("Integral is: ", res)
```

```
Integral is:  0.4168433779916697
```



[8]:
```
print("h\tN\txmax\t#nodes in peak\t Rel. Err.")
print("-----------------------------------------------")

for h in np.logspace(-4,0,10):
    N = int(np.pi/h)
    ht = HankelTransform(nu=0, h=h, N=N)
    G = ht.G(ff,h)
    ans,cum  = ht.integrate(f= ff,ret_cumsum=True,ret_err=False)
    print(f"{h:.2e}\t{N}\t{np.pi*N:.2e}\t{np.sum(np.logical_and(ht.x>78,ht.x<82))}\t
↪{ans/res - 1:.2e}")
```

```
h       N       xmax    #nodes in peak  Rel. Err.
------------------------------
1.00e-04        31415   9.87e+04        10      -2.47e-11
2.78e-04        11290   3.55e+04        6       -2.47e-11
7.74e-04        4057    1.27e+04        4       -2.47e-11
2.15e-03        1458    4.58e+03        3       -2.47e-11
5.99e-03        524     1.65e+03        1       2.45e-08
1.67e-02        188     5.91e+02        1       -1.09e-03
4.64e-02        67      2.10e+02        1       7.81e-02
1.29e-01        24      7.54e+01        0       5.37e-01
3.59e-01        8       2.51e+01        0       5.97e-01
1.00e+00        3       9.42e+00        0       6.09e-01
```

Here we are able to achieve good precision with just ~500 terms.

These ideas are built into the `get_h` function in `hankel`. In particular, this function progressively iterates through a series of values for h, and stops when two consecutive results are within a certain tolerance of each other. On each iteration, it uses $N = \pi/h$.

While this function is not entirely general – there are cases in which the algorithm will return prematurely – it should

do well in cases where the integrand is reasonably smooth. In addition to the convergence of the result, it also checks that the derivative of G is decreasing. Furthermore, it truncates N if possible such that $f(x_k) = 0 \ \forall k > N$.

An example:

[9]:
```
best_h, result, best_N = hankel.get_h(
    f =lambda x : np.exp(-(x-2)**2),
    nu=0
)
print(f"best_h = {best_h}, best_N={best_N}")
print("Relative Error: ", result/res - 1)
```

```
best_h = 0.0125, best_N=22
Relative Error:  -0.00011951202608917466
```

Here we can see that while a low value of h was required, a corresponding high value of N was not, as the function itself converges quickly.

We can also repeat our previous experiment where a sharp Gaussian sits at high x:

[10]:
```
best_h, result, best_N = hankel.get_h(
    f =lambda x : np.exp(-(x-80.)**2),
    nu =0
)

print(f"best_h = {best_h}, best_N={best_N}")
print("Relative Error: ", result/res80 - 1)
```

```
best_h = 0.00078125, best_N=168
Relative Error:  0.002185348402481635
```

Here both a lower h and higher N are required, but again, N does not need to be as large as π/h.

We can modify the required tolerance:

[11]:
```
best_h, result, best_N = hankel.get_h(
    f =lambda x : np.exp(-(x-2)**2),
    nu=0,
    atol=1e-10,
    rtol=1e-15
)
print(f"best_h = {best_h}, best_N={best_N}")
print("Relative Error: ", result/res - 1)
```

```
best_h = 0.003125, best_N=44
Relative Error:  -2.470723625691562e-11
```

Remember that the tolerances submitted to the `get_h` function apply to the similarity between successive iterations. The relative error displayed here may be dominated by numerical error on the simpson's integration.

Finally we try a different kind of function, namely one that converges slowly:

[12]:
```
best_h, result, best_N = hankel.get_h(
    f =lambda x : x**0.4,
    nu=0.5,
    atol=1e-5,
    rtol=1e-5
)
print(f"best_h = {best_h}, best_N={best_N} (max is {int(np.pi/best_h)})")
print("Relative Error: ", result/0.8421449 - 1)
```

```
best_h = 0.00625, best_N=502 (max is 502)
Relative Error:  2.317849153432583e-05
```

In this case, the required N is the maximum required, as we need to integrate quite far in order to converge.

Symmetric Fourier Transform

In the context of the symmetric Fourier Transform, much of the work is exactly the same – the relationship between N and h necessarily remains. The only differences are that we now use the `SymmetricFourierTransform` class instead of the `HankelTransform` (and pass `ndim` rather than `nu`), and that we are now interested in the *transform*, rather than the integral, so we have a particular set of scales, K, in mind.

For a given K, the minimum and maximum values of x that we evaluate $f(x)$ for are $x_{\min} \approx \pi^2 h r_{\nu 1}^2/2K$ and $x_{\max} \approx \pi N/K$. We suggest find a value h that works for both the minimum and maximum K desired. All scales in between should work in this case.

We have already written this functionality into the `get_h` function. However, here `nu` is interpreted directly as `n`, and we need to pass a value for K.

[13]:
```
hankel.get_h(
    f = lambda x : np.exp(-x**2), nu=2,
    K= np.array([0.01, 100]),
    cls=SymmetricFourierTransform
)
```

[13]:
```
(0.05, array([8.25274593e-88, 1.38996118e-09]), 62)
```

5.1.4 Computing Laplacian Transforms with Hankel vs FFT

This will compare the forward and inverse transforms for both Hankel and Fourier by either computing partial derivatives of solving a parital differential equation.

This notebook focuses on the Laplacian operator in the case of radial symmetry.

Consider two 2D circularly-symmetric functions $f(r)$ and $g(r)$ that are related by the following differential operator,

$$g(r) = \nabla^2 f(r) = \frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial f}{\partial r}\right)$$

In this notebook we will consider two problems: 1. Given $f(r)$, compute the Laplacian to obtain $g(r)$ 2. Given $g(r)$, invert the Laplacian to obtain $f(r)$

We can use the 1D Hankel (or 2D Fourier) transform to compute the Laplacian in three steps:

1. Compute the Forward Transform

$$\mathcal{H}[f(r)] = \hat{f}(k) \tag{5.8}$$

2. Differentiate in Spectral space

$$\hat{g}(k) = -k^2 \hat{f}(k) \tag{5.9}$$

3. Compute the Inverse Transform

$$g(r) = \mathcal{H}^{-1}[\hat{g}(k)] \tag{5.10}$$

This is easily done in two-dimensions using the Fast Fourier Transform (FFT) but one advantage of the Hankel transform is that we only have a one-dimensional transform.

In particular, we will use the function

$$f(r) = e^{-r^2},$$

and, though it is arbitrary, we will use the Fourier convention that $(a, b) = (1, 1)$ (i.e. $F(k) = \int dr\, f(r) e^{-ik\cdot r}$). This gives the corollary functions

$$\hat{f}(k) = \pi e^{-k^2/4} \tag{5.11}$$

$$\hat{g}(k) = -k^2 \pi e^{-k^2/4} \tag{5.12}$$

$$g(r) = 4e^{-r^2}(r^2 - 1) \tag{5.13}$$

$$\tag{5.14}$$

Setup

Import Relevant Libraries

[3]:
```
# Import Libraries

import numpy as np                                                    # Numpy
from powerbox import dft                                              # powerbox␣
↪for DFTs (v0.6.0+)
from hankel import SymmetricFourierTransform                          # Hankel
from scipy.interpolate import InterpolatedUnivariateSpline as spline  # Splines
import matplotlib.pyplot as plt                                       # Plotting
%matplotlib inline

import hankel
print("Using hankel v{}".format(hankel.__version__))
```

```
Using hankel v0.3.8
```

Define Sample Functions

[4]:
```
f = lambda r: np.exp(-r**2)
g = lambda r: 4.0*np.exp(-r**2)*(r**2 - 1.0)
fhat = lambda k : np.pi * np.exp(-k**2/4)
ghat = lambda k : -k**2 * np.pi * np.exp(-k**2/4)
```

Define Transformation Functions

[15]:
```
def ft2d(X, L, inverse=False):
    """Convenience function for getting nD Fourier Transform and a 1D function of it""
↪"
    if inverse:
        ft, _, xgrid = dft.ifft(X, L=L, a=1, b=1, ret_cubegrid=True)
    else:
        ft, _, xgrid = dft.fft(X, L=L, a=1, b=1, ret_cubegrid=True)

    # Get a sorted array of the transform
    ind = np.argsort(xgrid.flatten())
```

(continues on next page)

(continued from previous page)

```
    _xgrid = xgrid.flatten()[ind]
    _ft = ft.flatten()[ind]

    # Just take unique values.
    _xgrid, ind = np.unique(_xgrid, return_index=True)
    _ft = _ft[ind]

    return ft, xgrid, _xgrid, _ft.real # we only deal with real functions here.
```

Grid Setup

[4]:
```
L     = 10.0    # Size of box for transform
N     = 128     # Grid size
b0    = 1.0
```

[77]:
```
# Create the Hankel Transform object

ht = SymmetricFourierTransform(ndim=2, h=0.005)
```

[20]:
```
dr = L/N

# Create persistent dictionaries to track Hankel and Fourier results throughout.
H = {} # Hankel dict
F = {} # Fourier dict

r = np.linspace(dr/2, L-dr/2, N)

# 1D Grid for Hankel
H['r'] = r

# To decide on k values to use, we need to know the scales
# we'll require to evaluate on the *inverse* transform:
H['k'] = np.logspace(-0.5, 2.1, 2*N)       # k-space is rather arbitrary.

# 2D Grid for Fourier
F['x'] = np.arange(-N, N)*dr
F['rgrid'] = np.sqrt(np.add.outer(F['x']**2, F['x']**2))
F['fgrid'] = f(F['rgrid'])
```

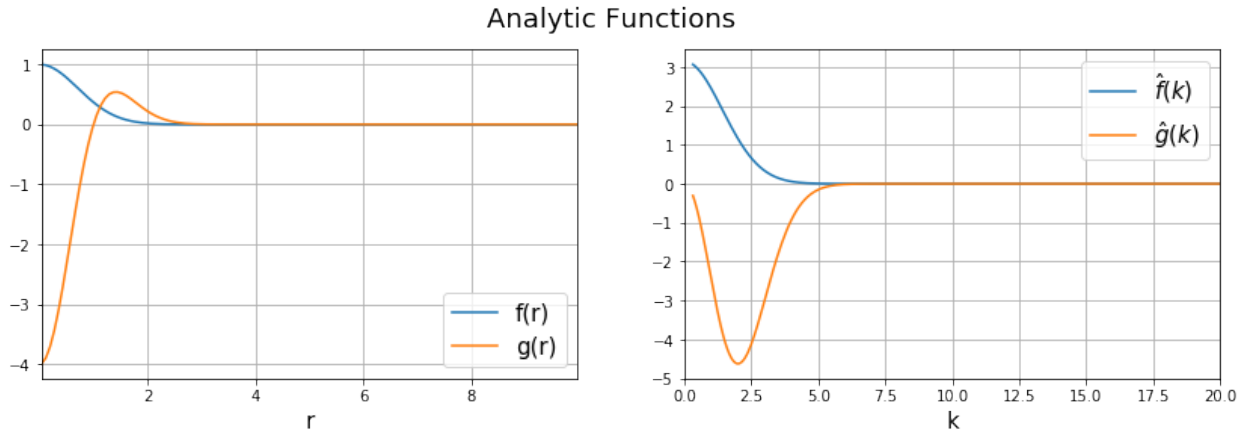
Analytic Result

[21]:
```
fig, ax = plt.subplots(1,2, gridspec_kw={"hspace":0.05}, figsize=(14, 4))

ax[0].plot(r, f(r), label="f(r)")
ax[0].plot(r, g(r), label='g(r)')
ax[0].set_xlim(r.min(), r.max())
plt.suptitle('Analytic Functions', fontsize=18)
ax[0].legend(loc='best', fontsize=15);
ax[0].set_xlabel("r", fontsize=15)

ax[1].plot(H['k'], fhat(H['k']), label="$\hat{f}(k)$")
```

(continues on next page)

(continued from previous page)

```
ax[1].plot(H['k'], ghat(H['k']), label="$\hat{g}(k)$")
ax[1].legend(loc='best', fontsize=15)
ax[1].set_xlabel('k', fontsize=15)
ax[1].set_xlim(0, 20)

ax[0].grid(True)
ax[1].grid(True)
```



Utility Functions

[22]:
```
def plot_comparison(x, y, ylabel, comp_ylim=None, F=F, H=H, **subplot_kw):
    fnc = globals()[y]

    subplot_kw.update({"xscale":'log'})

    fig, ax = plt.subplots(
        2,1, sharex=True, subplot_kw=subplot_kw,
        figsize=(12, 7), gridspec_kw={"hspace":0.07})

    ax[0].plot(H[x], fnc(H[x]), linewidth=2, label="Analytic", color="C2")
    ax[0].plot(H[x], H[y],linewidth=2, ls="--", label="Hankel")
    ax[0].plot(F[x], F[y],linewidth=2, ls="--", label="Fourier")

    ax[0].legend(fontsize=14)
    ax[0].grid(True)
    ax[0].set_ylabel(ylabel, fontsize=15)

    ax[1].plot(H[x], np.abs(H[y] - fnc(H[x])))
    ax[1].plot(F[x], np.abs(F[y] - fnc(F[x])))
    ax[1].grid(True)
    ax[1].set_ylabel("Residual", fontsize=15)
    ax[1].set_xlabel(x, fontsize=15)
    ax[1].set_yscale('log')

    if comp_ylim:
        ax[1].set_ylim(comp_ylim)
```

Example 1: Compute Laplacian

1. Forward Transform

a. Hankel

[195]:
```
# Compute Hankel transform

H['fhat'] = ht.transform(f, H['k'], ret_err=False)        # Return the transform of f␣
↪at k.
```

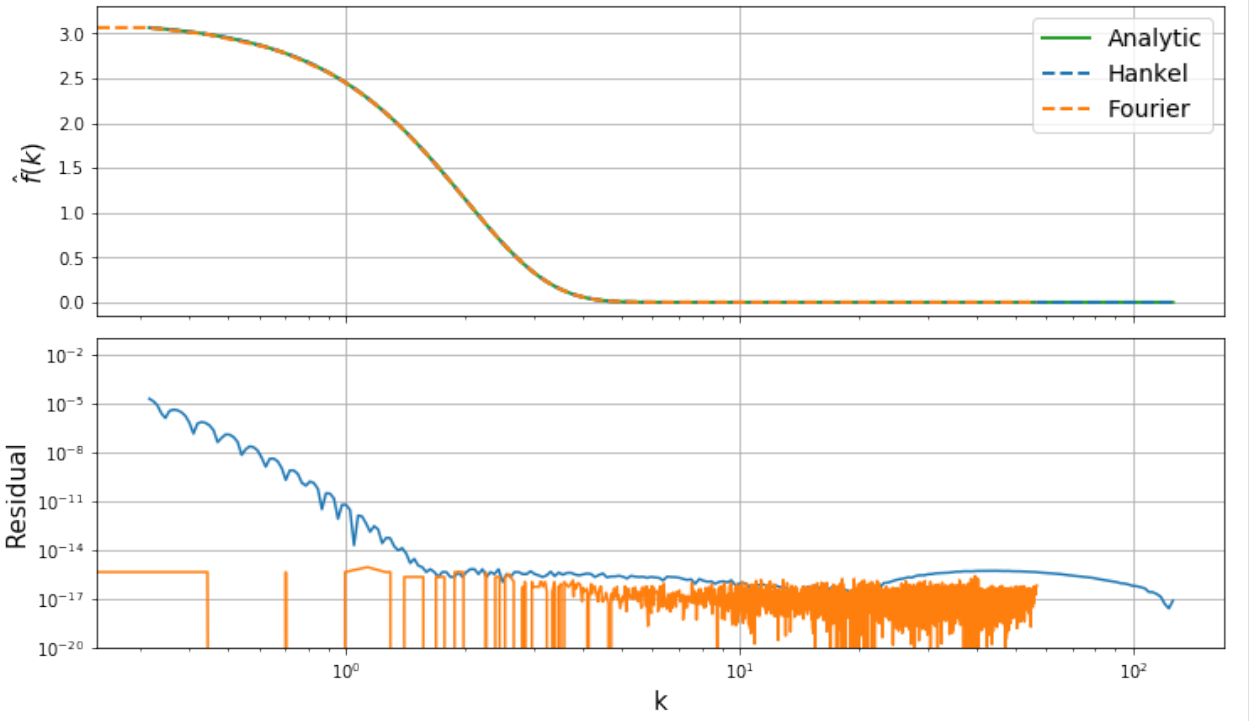
b. Fourier

[196]:
```
F['fhat_grid'], F['kgrid'], F['k'], F['fhat'] = ft2d(F['fgrid'], 2*L)
```

c. Comparison

Comparing the results of the Hankel and Fourier Transforms, we have

[197]:
```
plot_comparison('k', 'fhat', "$\hat{f}(k)$", comp_ylim=(1e-20, 1e-1))
```



2. Inverse Transform

a. Hankel

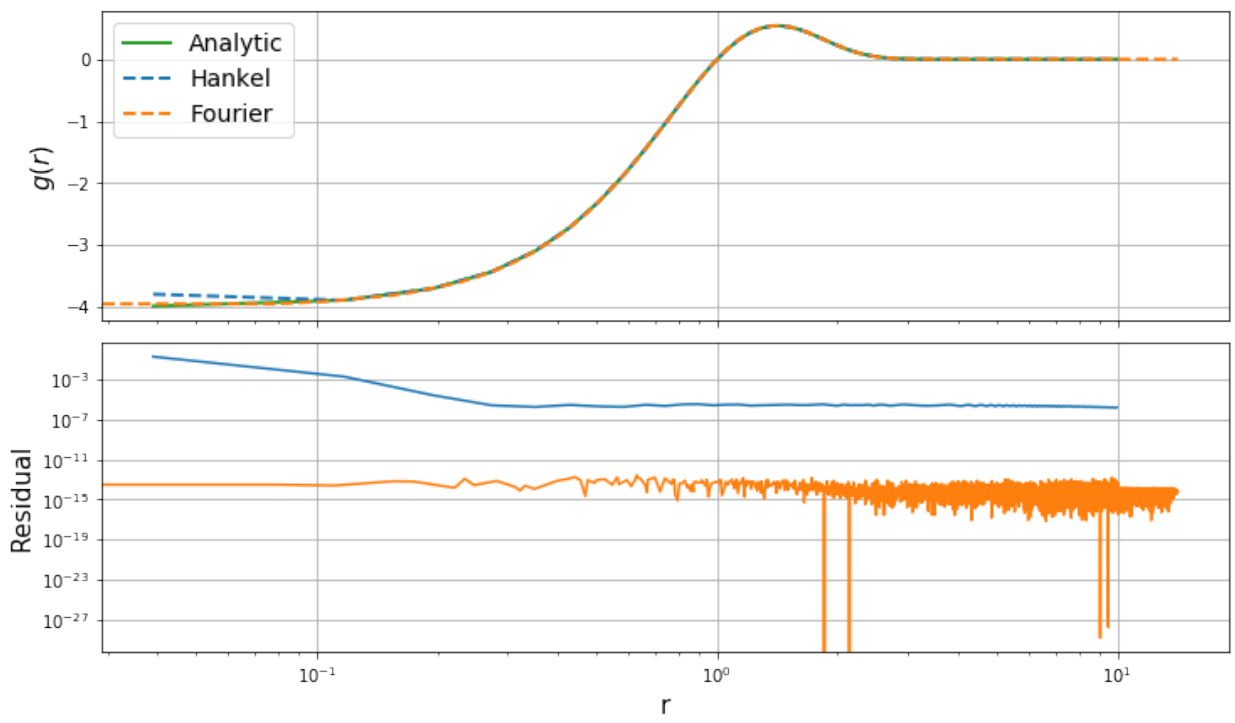
[198]:
```
# Build a spline to approximate ghat

H['ghat'] =  -H['k']**2 * H['fhat']

# We can build the spline carefully since we know something about the function
spl = spline(np.log(H['k'][H['ghat']!=0]), np.log(np.abs(H['ghat'][H['ghat']!=0])),␣
↪k=2)
_fnc = lambda k: np.where(k < 1e2, -np.exp(spl(np.log(k))), 0)

# Do the transform
H['g'] = ht.transform(_fnc, H['r'], ret_err=False, inverse=True)
```

b. Fourier

[199]:
```
F['ghat'] =  -F['kgrid']**2 *F['fhat_grid']
F['ggrid'], _, F['r'], F['g'] = ft2d(F['ghat'], 2*L, inverse=True)
```

c. Compare

[200]:
```
plot_comparison('r', 'g', "$g(r)$")
```



Example 2: Invert Laplacian

We use the 1D Hankel (or 2D Fourier) transform to compute the inverse Laplacian in three steps: 1. Compute the Forward Transform

```
$$
\mathcal{H}[g(r)] = \hat g(k)
$$
```

2. Differentiate in Spectral space

$$\hat{f}(k) = -\frac{1}{k^2}\hat{g}(k)$$

3. Compute the Inverse Transform

$$f(r) = \mathcal{H}^{-1}[\hat{f}(k)]$$

1. Forward Transform

a. Hankel

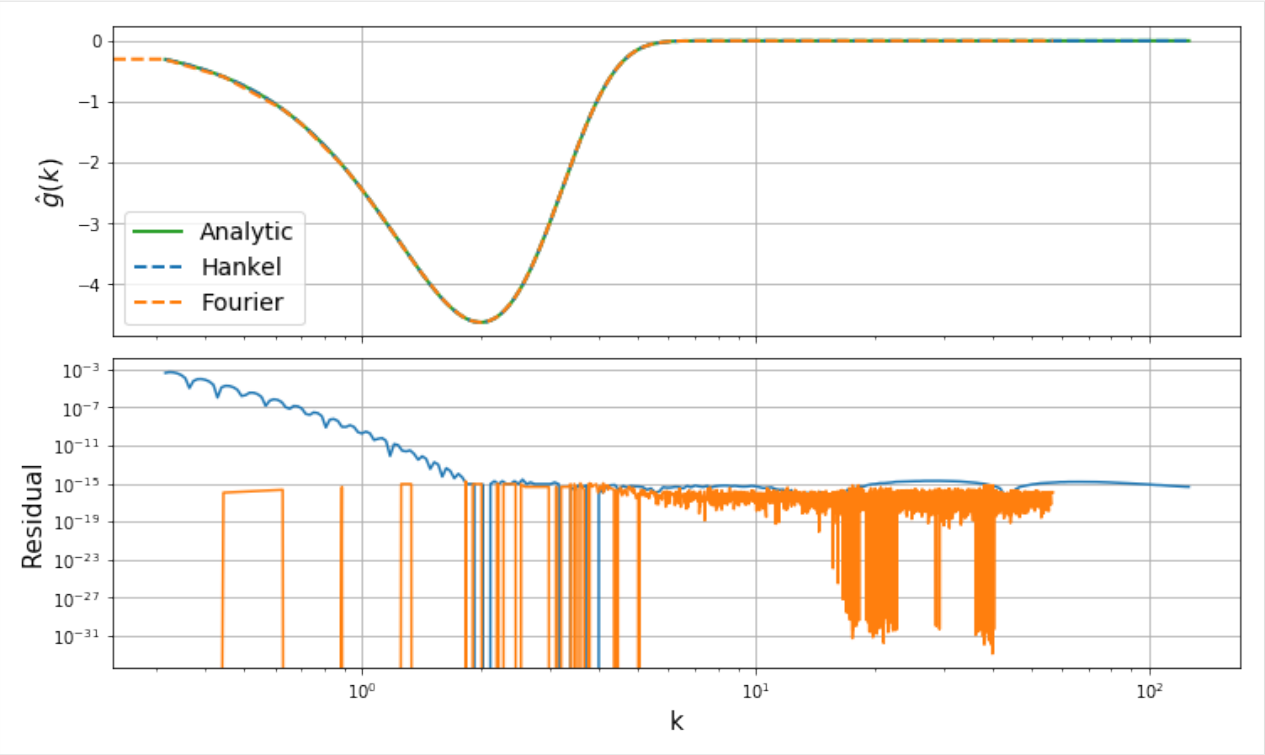
[201]:
```
H['ghat'] = ht.transform(g, H['k'], ret_err=False)          # Return the transform of g␣
↪at k.
```

b. Fourier

[202]:
```
# Compute Fourier Transform

F['ghat_grid'], _, _, F['ghat'] = ft2d(g(F['rgrid']), 2*L)
```

c. Compare

[203]:
```
plot_comparison('k', 'ghat', "$\hat{g}(k)$")
```



2. Inverse Transform

a. Hankel

[204]:
```
# Interpolate onto a spline

spl_inv = spline(H['k'], -H['ghat']/H['k']**2)
H['f'] = ht.transform(spl_inv, H['r'], ret_err=False, inverse=True)
```

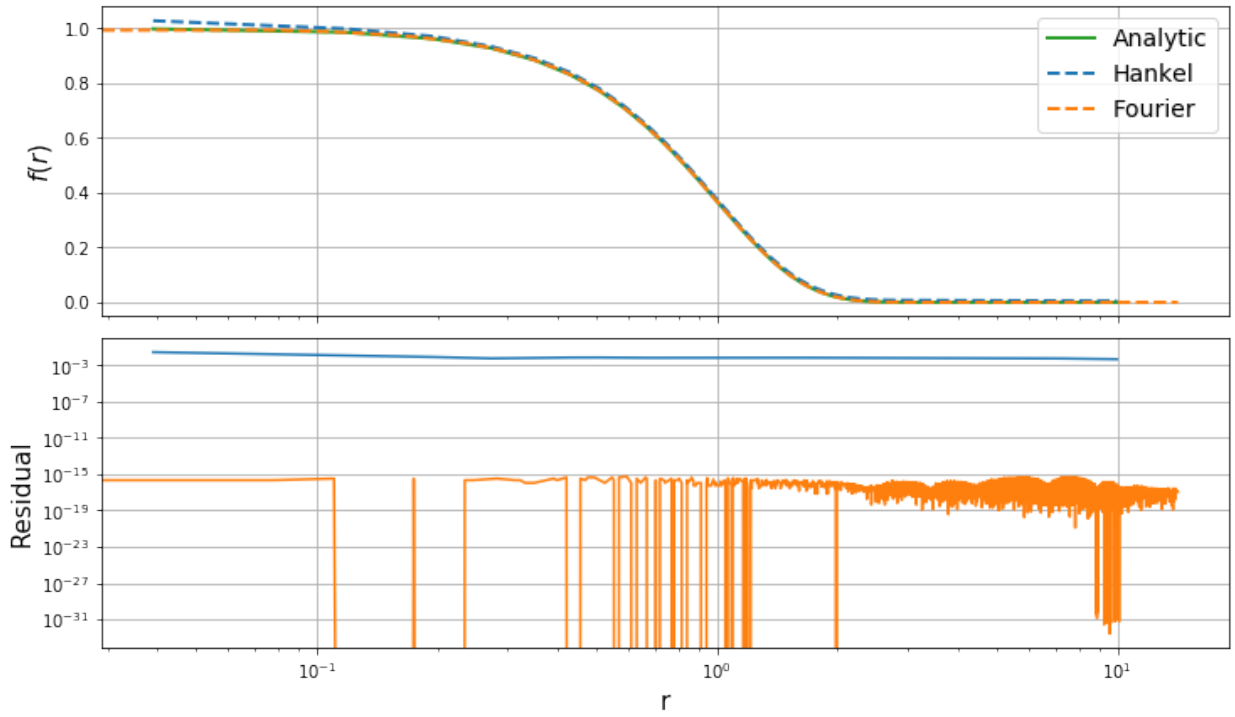
b. Fourier

[205]:
```
# Differentiate in spectral space

F['fhat'] =  -F['ghat_grid']/ F['kgrid']**2
F['fhat'][np.isnan(F['fhat'])] = np.pi # This is the limiting value at k=0.

F['fgrid'], _, F['r'], F['f'] = ft2d(F['fhat'], 2*L, inverse=True)
```

```
/home/steven/miniconda3/envs/hankel/lib/python3.7/site-packages/ipykernel/__main__.
↪py:3: RuntimeWarning: divide by zero encountered in true_divide
  app.launch_new_instance()
/home/steven/miniconda3/envs/hankel/lib/python3.7/site-packages/ipykernel/__main__.
↪py:3: RuntimeWarning: invalid value encountered in true_divide
  app.launch_new_instance()
```

c. Compare

[206]:
```
plot_comparison('r', 'f', "$f(r)$")
```



Example 3: Laplacian in 3D

Generalising the previous calculations slightly, we can perform the Laplacian transformation in 3D using the Hankel transform correspondin to a circularly-symmetric FT in 3-dimensions (see the "Getting Started" notebook for details). Analytically, we obtain

$$\hat{f}(k) = \pi^{3/2} e^{-k^2/4} \tag{5.15}$$

$$\hat{g}(k) = -k^2 \pi^{3/2} e^{-k^2/4} \tag{5.16}$$

$$g(r) = 4e^{-r^2}(r^2 - 1.5) \tag{5.17}$$

$$\tag{5.18}$$

Analytic Functions

[6]:
```
g3 = lambda r: 4.0*np.exp(-r**2)*(r**2 - 1.5)
fhat3 = lambda k : np.pi**(3./2) * np.exp(-k**2/4)
ghat3 = lambda k : -k**2 * fhat3(k)
```

Grid Setup

[2]:
```
L     = 10.0    # Size of box for transform
N     = 128     # Grid size
b0    = 1.0
```

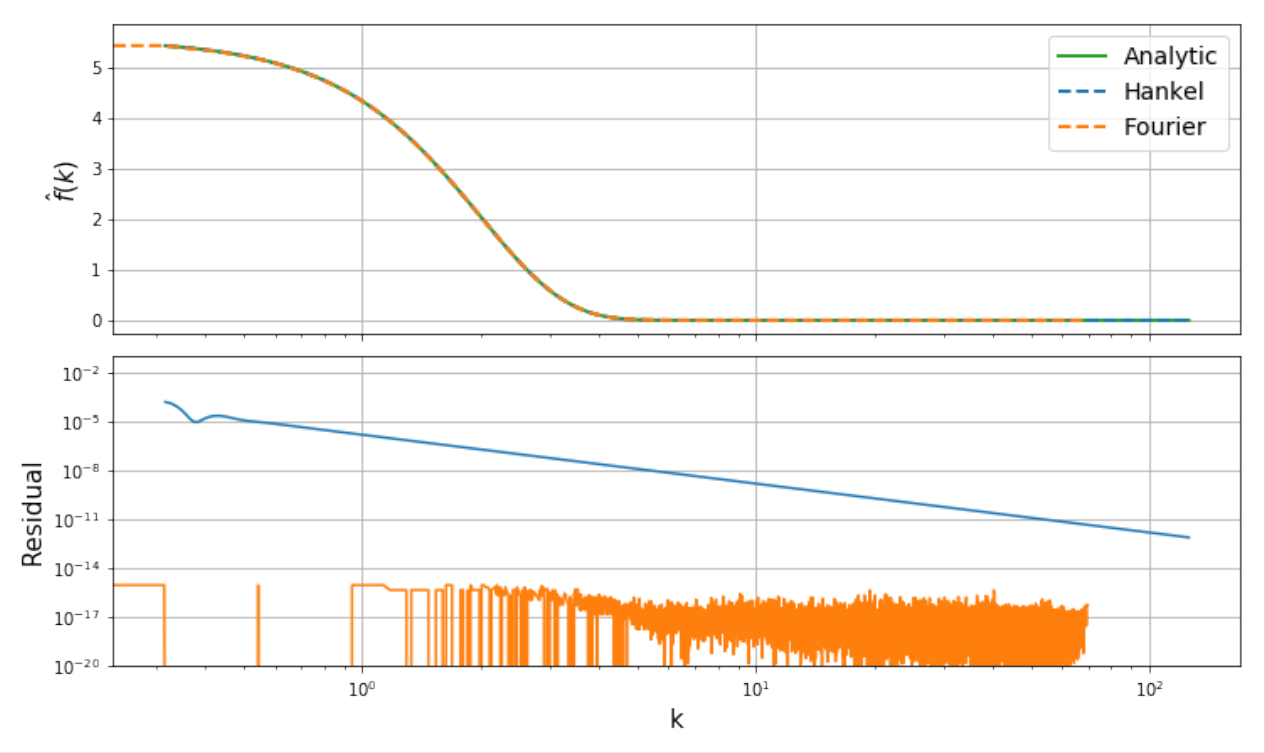
[7]:
```
# Create the Hankel Transform object

ht = SymmetricFourierTransform(ndim=3, h=0.005)
```

[11]:
```
dr = L/N

# Create persistent dictionaries to track Hankel and Fourier results throughout.
H3 = {} # Hankel dict
F3 = {} # Fourier dict

r = np.linspace(dr/2, L-dr/2, N)

# 1D Grid for Hankel
H3['r'] = r

# To decide on k values to use, we need to know the scales
# we'll require to evaluate on the *inverse* transform:
H3['k'] = np.logspace(-0.5, 2.1, 2*N)       # k-space is rather arbitrary.

# 2D Grid for Fourier
F3['x'] = np.arange(-N, N)*dr
F3['rgrid'] = np.sqrt(np.add.outer(F3['x']**2, np.add.outer(F3['x']**2, F3['x']**2))).
↪reshape((len(F3['x']),)*3)
F3['fgrid'] = f(F3['rgrid'])
```

Forward

[24]:
```
H3['fhat3'] = ht.transform(f, H3['k'], ret_err=False)
F3['fhat_grid'], F3['kgrid'], F3['k'], F3['fhat3'] = ft2d(F3['fgrid'], 2*L)
```

[25]:
```
plot_comparison('k', 'fhat3', "$\hat{f}(k)$", comp_ylim=(1e-20, 1e-1), F=F3, H=H3)
```

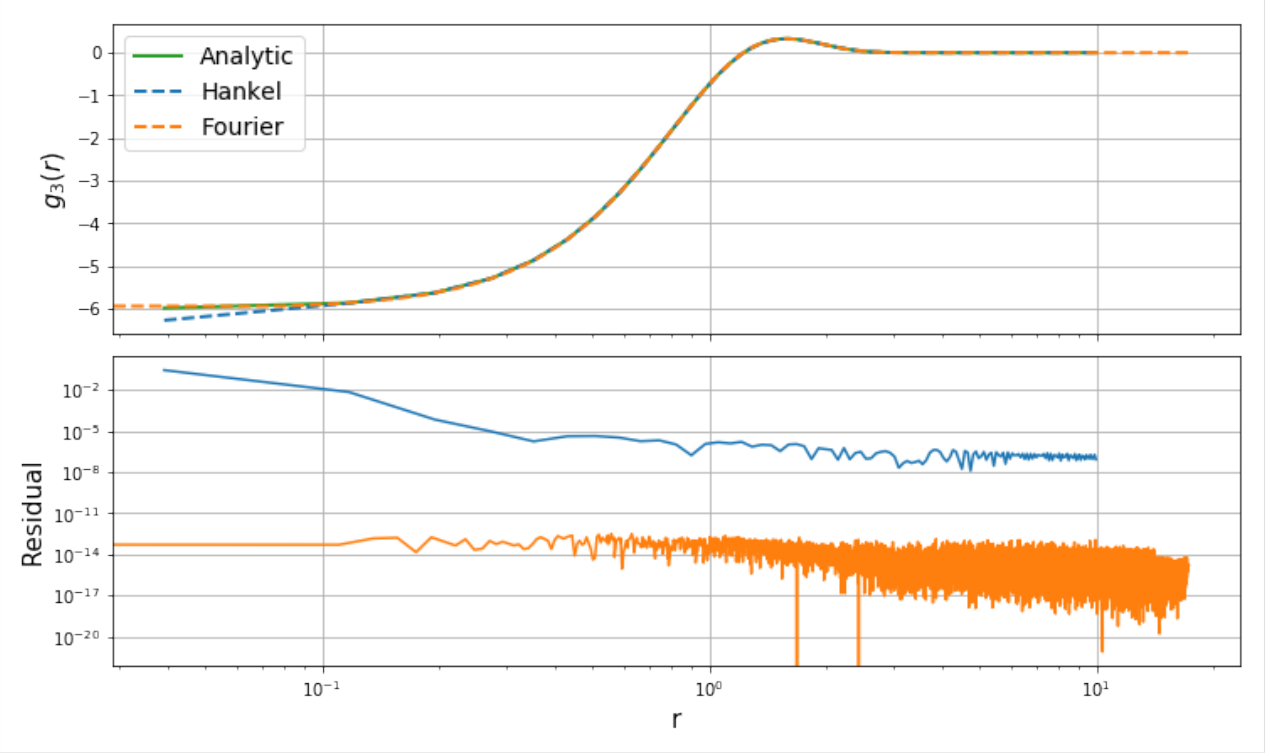


Backward

[28]:
```
# Build a spline to approximate ghat

H3['ghat3'] =  -H3['k']**2 * H3['fhat3']

# We can build the spline carefully since we know something about the function
spl = spline(np.log(H3['k'][H3['ghat3']!=0]), np.log(np.abs(H3['ghat3'][H3['ghat3']!
↪=0])), k=2)
_fnc = lambda k: np.where(k < 1e2, -np.exp(spl(np.log(k))), 0)

# Do the transform
H3['g3'] = ht.transform(_fnc, H3['r'], ret_err=False, inverse=True)
```

[34]:
```
F3['ghat3'] =  -F3['kgrid']**2 *F3['fhat_grid']
F3['ggrid'], _, F3['r'], F3['g3'] = ft2d(F3['ghat3'], 2*L, inverse=True)
```

[35]:
```
plot_comparison('r', 'g3', "$g_3(r)$", H=H3, F=F3)
```



n-Dimensional Transform

Generalising again, the n-dimensionsal transform functions can be written

$$\hat{f}(k) = \pi^{n/2} e^{-k^2/4} \tag{5.19}$$

$$\hat{g}(k) = -k^2 \pi^{n/2} e^{-k^2/4} \tag{5.20}$$

$$g(r) = 2e^{-r^2}(2r^2 - n) \tag{5.21}$$

$$\tag{5.22}$$

In this section, we show the performance (both in speed and accuracy) of the Hankel transform algorithm to produce $g(r)$ for arbitrary dimensions. We do not use the FT-method in this section as its memory consumption obviously scales as N^n, and its CPU performance as $nN^n \log N$.

[40]:
```
gn = lambda r, n: 2* np.exp(-r**2) * (2*r**2 - n)
```

[37]:
```
def gn_hankel(r, n, h=0.005):
    ht = SymmetricFourierTransform(ndim=n, h=h)
    k = np.logspace(-0.5, 2.1, 2*N)

    fhat = ht.transform(f, k, ret_err=False)

    ghat =  -k**2 * fhat

    # We can build the spline carefully since we know something about the function
    spl = spline(np.log(k[ghat!=0]), np.log(np.abs(ghat[ghat!=0])), k=2)
    _fnc = lambda k: np.where(k < 1e2, -np.exp(spl(np.log(k))), 0)
```

(continues on next page)

(continued from previous page)

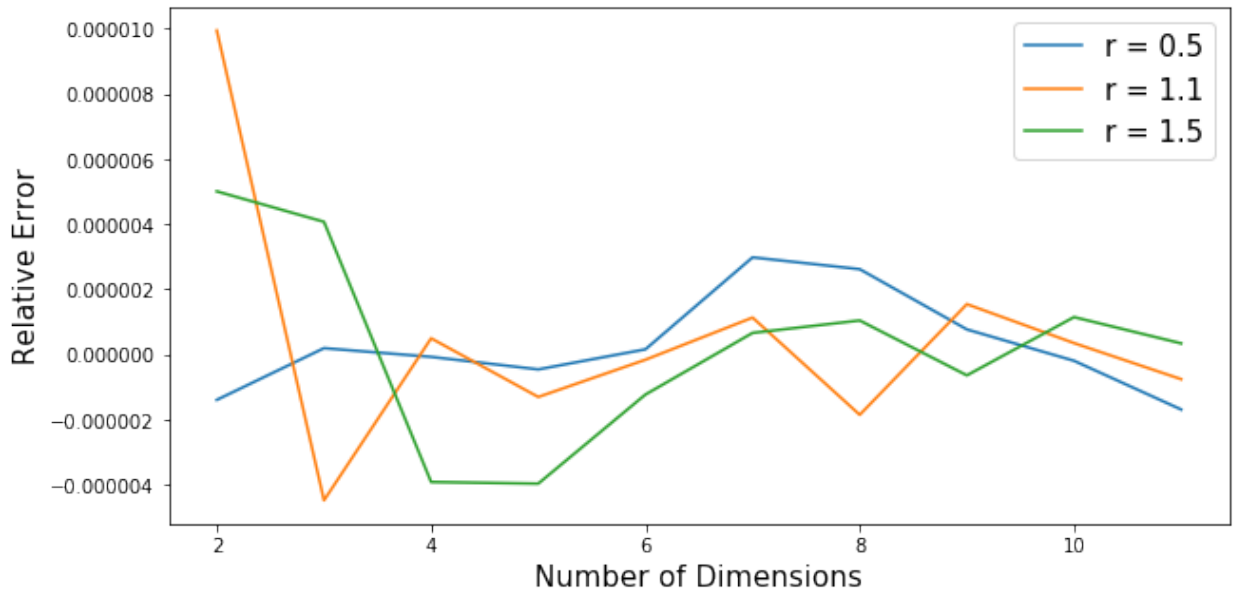
```
    # Do the transform
    return ht.transform(_fnc, r, ret_err=False, inverse=True)
```

[49]:
```
rs = np.array([0.5, 1.1, 1.5])
ns = np.arange(2, 12)
```

[50]:
```
transform_at_r1 = np.array([gn_hankel(rs, n) for n in ns])    # Get numerical answers
```

[56]:
```
transform_at_r1_anl = np.array([gn(rs, n) for n in ns])       # Get analytic answers
```

[62]:
```
plt.figure(figsize=(10,5))
for i, r in enumerate(rs):
    plt.plot(ns, transform_at_r1[:, i]/ transform_at_r1_anl[:, i] - 1, label="r = %s"
↪%r)

plt.legend(fontsize=15)
plt.xlabel("Number of Dimensions", fontsize=15)
plt.ylabel("Relative Error", fontsize=15);
```



Let's get the timing information for each calculation:

[82]:
```
times = [0]*len(ns)
for i, n in enumerate(ns):
    times[i] = %timeit -o gn_hankel(rs, n);
```

```
9.86 ms ± 489 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
7.91 ms ± 145 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
11.2 ms ± 507 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
327 ms ± 6.09 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
11 ms ± 90.1 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
324 ms ± 4.82 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
11.1 ms ± 104 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
329 ms ± 2.01 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

(continues on next page)

(continued from previous page)

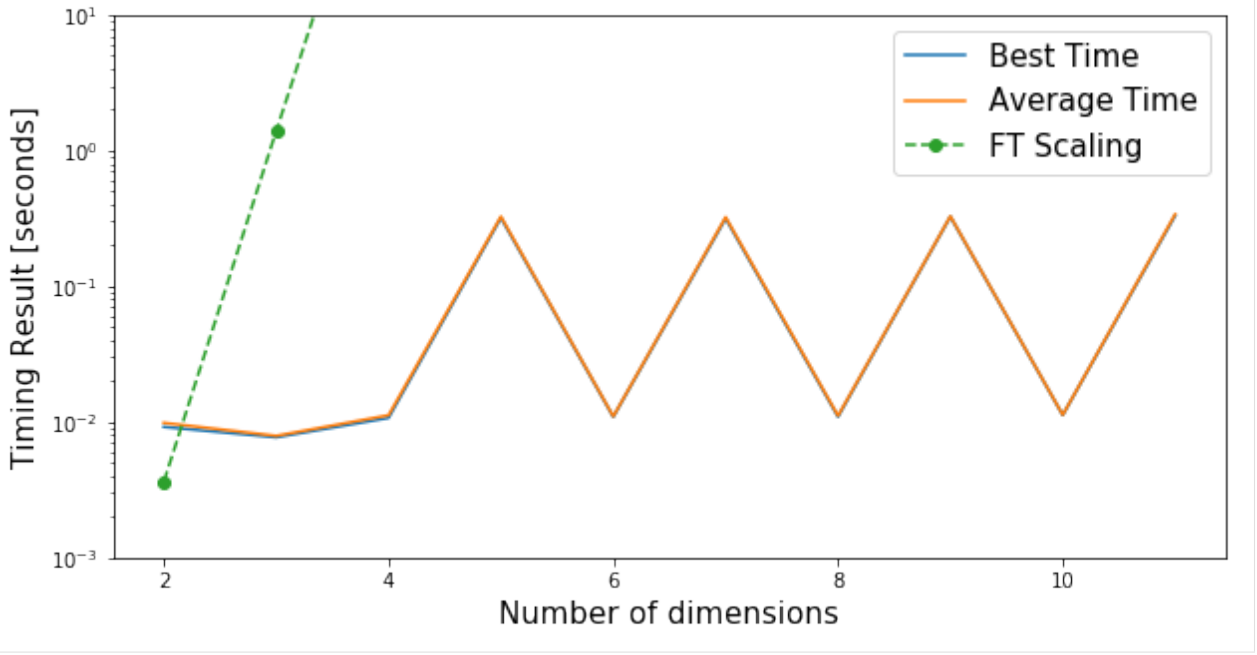
```
11.3 ms ± 34 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
339 ms ± 8.27 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Here's a plot of the timing results. Note how odd-dimensions result in longer run-times, as the Bessel functions are 1/2-order, which requires more sophisticated methods from `mpmath` to calculate. There is no overall progression of the computational load with dimensionality, however.

This is in contrast to a FT approach, which scales as N^n, shown in green.

[100]:
```
plt.figure(figsize=(10, 5))

plt.plot(ns, [t.best for t in times], label="Best Time")
plt.plot(ns, [t.average for t in times], label='Average Time')
plt.plot(ns[:6], 5e-9 * ns[:6] * (2*N)**ns[:6] * np.log(2*N), label="FT Scaling", ls =
↪'--', marker='o')
plt.xlabel("Number of dimensions", fontsize=15)
plt.ylabel("Timing Result [seconds]", fontsize=15)
plt.ylim(1e-3, 10)
plt.yscale('log')
plt.legend(fontsize=15);
```

[100]:
```
<matplotlib.legend.Legend at 0x7fb3df123d30>
```



5.2 License

5.3 Changelog

5.3.1 dev

Bugfixes

- Fixed SymmetricFourierTransform default N to be dynamic, like HankelTransform.

5.3.2 v0.3.8 [08 Jan 2019]

Enhancements

- Swapped to pytest from nose for all testing
- Removed warnings of overflow for cosh and sinh functions
- Improved documentation in many areas.
- Swapped out default 3.2/h for pi/h, which sounds way cooler :-)

Bugfixes

- Fixed an error in tests such that the wrong analytic function was being tested.
- Removed a check for deltaG in get_h, which sometimes fails spectacularly.

5.3.3 v0.3.7 [11 Dec 2018]

Bugfixes

- Fixed issue in Py3 in which dim=2 would not run the fast function (thanks @MuellerSeb)

5.3.4 v0.3.6 [6 Aug 2018]

Enhancements

- Added support for transforms with `nu=-0.5` (thanks @MuellerSeb)

5.3.5 v0.3.5 [8 Dec 2017]

Bugfixes

- Fixed Python 3 support from v0.3.4

5.3.6 v0.3.4 [28 July 2017]

Features

- Added get_h function to aide in determining optimal h value for a given transformation.

Enhancements

- Added _get_series method to quickly retrieve the summed series for the integration.
- Two updated notebook examples.

Bugfixes

- Moved setting of N to avoid error.

5.3.7 v0.3.3 [28 July 2017]

Features

- Some additional tools to determine accuracy – quick calculation of last term in sum, and evaluated range.

Enhancements

- Default setting of N=3.2/h, which is the maximum possible N that should be chosen, as above this, the series truncates due to the double-exponential convergence to the roots of the Bessel function.

Bugfixes

- Fixed error in cumulative sum when k is not scalar.

5.3.8 v0.3.2 [12 July 2017]

Enhancements

- Documentation! See it at https://hankel.readthedocs.io
- Two new jupyter notebook demos (find them in the docs) by @francispoulin

Bugfixes

- Fixed relative import in Python 3 (tests now passing), thanks to @louity
- Fixed docstring of SymmetricFourierTransform to have correct Fourier convention equation
- Fixed bug in choosing alternative conventions in which the fourier-dual variable was unchanged.

5.3.9 v0.3.1 [5 Jan 2017]

Bugfixes

- Fixed normalisation for inverse transform in `SymmetricFourierTransform`.

Features

- Ability to set Fourier conventions arbitrarily in `SymmetricFourierTransform`.

5.3.10 v0.3.0 [4 Jan 2017]

Features

- New class *SymmetricFourierTransform* which makes it incredibly easy to do arbitrary *n*-dimensional fourier transforms when the function is radially symmetric (includes inverse transform).
- Addition of *integrate* method to base class to perform Hankel-type integrals, which were previously handled by the *transform* method. This latter method is now used for actual Hankel transforms.
- Documentation!

Enhancements

- Addition of many tests against known integrals.
- Continuous integration
- Restructuring of package for further flexibility in the future.
- Quicker zero-finding of 1/2-order bessel functions.
- This changelog.
- Some notebooks in the devel/ directory which show how various integrals/transforms behave under different choices of integration steps.

5.3.11 v0.2.2 [29 April 2016]

Enhancements

- Compatibility with Python 3 (thanks to @diazona)
- Can now use with array-value functions (thanks to @diazona)

5.3.12 v0.2.1 [18 Feb 2016]

Bugfixes

- Fixed pip install by changing readme –> README

Enhancements

- updated docs to show dependence on mpmath

5.3.13 v0.2.0 [10 Sep 2014]

Features

- Non-integer orders supported through mpmath.

5.3.14 v0.1.0

- First working version. Only integer orders (and 1/2) supported.

5.4 Authors

Steven Murray: @steven-murray

5.4.1 Contributors

- Francis Poulin @francispoulin
- Sebastian Mueller @MuellerSeb

5.5 hankel API Summary

General quadrature method for Hankel transformations.

Based on the algorithm provided in H. Ogata, A Numerical Integration Formula Based on the Bessel Functions, Publications of the Research Institute for Mathematical Sciences, vol. 41, no. 4, pp. 949-970, 2005.

5.5.1 Classes

HankelTransform([nu, N, h])	The basis of the Hankel Transformation algorithm by Ogata 2005.
SymmetricFourierTransform([ndim, a, b, N, h])	Determine the Fourier Transform of a radially symmetric function in arbitrary dimensions.

5.5.2 Functions

get_h(f, nu[, K, cls, hstart, hdecrement, ...])	Determine the largest value of h which gives a converged solution.

class hankel.**HankelTransform**(*nu=0*, *N=None*, *h=0.05*)

The basis of the Hankel Transformation algorithm by Ogata 2005.

This algorithm is used to solve the equation $\int_0^\infty f(x)J_\nu(x)dx$ where $J_\nu(x)$ is a Bessel function of the first kind of order ν, and $f(x)$ is an arbitrary (slowly-decaying) function.

The algorithm is presented in H. Ogata, A Numerical Integration Formula Based on the Bessel Functions, Publications of the Research Institute for Mathematical Sciences, vol. 41, no. 4, pp. 949-970, 2005.

This class provides a method for directly performing this integration, and also for doing a Hankel Transform.

Parameters

nu [scalar, optional] The order of the bessel function (of the first kind) J_nu(x)

N [int, optional, default = `pi/h`] The number of nodes in the calculation. Generally this must increase for a smaller value of the step-size h. Default value is based on where the series

will truncate according to the double-exponential convergence to the roots of the Bessel function.

h [float, optional] The step-size of the integration.

classmethod G(*f*, *h*, *k=None*, **args*, ***kwargs*)
The absolute value of the non-oscillatory of the summed series' last term, up to a scaling constant.

This can be used to get the sign of the slope of G with h.

Parameters

f [callable] The function to integrate/transform

h [float] The resolution parameter of the hankel integration

k [float or array-like, optional] The scale at which to evaluate the transform. If None, assume an integral.

Returns

The value of G.

classmethod deltaG(*f*, *h*, **args*, ***kwargs*)
The slope (up to a constant) of the last term of the series with h

integrate(*f*, *ret_err=True*, *ret_cumsum=False*)
Do the Hankel-type integral of the function f.

This is *not* the Hankel transform, but rather the simplified integral, $\int_0^\infty f(x)J_\nu(x)dx$, equivalent to the transform of $f(r)/r$ at *k=1*.

Parameters

f [callable] A function of one variable, representing $f(x)$

ret_err [boolean, optional, default = True] Whether to return the estimated error

ret_cumsum [boolean, optional, default = False] Whether to return the cumulative sum

transform(*f*, *k=1*, *ret_err=True*, *ret_cumsum=False*, *inverse=False*)
Do the Hankel-transform of the function f.

Parameters

f [callable] A function of one variable, representing $f(x)$

ret_err [boolean, optional, default = True] Whether to return the estimated error

ret_cumsum [boolean, optional, default = False] Whether to return the cumulative sum

Returns

ret [array-like] The Hankel-transform of f(x) at the provided k. If *k* is scalar, then this will be scalar.

err [array-like] The estimated error of the approximate integral, at every *k*. It is merely the last term in the sum. Only returned if *ret_err=True*.

cumsum [array-like] The total cumulative sum, for which the last term is itself the transform. One can use this to check whether the integral is converging. Only returned if *ret_cumsum=True*

Notes

The Hankel transform is defined as

$$F(k) = \int_0^\infty r f(r) J_\nu(kr) dr.$$

The inverse transform is identical (swapping *k* and *r* of course).

xrange(*k=1*)
Tuple giving (min,max) x value evaluated by f(x).

Parameters

k [array-like, optional] Scales for the transformation. Leave as 1 for an integral.

See also:

See meth:*xrange_approx* for an approximate version of this method

which

classmethod xrange_approx(*h*, *nu*, *k=1*)
Tuple giving approximate (min,max) x value evaluated by f(x/k).

Operates under the assumption that N = 3.2/h.

Parameters

h [float] The resolution parameter of the Hankel integration

nu [float] Order of the integration/transform

k [array-like, optional] Scales for the transformation. Leave as 1 for an integral.

See also:

xrange the actual x-range under a given choice of parameters.

class hankel.**SymmetricFourierTransform**(*ndim=2*, *a=1*, *b=1*, *N=None*, *h=0.05*)
Determine the Fourier Transform of a radially symmetric function in arbitrary dimensions.

Parameters

ndim [int] Number of dimensions the transform is in.

a, b [float, default 1] This pair of values defines the Fourier convention used (see Notes below for details)

N [int, optional] The number of nodes in the calculation. Generally this must increase for a smaller value of the step-size h.

h [float, optional] The step-size of the integration.

Notes

We allow for arbitrary Fourier convention, according to the scheme in http://mathworld.wolfram.com/FourierTransform.html. That is, we define the forward and inverse *n*-dimensional transforms respectively as

$$F(k) = \sqrt{\frac{|b|}{(2\pi)^{1-a}}}^n \int f(r) e^{ib\mathbf{k}\cdot\mathbf{r}} d^n\mathbf{r}$$

and

$$f(r) = \sqrt{\frac{|b|}{(2\pi)^{1+a}}}^n \int F(k) e^{-ib\mathbf{k}\cdot\mathbf{r}} d^n\mathbf{k}.$$

By default, we set both *a* and *b* to 1, so that the forward transform has a normalisation of unity.

In this general sense, the forward and inverse Hankel transforms are respectively

$$F(k) = \sqrt{\frac{|b|}{(2\pi)^{1-a}}}^n \frac{(2\pi)^{n/2}}{(bk)^{n/2-1}} \int_0^\infty r^{n/2-1} f(r) J_{n/2-1}(bkr) r dr$$

and

$$f(r) = \sqrt{\frac{|b|}{(2\pi)^{1+a}}}^n \frac{(2\pi)^{n/2}}{(br)^{n/2-1}} \int_0^\infty k^{n/2-1} f(k) J_{n/2-1}(bkr) k dk.$$

classmethod G(*f*, *h*, *k=None*, *ndim=2*)

The absolute value of the non-oscillatory part of the summed series' last term, up to a scaling constant.

This can be used to get the sign of the slope of G with h.

Parameters

f [callable] The function to integrate/transform

h [float] The resolution parameter of the hankel integration

k [float or array-like, optional] The scale at which to evaluate the transform. If None, assume an integral.

ndim [float] The number of dimensions of the transform

Returns

The value of G.

transform(*f*, *k*, **args*, ***kwargs*)

Do the *n*-symmetric Fourier transform of the function f.

Parameters and returns are precisely the same as *HankelTransform.transform()*.

Notes

The *n*-symmetric fourier transform is defined in terms of the Hankel transform as

$$F(k) = \frac{(2\pi)^{n/2}}{k^{n/2-1}} \int_0^\infty r^{n/2-1} f(r) J_{n/2-1}(kr) r dr.$$

The inverse transform has an inverse normalisation.

classmethod xrange_approx(*h*, *ndim*, *k=1*)

Tuple giving approximate (min,max) x value evaluated by f(x/k).

Operates under the assumption that N = pi/h.

Parameters

h [float] The resolution parameter of the Hankel integration

ndim [float] Number of dimensions of the transform.

k [array-like, optional] Scales for the transformation. Leave as 1 for an integral.

See also:

xrange the actual x-range under a given choice of parameters.

hankel.**get_h**(*f*, *nu*, *K=None*, *cls=<class 'hankel.hankel.HankelTransform'>*, *hstart=0.05*, *hdecrement=2*, *atol=0.001*, *rtol=0.001*, *maxiter=15*, *inverse=False*)

Determine the largest value of h which gives a converged solution.

Parameters

f [callable] The function to be integrated/transformed.

nu [float] Either the order of the transformation, or the number of dimensions (if *cls* is a *SymmetricFourierTransform*)

K [float or array-like, optional] The scale(s) of the transformation. If None, assumes an integration over f(x)J_nu(x) is desired. It is recommended to use a down-sampled K for this routine for efficiency. Often a min/max is enough.

cls [*HankelTransform* subclass, optional] Either *HankelTransform* or a subclass, specifying the type of transformation to do on *f*.

hstart [float, optional] The starting value of h.

hdecrement [float, optional] How much to divide h by on each iteration.

atol, rtol [float, optional] The tolerance parameters, passed to *np.isclose*, defining the stopping condition.

maxiter [int, optional] Maximum number of iterations to perform.

inverse [bool, optional] Whether to treat as an inverse transformation.

Returns

h [float] The h value at which the solution converges.

res [scalar or tuple] The value of the integral/transformation using the returned h – if a transformation, returns results at K.

N [int] The number of nodes necessary in the final calculation. While each iteration uses N=pi/h, the returned N checks whether nodes are numerically zero above some threshold.

Notes

This function is not completely general. The function *f* is assumed to be reasonably smooth and non-oscillatory.

The idea is to use successively smaller values of *h*, with N=pi/h on each iteration, until the result betweeniterations becomes stable.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

h

Index